第3章 分治法

分治法是使用最广泛的算法设计方法之一。其基本策略是采用递归思想把大问题分解成一些小问题，然后由小问题的解方便地构造出大问题的解。本章介绍分治法求解问题的一般方法，并给出一些用分治法求解的经典示例。

3.1 分治法概述

3.1.1 分治法的设计思想

对于一个规模为 n 的问题，若该问题可以容易地解决（例如规模 n 较小）则直接解决，否则将其分解为 k 个规模较小的子问题，这些子问题互相独立且与原问题形式相同，递归地解这些子问题，然后将各子问题的解合并得到原问题的解，这种算法设计策略叫**分治法**。

如果原问题可分割成 $k(1<k\leqslant n)$ 个子问题，且这些子问题都可解并可利用这些子问题的解求出原问题的解，那么这种分治法就是可行的。由分治法产生的子问题往往是原问题的较小模式，这就为使用递归技术提供了方便。在这种情况下，反复应用分治手段可以使子问题与原问题类型一致而其规模却不断缩小，最终使子问题缩小到很容易直接求出其解，这自然导致递归过程的产生。分治与递归像一对孪生兄弟，经常同时应用在算法设计之中，并由此产生许多高效算法。

分治法所能解决的问题一般具有以下几个特征：

（1）该问题的规模缩小到一定的程度就可以容易地解决。

（2）该问题可以分解为若干个规模较小的相似问题。

（3）利用该问题分解出的子问题的解可以合并为该问题的解。

（4）该问题所分解出的各个子问题是相互独立的，即子问题之间不包含公共的子问题。

上述特征(1)是绝大多数问题都可以满足的，因为问题的计算复杂性一般是随着问题规模的增加而增加；特征(2)是应用分治法的前提，它也是大多数问题可以满足的，此特征反映了递归思想的应用；特征(3)是关键，能否利用分治法完全取决于问题是否具有该特征，如果具备了特征(1)和(2)，而不具备特征(3)，则可以考虑用贪心法或动态规划法；特征(4)涉及分治法的效率，如果各子问题是不独立的则分治法要做许多不必要的工作，重复地解公共的子问题，此时虽然可用分治法，但一般用动态规划法较好。

从上看到，分治是一种解题的策略，它的基本思想是“如果整个问题比较复杂，可以将问题分化，各个击破”。分治包含“分”和“治”两层含义，如何分，分后如何治成为解决问题的关键所在。不是所有的问题都可以采用分治，只有那些能将问题分成与原问题类似的子问题并且归并后符合原问题的性质的问题才能进行分治。分治可进行二分、三分等，具体怎么分，需看问题的性质和分治后的效果。只有深刻地领会分治的思想，认真分析分治后可能产生的预期效率，才能灵活地运用分治思想解决实际问题。

3.1.2 分治法的求解过程

递归特别适合解决结构自相似的问题，所谓结构自相似，是指构成原问题的子问题与原问题在结构上相似，可以采用类似的方法解决。所以分治法通常采用递归算法设计技术，在每一层递归上都有 3 个步骤。

（1）分解成若干个子问题：将原问题分解为若干个规模较小、相互独立、与原问题形式相同的子问题。

（2）求解子问题：若子问题规模较小，容易被解决，则直接求解，否则递归地求解各个子问题。

（3）合并子问题：将各个子问题的解合并为原问题的解。

分治法的一般算法设计模式如下：

```
divide-and-conquer(P)
{   if |P|≤n0 return adhoc(P);
    将 P 分解为较小的子问题 P1、P2、…、Pk;
    for(i=1;i<=k;i++)                    //循环处理 k 次
        yi=divide-and-conquer(Pi);       //递归解决 Pi
    return merge(y1,y2,…,yk);            //合并子问题
}
```

其中，$|P|$表示问题 P 的规模；n_0 为一阈值，表示当问题 P 的规模不超过 n_0 时（即 P 问题规模足够小时）已容易直接解出，不必再继续分解。adhoc(P)是该分治法中的基本子算法，用于直接解小规模的问题 P。算法 merge(y_1,y_2,…,y_k)是该分治法中的合并子算法，用于将 P 的子问题 P_1、P_2、…、P_k 的相应解 y_1、y_2、…、y_k 合并为 P 的解。

根据分治法的分解原则，原问题应该分解为多少个子问题才较适合？各个子问题的规模应该怎样才为适当？这些问题很难给予肯定的回答。但人们从大量的实践中发现，在用分治法设计算法时最好使子问题的规模大致相同。换句话说，将一个问题分成大小相等的 k 个子问题的处理方法是行之有效的。当 $k=1$ 时称为**减治法**。许多问题可以取 $k=2$，称为二分法，如图 3.1 所示，这种使子问题规模大致相等的做法出自一种平衡子问题的思想，它几乎总是比子问题规模不等的做法要好。

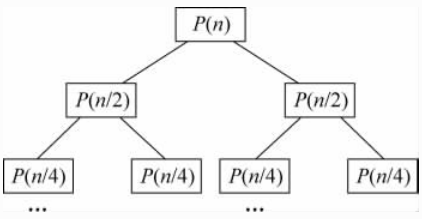

图 3.1　二分法的基本策略

分治法的合并步骤是算法的关键所在。有些问题的合并方法比较明显，有些问题的合并方法比较复杂，或者是有多种合并方案；或者是合并方案不明显。究竟应该怎样合并没有统一的模式，需要具体问题具体分析。

尽管许多分治法算法都是采用递归实现的，但要注意分治法和递归是有区别的，分治法是一种求解问题的策略，而递归是一种实现求解算法的技术。分治法算法也可以采用非递归方法实现。就像二分查找，作为一种典型的分治法算法，既可以采用递归实现，也可以采用非递归实现。

3.2 求解排序问题

对于给定的含有 n 个元素的数组 a，对其按元素值递增排序。快速排序和归并排序是典型的采用分治法进行排序的方法。

3.2.1 快速排序

快速排序的基本思想是在待排序的 n 个元素中任取一个元素(通常取第一个元素)作为基准,把该元素放入最终位置后,整个数据序列被基准分割成两个子序列,所有小于基准的元素放置在前子序列中,所有大于基准的元素放置在后子序列中,并把基准排在这两个子序列的中间,这个过程称为划分,如图 3.2 所示。然后对两个子序列分别重复上述过程,直到每个子序列内只有一个元素或空为止。

图 3.2 快速排序的一趟排序过程

这是一种二分法思想,每次将整个无序序列一分为二,归位一个元素,对两个子序列采用同样的方式进行排序,直到子序列的长度为 1 或 0 为止。

快速排序的分治策略如下。

(1) 分解:将原序列 $a[s..t]$分解成两个子序列 $a[s..i-1]$和 $a[i+1..t]$,其中 i 为划分的基准位置,即将整个问题分解为两个子问题。

(2) 求解子问题:若子序列的长度为 0 或 1,则它是有序的,直接返回;否则递归地求解各个子问题。

(3) 合并:由于整个序列存放在数组 a 中,排序过程是就地进行的,合并步骤不需要执行任何操作。

例如,对于(2,5,1,7,10,6,9,4,3,8)序列,其快速排序过程如图 3.3 所示,图中虚线表示一次划分,虚线旁的数字表示执行次序,圆圈表示归位的基准。

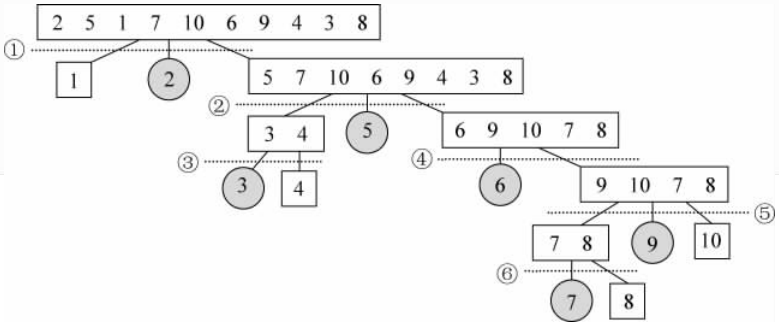

图 3.3 (2,5,1,7,10,6,9,4,3,8)序列的快速排序过程

实现快速排序的完整程序如下：

```
#include <stdio.h>
void disp(int a[],int n)                    //输出 a 中的所有元素
{   int i;
    for (i=0;i<n;i++)
        printf("%d ",a[i]);
    printf("\n");
}
int Partition(int a[],int s,int t)          //划分算法
{   int i=s,j=t;
    int tmp=a[s];                           //用序列的第 1 个记录作为基准
    while (i!=j)                            //从序列两端交替向中间扫描,直到 i=j 为止
    {   while (j>i && a[j]>=tmp)
            j--;                            //从右向左扫描,找第 1 个关键字小于 tmp 的 a[j]
        a[i]=a[j];                          //将 a[j]前移到 a[i]的位置
        while (i<j && a[i]<=tmp)
            i++;                            //从左向右扫描,找第 1 个关键字大于 tmp 的 a[i]
        a[j]=a[i];                          //将 a[i]后移到 a[j]的位置
    }
    a[i]=tmp;
    return i;
}
void QuickSort(int a[],int s,int t)         //对 a[s..t]元素序列进行递增排序
{   if (s<t)                                //序列内至少存在两个元素的情况
    {   int i=Partition(a,s,t);
        QuickSort(a,s,i-1);                 //对左子序列递归排序
        QuickSort(a,i+1,t);                 //对右子序列递归排序
    }
}
void main()
{   int n=10;
    int a[]={2,5,1,7,10,6,9,4,3,8};
    printf("排序前:"); disp(a,n);
    QuickSort(a,0,n-1);
    printf("排序后:"); disp(a,n);
}
```

【算法分析】 快速排序的时间主要耗费在划分操作上，对长度为 n 的区间进行划分，共需 $n-1$ 次关键字的比较，时间复杂度为 $O(n)$。

对 n 个元素进行快速排序的过程构成一棵递归树，在这样的递归树中，每一层最多对 n 个元素进行划分，所花的时间为 $O(n)$。当初始排序数据正序或反序时，递归树高度为 n，快速排序呈现最坏情况，即最坏情况下的时间复杂度为 $O(n^2)$；当初始排序数据随机分布，使每次分成的两个子区间中的元素个数大致相等时，递归树高度为 $\log_2 n$，快速排序呈现最好情况，即最好情况下的时间复杂度为 $O(n\log_2 n)$。快速排序算法的平均时间复杂度也是 $O(n\log_2 n)$。所以快速排序是一种高效的算法，STL 中的 sort()算法就是采用快速排序方法实现的。

3.2.2 归并排序

归并排序的基本思想是首先将 $a[0..n-1]$ 看成 n 个长度为 1 的有序表，将相邻的 $k(k\geqslant 2)$ 个有序子表成对归并，得到 n/k 个长度为 k 的有序子表；然后再将这些有序子表继续归并，得到 n/k^2 个长度为 k^2 的有序子表，如此反复进行下去，最后得到一个长度为 n 的有序表。由于整个排序结果放在一个数组中，所以不需要特别地进行合并操作。

若 $k=2$，即归并是在相邻的两个有序子表中进行的，称为**二路归并排序**。若 $k>2$，即归并操作在相邻的多个有序子表中进行，则叫**多路归并排序**。这里仅讨论二路归并排序算法，二路归并排序算法主要有两种，下面一一讨论。

1. 自底向上的二路归并排序算法

自底向上的二路归并算法采用归并排序的基本原理，第 1 趟归并排序时将待排序的表 $a[0..n-1]$ 看作是 n 个长度为 1 的有序子表，将这些子表两两归并，若 n 为偶数，则得到 $\lceil n/2\rceil$ 个长度为 2 的有序子表；若 n 为奇数，则最后一个子表轮空（不参与归并），故本趟归并完成后，前 $\lceil n/2\rceil-1$ 个有序子表长度为 2，但最后一个子表长度仍为 1；第 2 趟归并则是将第 1 趟归并所得到的 $\lceil n/2\rceil$ 个有序子表两两归并，如此反复，直到最后得到一个长度为 n 的有序表为止。

首先设计算法 Merge()用于将两个有序子表归并为一个有序子表。设两个有序子表存放在同一个表中相邻的位置上，即 $a[low..mid]$（有 mid−low+1 个元素）、$a[mid+1..high]$（有 high−mid 个元素），先将它们合并到一个临时表 tmpa[0..high−low]中，在合并完成后将 tmpa 复制到 a 中。其归并过程是循环从两个子表中顺序取出一个元素进行比较，并将较小者放到 tmpa 中，当一个子表元素取完时将另一个子表中余下的部分直接复制到 tmpa 中。这样 tmpa 是一个有序表，再将其复制到 a 中。

其次，设计算法 MergePass()通过调用 Merge()算法解决一趟归并问题。在某趟归并中，设各子表长度为 length（最后一个子表的长度可能小于 length），则归并前 $a[0..n-1]$ 中共 $\left\lceil \frac{n}{\text{length}} \right\rceil$ 个有序子表，即 $a[0..1\text{ength}-1]$、$a[\text{length}..2\text{length}-1]$、…、$a\left[\left(\left\lceil \frac{n}{\text{length}} \right\rceil\right)\text{length}..n-1\right]$。调用 Merge()一次将相邻的一对子表进行归并，另外需要对表的个数可能是奇数以及最后一个子表的长度小于 length 这两种特殊情况进行处理：若子表的个数为奇数，则最后一个子表无须和其他子表归并（即本趟轮空）；若子表的个数为偶数，则要注意到最后一对子表中后一个子表的区间上界是 $n-1$。

最后，对于含有 n 个元素的序列 a，设计算法 MergeSort()调用 MergePass()算法 $\lceil \log_2 n\rceil$ 次实现二路归并排序。

二路归并排序的分治策略如下：

循环 $\lceil \log_2 n\rceil$ 次，length 依次取 1、2、…、$\log_2 n$，每次执行以下步骤。

（1）分解：将原序列分解成 length 长度的若干个子序列。

（2）求解子问题：对相邻的两个子序列调用 Merge 算法合并成一个有序子序列。

（3）合并：由于整个序列存放在数组 a 中，排序过程是就地进行的，合并步骤不需要执行任何操作。

例如，对于(2,5,1,7,10,6,9,4,3,8)序列，其排序过程如图 3.4 所示，图中方括号内是一个有序子序列。

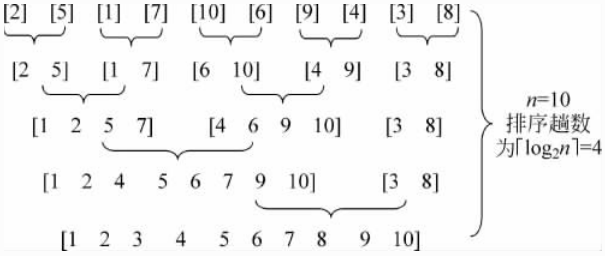

图 3.4　自底向上的二路归并排序过程

实现二路归并排序的完整程序如下：

```
#include <stdio.h>
#include <malloc.h>
void disp(int a[],int n)                    //输出 a 中的所有元素
{   int i;
    for (i=0;i<n;i++)
        printf("%d ",a[i]);
    printf("\n");
}
void Merge(int a[],int low,int mid,int high)
//将 a[low..mid]和 a[mid+1..high]两个相邻的有序子序列归并为一个有序子序列 a[low..high]
{   int *tmpa;
    int i=low,j=mid+1,k=0;                  //k 是 tmpa 的下标,i、j 分别为两个子表的下标
    tmpa=(int *)malloc((high-low+1)*sizeof(int));
    while (i<=mid && j<=high)               //在第 1 个子表和第 2 个子表均未扫描完时循环
        if (a[i]<=a[j])                     //将第 1 个子表中的元素放入 tmpa 中
        {   tmpa[k]=a[i];
            i++;k++;
        }
        else                                //将第 2 个子表中的元素放入 tmpa 中
        {   tmpa[k]=a[j];
            j++;k++;
        }
    while (i<=mid)                          //将第 1 个子表余下的部分复制到 tmpa
    {   tmpa[k]=a[i];
        i++;k++;
    }
    while (j<=high)                         //将第 2 个子表余下的部分复制到 tmpa
    {   tmpa[k]=a[j];
        j++;k++;
    }
    for (k=0,i=low;i<=high;k++,i++)         //将 tmpa 复制回 a 中
        a[i]=tmpa[k];
    free(tmpa);                             //释放临时空间
}
```

```
void MergePass(int a[],int length,int n)                    //一趟二路归并排序
{   int i;
    for (i=0;i+2*length-1<n;i=i+2*length)                   //归并 length 长的两个相邻子表
        Merge(a,i,i+length-1,i+2*length-1);
    if (i+length-1<n)                                       //余下两个子表,后者的长度小于 length
        Merge(a,i,i+length-1,n-1);                          //归并这两个子表
}
void MergeSort(int a[],int n)                               //二路归并算法
{   int length;
    for (length=1;length<n;length=2*length)
        MergePass(a,length,n);
}
void main()
{   int n=10;
    int a[]={2,5,1,7,10,6,9,4,3,8};
    printf("排序前:"); disp(a,n);
    MergeSort(a,n);
    printf("排序后:"); disp(a,n);
}
```

【算法分析】 对于上述二路归并排序算法,当有 n 个元素时需要 $\lceil \log_2 n \rceil$ 趟归并,每一趟归并,其元素比较次数不超过 $n-1$,元素移动次数都是 n,因此二路归并排序的时间复杂度为 $O(n\log_2 n)$。

2. 自顶向下的二路归并排序算法

上述自底向上的二路归并算法虽然效率较高,但可读性较差。另一种是采用自顶向下的方法设计,算法更为简洁,属典型的二分法算法。

设归并排序的当前区间是 a[low..high],则递归归并的步骤如下。

(1) 分解:将当前序列 a[low..high]一分为二,即求 mid=(low+high)/2,分解为两个子序列 a[low..mid]和 a[mid+1..high]。

(2) 子问题求解:递归地对两个子序列 a[low..mid]和 a[mid+1..high]二路归并排序。其终结条件是子序列的长度为 1 或者 0(因为一个元素的子表或者空表可以看成有序表)。

(3) 合并:与分解过程相反,将已排序的两个子序列 a[low..mid]和 a[mid+1..high]归并为一个有序序列 a[low..high]。

对应的二路归并排序算法如下:

```
void MergeSort(int a[],int low,int high)                    //二路归并算法
{   int mid;
    if (low<high)                                           //子序列有两个或两个以上元素
    {   mid=(low+high)/2;                                   //取中间位置
        MergeSort(a,low,mid);                               //对 a[low..mid]子序列排序
        MergeSort(a,mid+1,high);                            //对 a[mid+1..high]子序列排序
        Merge(a,low,mid,high);                              //将两个子序列合并,见前面的算法
    }
}
```

例如，对于(2,5,1,7,10,6,9,4,3,8)序列，其排序过程如图 3.5 所示，图中圆括号内的数字指出操作顺序。

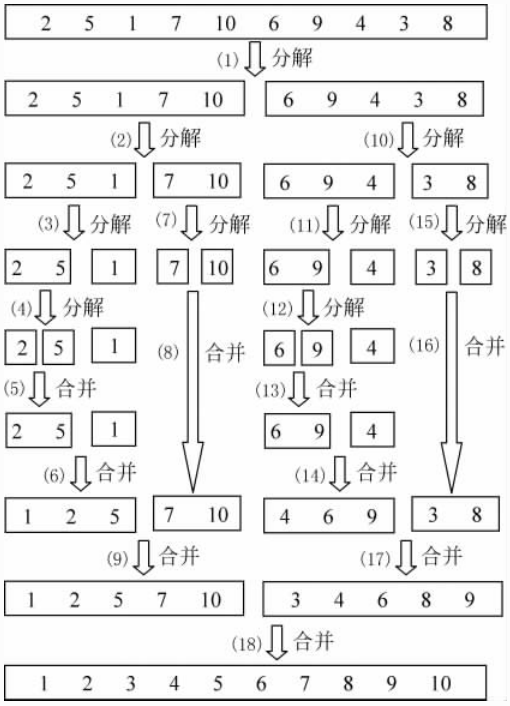

图 3.5　自顶向下的二路归并排序过程

【算法分析】 设 MergeSort(a,0,$n-1$)算法的执行时间为 $T(n)$，显然 Merge(a,0,$n/2$,$n-1$)合并操作的执行时间为 $O(n)$，所以得到以下递推式：

$$T(n)=1 \quad \text{当 } n=1 \text{ 时}$$
$$T(n)=2T(n/2)+O(n) \quad \text{当 } n>1 \text{ 时}$$

容易推出 $T(n)=O(n\log_2 n)$。

3.3 求解查找问题

3.3.1 查找最大和次大元素

【问题描述】 对于给定的含有 n 个元素的无序序列，求这个序列中最大和次大的两个不同元素。

【问题求解】 对于无序序列 a[low..high]，采用分治法求最大元素 max1 和次大元素 max2 的过程如下：

(1) 若 a[low..high]中只有一个元素，则 max1 = a[low]，max2 =

扫一扫

视频讲解

−INF(−∞)。

(2) 若 a[low..high]中只有两个元素，则 max1＝max{a[low],a[high]}，max2＝min{a[low],a[high]}。

(3) 若 a[low..high]中有两个以上元素，按中间位置 mid＝(low＋high)/2 划分为 a[low..mid]和 a[mid＋1..high]两个区间(注意左区间包含 a[mid]元素)。求出左区间的最大元素 lmax1 和次大元素 lmax2，求出右区间的最大元素 rmax1 和次大元素 rmax2。

若 lmax1＞rmax1，则 max1＝lmax1，max2＝max{lmax2，rmax1}；否则 max1＝rmax1，max2＝max{lmax1，rmax2}。

例如，对于 a[0..4]＝{5,2,1,4,3}，mid＝(0＋4)/2＝2，划分为左区间 a[0..2]＝{5,2,1}，右区间 a[3..4]＝{4,3}。在左区间中求出 lmax1＝5，lmax2＝2，在右区间中求出 rmax1＝4，rmax2＝3。所以 max1＝max{lmax1，rmax1}＝5，max2＝max{lmax2，rmax1}＝4。

对应的算法如下：

```
void solve(int a[],int low,int high,int &max1,int &max2)
{   if (low==high)                              //区间中只有一个元素
    {   max1=a[low];
        max2=-INF;
    }
    else if (low==high-1)                       //区间中只有两个元素
    {   max1=max(a[low],a[high]);
        max2=min(a[low],a[high]);
    }
    else                                        //区间中有两个以上元素
    {   int mid=(low+high)/2;
        int lmax1,lmax2;
        solve(a,low,mid,lmax1,lmax2);           //左区间求 lmax1 和 lmax2
        int rmax1,rmax2;
        solve(a,mid+1,high,rmax1,rmax2);        //右区间求 rmax1 和 rmax2
        if (lmax1>rmax1)
        {   max1=lmax1;
            max2=max(lmax2,rmax1);              //lmax2、rmax1 中求次大元素
        }
        else
        {   max1=rmax1;
            max2=max(lmax1,rmax2);              //lmax1、rmax2 中求次大元素
        }
    }
}
```

【算法分析】 对于 solve(a,0,n−1,max1,max2)调用，其比较次数的递推式如下：

$$T(1)=T(2)=1$$
$$T(n)=2T(n/2)+1 \qquad \text{//合并的时间为 } O(1)$$

可以推导出 $T(n)=O(n)$。

3.3.2 折半查找

折半查找又称二分查找，它是一种效率较高的查找方法。但是折半查找要求查找序列中的元素是有序的，为了简单，假设是递增有序的。

折半查找的基本思路：设 $a[low..high]$是当前的查找区间，首先确定该区间的中点位置 $mid=\lfloor(low+high)/2\rfloor$；然后将待查的 k 值与 $a[mid].key$ 比较。

(1) 若 $k==a[mid].key$，则查找成功并返回该元素的物理下标。

(2) 若 $k<a[mid]$，则由表的有序性可知 $a[mid..high]$均大于 k，因此若表中存在关键字等于 k 的元素，则该元素必定位于左子表 $a[low..mid-1]$中，故新的查找区间是左子表 $a[low..mid-1]$。

(3) 若 $k>a[mid]$，则要查找的 k 必定位于右子表 $a[mid+1..high]$中，即新的查找区间是右子表 $a[mid+1..high]$。

下一次查找是针对新的查找区间进行的。

因此可以从初始的查找区间 $a[0..n-1]$开始，每经过一次与当前查找区间的中点位置上的关键字比较就可确定查找是否成功，不成功则当前的查找区间缩小一半。重复这一过程，直到找到关键字为 k 的元素，或者直到当前的查找区间为空(即查找失败)时为止。

折半查找对应的完整程序如下：

```
#include <stdio.h>
int BinSearch(int a[],int low,int high,int k)          //折半查找算法
{   int mid;
    if (low<=high)                                     //当前区间存在元素时
    {   mid=(low+high)/2;                              //求查找区间的中间位置
        if (a[mid]==k)                                 //找到后返回其物理下标 mid
            return mid;
        if (a[mid]>k)                                  //当 a[mid]>k 时在 a[low..mid-1]中递归查找
            return BinSearch(a,low,mid-1,k);
        else                                           //当 a[mid]<k 时在 a[mid+1..high]中递归查找
            return BinSearch(a,mid+1,high,k);
    }
    else return -1;                                    //当前查找区间没有元素时返回-1
}
void main()
{   int n=10,i;
    int k=6;
    int a[]={1,2,3,4,5,6,7,8,9,10};
    i=BinSearch(a,0,n-1,k);
    if (i>=0)printf("a[%d]=%d\n",i,k);
    else printf("未找到%d元素\n",k);
}
```

可以将折半查找递归算法等价地转换成以下非递归算法：

```
int BinSearch1(int a[],int n,int k)                    //非递归折半查找算法
{   int low=0,high=n-1,mid;
```

```
    while (low<=high)                    //当前区间存在元素时循环
    {   mid=(low+high)/2;                //求查找区间的中间位置
        if (a[mid]==k)                   //找到后返回其物理下标 mid
            return mid;
        if (a[mid]>k)                    //继续在 a[low..mid-1]中查找
            high=mid-1;
        else                             //a[mid]<k
            low=mid+1;                   //继续在 a[mid+1..high]中查找
    }
    return -1;                           //当前查找区间没有元素时返回-1
}
```

【算法分析】 折半查找算法的主要时间花费在元素的比较上，对于含有 n 个元素的有序表，采用折半查找时最坏情况下的元素比较次数为 $C(n)$，则有：

$$\begin{aligned} &C(n)=1 && \text{当 } n=1 \text{ 时}\\ &C(n)\leqslant 1+C(\lfloor n/2\rfloor) && \text{当 } n\geqslant 2 \text{ 时}\end{aligned}$$

设对某个整数 $k\geqslant 2$，满足 $2^{k-1}\leqslant n<2^k$。展开上述递推式，可得到：

$$\begin{aligned} C(n) &\leqslant 1+C(\lfloor n/2\rfloor)\\ &\leqslant 2+C\lfloor n/4\rfloor)\\ &\cdots\\ &\leqslant (k-1)+C\lfloor n/2^{k-1}\rfloor)\\ &=(k-1)+1\\ &=k\end{aligned}$$

而 $2^{k-1}\leqslant n<2^k$，即 $k\leqslant \log_2 n+1<k+1$，$k=\lfloor \log_2 n\rfloor+1$。

由此得到 $C(n)\leqslant \lfloor \log_2 n\rfloor+1$。

也就是说，在含有 n 个元素的有序序列中采用折半查找算法查找指定的元素所需的元素比较次数不超过 $\lfloor \log_2 n\rfloor+1$（或者 $\lceil \log_2(n+1)\rceil$）。实际上，n 个元素的折半查找对应判定树的高度恰好是 $\lfloor \log_2 n\rfloor+1$。折半查找的主要时间花在元素的比较上，所以算法的时间复杂度为 $O(\log_2 n)$。

折半查找的思路很容易推广到三分查找，显然三分查找对应判断树的高度恰好是 $\lfloor \log_3 n\rfloor+1$，推出查找时间复杂度为 $O(\log_3 n)$，由于 $\log_3 n=\log_2 n/\log_2 3$，所以三分查找和二分查找的时间是同一个数量级的。

【例 3.1】 求解假币问题。有 100 个硬币，其中有一个假币（与真币一模一样，只是比真币的重量轻），采用天平称重方法找出这个假币，最少用天平称重多少次保证找出假币。

解 已知假币比真币的重量轻，可以将 100 个硬币分为两组，每组 50 个硬币，称重一次可以确定假币所在的组，即二分法。更好的方法是采用三分法，将 100 个硬币分为 33、33、34 三组，用天平一次称重可以找出假币所在的组，依次进行，对应一棵三分判定树，树高度恰好是称重次数，结果为 $\lceil \log_3(100+1)\rceil=5$。

3.3.3 寻找一个序列中第 k 小的元素

【问题描述】 对于给定的含有 n 个元素的无序序列，求这个序列中第 $k(1\leqslant k\leqslant n)$ 小的元素。

【问题求解】 假设无序序列存放在 $a[0..n-1]$ 中，若将 a 递增排序，则第 k 小的元素为 $a[k-1]$。采用类似快速排序的思想。

对于无序序列 $a[s..t]$，在其中查找第 k 小的元素的过程如下：

(1) 若 $s \geqslant t$，即其中只有一个元素或没有任何元素，如果 $s=t$ 且 $s=k-1$，表示只有一个元素且 $a[k-1]$ 就是要求的结果，返回 $a[k-1]$。

(2) 若 $s<t$，表示该序列中有两个或两个以上的元素，以基准为中心将其划分为 $a[s..i-1]$ 和 $a[i+1..t]$ 两个子序列，基准 $a[i]$ 已归位，$a[s..i-1]$ 中的所有元素均小于 $a[i]$，$a[i+1..t]$ 中的所有元素均大于 $a[i]$，也就是说 $a[i]$ 是第 $i+1$ 小的元素，有 3 种情况。

- 若 $k-1=i$，$a[i]$ 即为所求，返回 $a[i]$。
- 若 $k-1<i$，第 k 小的元素应在 $a[s..i-1]$ 子序列中，递归在该子序列中求解并返回其结果。
- 若 $k-1>i$，第 k 小的元素应在 $a[i+1..t]$ 子序列中，递归在该子序列中求解并返回其结果。

对应的完整程序如下：

```
#include <stdio.h>
int QuickSelect(int a[],int s,int t,int k)          //在 a[s..t]序列中找第 k 小的元素
{   int i=s,j=t;
    int tmp;
    if (s<t)                                        //区间内至少存在两个元素的情况
    {   tmp=a[s];                                   //用区间的第 1 个记录作为基准
        while (i!=j)                                //从区间两端交替向中间扫描,直到 i=j 为止
        {   while (j>i && a[j]>=tmp)
                j--;                                //从右向左扫描,找第 1 个关键字小于 tmp 的 a[j]
            a[i]=a[j];                              //将 a[j]前移到 a[i]的位置
            while (i<j && a[i]<=tmp)
                i++;                                //从左向右扫描,找第 1 个关键字大于 tmp 的 a[i]
            a[j]=a[i];                              //将 a[i]后移到 a[j]的位置
        }
        a[i]=tmp;
        if (k-1==i) return a[i];
        else if (k-1<i) return QuickSelect(a,s,i-1,k);      //在左区间中递归查找
        else return QuickSelect(a,i+1,t,k);         //在右区间中递归查找
    }
    else if (s==t && s==k-1)                        //区间内只有一个元素且为 a[k-1]
        return a[k-1];
}
void main()
{   int n=10,k;
    int e;
    int a[]={2,5,1,7,10,6,9,4,3,8};
    for (k=1;k<=n;k++)
    {   e=QuickSelect(a,0,n-1,k);
        printf("第%d 小的元素:%d\n",k,e);
    }
}
```

本程序的执行结果如下：

```
第 1 小的元素:1
第 2 小的元素:2
第 3 小的元素:3
第 4 小的元素:4
第 5 小的元素:5
第 6 小的元素:6
第 7 小的元素:7
第 8 小的元素:8
第 9 小的元素:9
第 10 小的元素:10
```

【算法分析】 对于 QuickSelect(a,s,t,k)算法，设序列 a 中含有 n 个元素，其比较次数的递推式为：

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

可以推导出 $T(n)=O(n)$，这是最好的情况，即每次划分的基准恰好是中位数，将一个序列划分为长度大致相等的两个子序列。在最坏情况下，每次划分的基准恰好是序列中的最大值或最小值，则处理区间只比上一次减少 1 个元素，此时比较次数为 $O(n^2)$。在平均情况下该算法的时间复杂度为 $O(n)$。

3.3.4 寻找两个等长有序序列的中位数

【问题描述】 对于一个长度为 n 的有序序列(假设均为升序序列) $a[0..n-1]$，处于中间位置的元素称为 a 的中位数。例如，若序列 $a=(11,13,15,17,19)$，其中位数是 15，若 $b=(2,4,6,8,20)$，其中位数为 6。两个等长有序序列的中位数是含它们所有元素的有序序列的中位数，例如 a、b 两个有序序列的中位数为 11。设计一个算法求给定的两个有序序列的中位数。

【问题求解】 对于含有 n 个元素的有序序列 $a[s..t]$，当 n 为奇数时，中位数出现在 $m=\lfloor(s+t)/2\rfloor$ 处；当 n 为偶数时，中位数下标有 $m=\lfloor(s+t)/2\rfloor$(下中位)和 $m=\lfloor(s+t)/2\rfloor+1$(上中位)两个。为了简单，这里仅考虑中位数下标为 $m=\lfloor(s+t)/2\rfloor$。

采用二分法求含有 n 个有序元素的序列 a、b 的中位数的过程如下：

(1) 分别求出 a、b 的中位数 $a[m_1]$和 $b[m_2]$。

(2) 若 $a[m_1]=b[m_2]$，则 $a[m_1]$或 $b[m_2]$即为所求中位数，如图 3.6(a)所示，算法结束。

(3) 若 $a[m_1]<b[m_2]$，则舍弃序列 a 中的前半部分(较小的一半)，同时舍弃序列 b 中的后半部分(较大的一半)，要求舍弃的长度相等，如图 3.6(b)所示。

(4) 若 $a[m_1]>b[m_2]$，则舍弃序列 a 中的后半部分(较大的一半)，同时舍弃序列 b 中的前半部分(较小的一半)，要求舍弃的长度相等，如图 3.6(c)所示。

在保留的两个升序序列中重复上述过程直到两个序列中只含有一个元素时为止，较小者即为所求的中位数。

为了保证每次取的两个子有序序列等长，对于 $a[s..t]$，$m=(s+t)/2$，若取前半部分，则

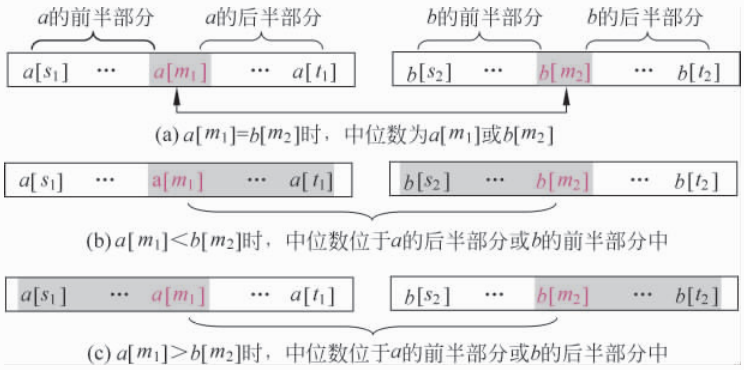

图 3.6　求两个等长有序序列中位数的过程

为 $a[s..m]$。在取后半部分时要区分 a 中的元素个数为奇数还是偶数，若为奇数（满足 $(s+t)\%2==0$ 的条件），则后半部分为 $a[m..t]$，若为偶数（满足 $(s+t)\%2==1$ 的条件），则后半部分为 $a[m+1..t]$。

例如，求 $a=(11,13,15,17,19)$、$b=(2,4,6,8,20)$ 两个有序序列的中位数的过程如图 3.7 所示。

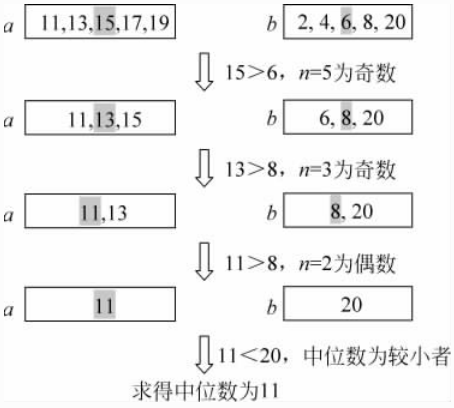

图 3.7　求 a、b 两个有序序列的中位数

对应的完整程序如下：

```
#include <stdio.h>
void prepart(int &s,int &t)                    //求 a[s..t]序列的前半子序列
{   int m=(s+t)/2;
    t=m;
}
void postpart(int &s,int &t)                   //求 a[s..t]序列的后半子序列
{   int m=(s+t)/2;
    if ((s+t)%2==0)                            //序列中有奇数个元素
        s=m;
    else                                       //序列中有偶数个元素
        s=m+1;
}
```

```
int midnum(int a[],int s1,int t1,int b[],int s2,int t2)
{                                          //求两个有序序列 a[s1..t1]和 b[s2..t2]的中位数
    int m1,m2;
    if (s1==t1 && s2==t2)                  //两个序列只有一个元素时返回较小者
        return a[s1]<b[s2]?a[s1]:b[s2];
    else
    {   m1=(s1+t1)/2;                      //求 a 的中位数
        m2=(s2+t2)/2;                      //求 b 的中位数
        if (a[m1]==b[m2])                  //两中位数相等时返回该中位数
            return a[m1];
        if (a[m1]<b[m2])                   //当 a[m1]<b[m2]时
        {   postpart(s1,t1);               //a 取后半部分
            prepart(s2,t2);                //b 取前半部分
            return midnum(a,s1,t1,b,s2,t2);
        }
        else                               //当 a[m1]>b[m2]时
        {   prepart(s1,t1);                //a 取前半部分
            postpart(s2,t2);               //b 取后半部分
            return midnum(a,s1,t1,b,s2,t2);
        }
    }
}
void main()
{   int a[]={11,13,15,17,19};
    int b[]={2,4,6,8,20};
    printf("中位数:%d\n",midnum(a,0,4,b,0,4));
}
```

其中求 a、b 两个有序序列的中位数的算法也可以用循环语句来替换，等价的非递归算法如下：

```
int midnum1(int a[],int b[],int n)
{   int s1,t1,m1,s2,t2,m2;
    s1=0; t1=n-1;
    s2=0; t2=n-1;
    while (s1!=t1 || s2!=t2)
    {   m1=(s1+t1)/2;
        m2=(s2+t2)/2;
        if (a[m1]==b[m2])
            return a[m1];
        if (a[m1]<b[m2])
        {   postpart(s1,t1);
            prepart(s2,t2);
        }
        else
        {   prepart(s1,t1);
            postpart(s2,t2);
        }
    }
```

```
    return a[s1]<b[s2]?a[s1]:b[s2];
}
```

【算法分析】 对于含有 n 个元素的有序序列 a 和 b，设调用 midnum(a,0,$n-1$,b,0,$n-1$)求中位数的执行时间为 $T(n)$，显然有以下递推式：

$$T(n)=1 \quad \text{当 } n=1 \text{ 时}$$
$$T(n)=2T(n/2)+1 \quad \text{当 } n>1 \text{ 时}$$

容易推出 $T(n)=O(\log_2 n)$。

【例 3.2】 给出已排序数组 a、b，长度分别为 n、m(n 与 m 不必相等)，请找出一个时间复杂度为 $O(\log_2(n+m))$的算法，找到排在第 k($1\leqslant k\leqslant n+m$)位置的元素。

解 假设是递增排序，先考虑 a 和 b 的元素个数都大于 $k/2$ 的情况。将 a 的第 $k/2$ 个元素(即 $a[k/2-1]$)和 b 的第 $k/2$ 个元素(即 $b[k/2-1]$)进行比较，有以下 3 种情况(为了简化，这里先假设 k 为偶数，所得到的结论对于 k 是奇数也是成立的，合并后第 k 小的元素用 top$k$ 表示)。

- $a[k/2-1]=b[k/2-1]$：则 $a[0..k/2-2]$($k/2-1$ 个元素)和 $b[0..k/2-2]$($k/2-1$ 个元素)共 $k-2$ 个元素均小于等于 top$k$，再加上 $a[k/2-1]$、$b[k/2-1]$两个元素，说明找到了 top$k$，即 top$k$ 等于 $a[k/2-1]$或 $b[k/2-1]$，直接返回 $a[k/2-1]$或 $b[k/2-1]$即可。
- $a[k/2-1]<b[k/2-1]$：如果 $a[k/2-1]<b[k/2-1]$，意味着 $a[0]\sim a[k/2-1]$(共 $k/2$ 个元素)肯定均小于等于 top$k$，换句话说，$a[k/2-1]$一定小于等于 top$k$(可以用反证法证明，假设 $a[k/2-1]>$top$k$，那么 $a[k/2-1]$后面的元素均大于 top$k$，因此 $b[k/2-1]$及后面一定有一个元素为 top$k$，也就是说 $b[k/2-1]\leqslant$top$k$，与 $a[k/2-1]<b[k/2-1]$矛盾，即证)。这样 $a[0]\sim a[k/2-1]$均小于等于 top$k$ 并且尚未找到第 k 个元素，因此可以删除 a 数组的这 $k/2$ 个元素。
- $a[k/2-1]>b[k/2-1]$：同上，可以删除 b 数组的 $b[0..k/2-1]$共 $k/2$ 个元素。

因此可以设计一个递归函数求解，其递归出口如下：

- 当 a 或 b 为空时直接返回 $b[k-1]$或 $a[k-1]$ 。
- 当 $k=1$ 时返回 min($a[0]$,$b[0]$)。
- 当 $a[k/2-1]$ == $b[k/2-1]$时返回 $a[k/2-1]$或 $b[k/2-1]$。

考虑算法的通用性，当 k 是奇数，或者 a 或 b 的元素个数小于 $k/2$ 时，采用的方法如下：

(1) 总是让 a 中的元素个数最少，当 b 中的元素个数较少时交换参数 a、b 的位置即可。

(2) 将前面 $a[k/2-1]$和 $b[k/2-1]$的比较改为 $a[$numa$-1]$和 $b[$numb$-1]$的比较，保证这两个元素前面的元素个数恰好为 $k-2$，即 top$k$ 来自 $a[$numa$-1]$或者 $b[$numb$-1]$。所以当 a 中的元素个数少于 $k/2$ 时取 numa$=n$，否则取 numa$=k/2$，而 numb$=k-$numa。

用 Findk(a,n,b,m,k)算法求有序序列 a 和 b 的 top$k$。例如，$a=(1,5,8)$，$n=3$，$b=(2,3,4,6,7)$，$m=5$ 时，求 $k=3$ 的 top$k$ 的过程如图 3.8 所示。每次递归调用，k 递减 numa 或者 numb，而 numa 或者 numb 近似于 $k/2$，相当于 k 减半，当 $k=1$ 时为递归出口，从而得到最终解。

上述过程每次递归调用时 k 减半，所以执行时间为 $\log_2 k$，最多 $k=n+m$，所以时间复杂度为 $O(\log_2(n+m))$。

图 3.8　求解 $k=3$ 的 topk 的过程

对应的完整程序如下：

```
#include <stdio.h>
int Findk(int a[],int n,int b[],int m, int k)   //在两个升序排列的数组中找到第 k 大的元素
{   if (k<0) return -1;
    if (n>m)                                    //用于保证 n≤m,即保证前一个数组的元素较少
        return Findk(b,m,a,n,k);
    if (n==0)
        return b[k-1];
    if (k==1)
        return ((a[0]>=b[0]) ? b[0]:a[0]);
    int numa=(n>=k/2)?k/2:n;                    //当数组中没有 k/2 个元素时取 n
    int numb=k-numa;
    if(a[numa-1]==b[numb-1])
        return a[numa-1];
    else if(a[numa-1]>b[numb-1])
        return Findk(a,n,&b[numb],m-numb,k-numb);
    else if(a[numa-1]<b[numb-1])
        return Findk(&a[numa],n-numa,b,m,k-numa);
}
void main()
{   int i,result;
    int a[]={1,5,8};
    int b[]={2,3,4,6,7};
    int n=sizeof(a)/sizeof(a[0]);
    int m=sizeof(b)/sizeof(b[0]);
    printf("求解结果:\n");
    for(i=1;i<=n+m;i++)
    {   result=Findk(a,n,b,m,i);
        printf(" 第%d 小的元素是:%d\n",i,result);
    }
}
```

上述程序的执行过程如下：

```
求解结果：
    第 1 小的元素是:1
    第 2 小的元素是:2
    第 3 小的元素是:3
    第 4 小的元素是:4
    第 5 小的元素是:5
    第 6 小的元素是:6
    第 7 小的元素是:7
    第 8 小的元素是:8
```

说明：对于含 n 个元素的数组 $a[0..n-1]$&$a[numa]$表示取 $a[numa..n-1]$部分的元素。

3.4 求解组合问题

3.4.1 求解最大连续子序列和问题

扫一扫

视频讲解

【问题描述】 给定一个有 $n(n\geqslant 1)$个整数的序列，求出其中最大连续子序列的和。例如序列$(-2,11,-4,13,-5,-2)$的最大子序列和为 20，序列$(-6,2,4,-7,5,3,2,-1,6,-9,10,-2)$的最大子序列和为 16。规定一个序列的最大连续子序列和至少是 0，如果小于 0，其结果为 0。

【问题求解】 对于含有 n 个整数的序列 $a[0..n-1]$，若 $n=1$，表示该序列仅含一个元素，如果该元素大于 0，则返回该元素，否则返回 0。

若 $n>1$，采用分治法求解最大连续子序列时取其中间位置 mid$=\lfloor(n-1)/2\rfloor$，该子序列只可能出现在 3 个地方，各种情况及求解方法如图 3.9 所示。

(1) 该子序列完全落在左半部，即 $a[0..\text{mid}]$中，采用递归求出其最大连续子序列和 maxLeftSum，如图 3.9(a)所示。

图 3.9 求解最大连续子序列和的过程

(2) 该子序列完全落在右半部,即 $a[\text{mid}+1..n-1]$中,采用递归求出其最大连续子序列和 maxRightSum,如图 3.9(a)所示。

(3) 该子序列跨越序列 a 的中部而占据左、右两部分。也就是说,这种情况下最大和的连续子序列含有 a_{mid},则从左半部(含 a_{mid} 元素)求出 $\text{maxLeftBorderSum}=\max\sum_{k=i}^{\text{mid}}a_k\{0\leqslant i\leqslant \text{mid}\}$,从右半部(不含 a_{mid} 元素)求出 $\text{maxRightBorderSum}=\max\sum_{k=\text{mid}+1}^{j}a_k\{\text{mid}+1\leqslant j\leqslant n-1\}$,这种情况下的最大连续子序列和 maxMidSum=maxLeftBorderSum+maxRightBorderSum,如图 3.9(b)所示。

最后整个序列 a 的最大连续子序列和为 maxLeftSum、maxRightSum 和 maxMidSum 中的最大值,如图 3.9(c)所示。

例如,$a[0..5]=\{-2,11,-4,13,-5,-2\}$,$n=6$,mid=(0+5)/2=2,划分为 $a[0..2]$和 $a[3..5]$左、右两个部分。递归求出左部分(-2,11,-4)的最大连续子序列和为 11,递归求出右部分(13,-5,-2)的最大连续子序列和为 13,再求出以 $a[\text{mid}]=-4$ 为中心的最大连续子序列和为 20(对应序列为 11,-4,13),最终结果为 max{11,13,20}=20。

求最大连续子序列和的完整程序如下:

```
#include <stdio.h>
long max3(long a,long b,long c)                    //求出 3 个 long 中的最大值
{   if (a<b) a=b;                                  //用 a 保存 a、b 中的最大值
    if (a>c) return a;                             //比较返回 a、c 中的最大值
    else return c;
}
long maxSubSum(int a[],int left,int right)         //求 a[left..high]序列中的最大连续子序列和
{   int i,j;
    long maxLeftSum,maxRightSum;
    long maxLeftBorderSum,leftBorderSum;
    long maxRightBorderSum,rightBorderSum;
    if (left==right)                               //当子序列只有一个元素时
    {   if (a[left]>0)                             //该元素大于 0 时返回它
            return a[left];
        else                                       //该元素小于或等于 0 时返回 0
            return 0;
    }
    int mid=(left+right)/2;                        //求中间位置
    maxLeftSum=maxSubSum(a,left,mid);              //求左边的最大连续子序列之和
    maxRightSum=maxSubSum(a,mid+1,right);          //求右边的最大连续子序列之和
    maxLeftBorderSum=0,leftBorderSum=0;
    for (i=mid;i>=left;i--)                        //求出以左边加上 a[mid]元素构成的序列的最大和
    {   leftBorderSum+=a[i];
        if (leftBorderSum>maxLeftBorderSum)
            maxLeftBorderSum=leftBorderSum;
    }
    maxRightBorderSum=0,rightBorderSum=0;
    for (j=mid+1;j<=right;j++)                     //求出 a[mid]右边元素构成的序列的最大和
    {   rightBorderSum+=a[j];
```

```
        if (rightBorderSum > maxRightBorderSum)
            maxRightBorderSum = rightBorderSum;
    }
    return max3(maxLeftSum, maxRightSum, maxLeftBorderSum + maxRightBorderSum);
}
void main()
{   int a[] = {-2,11,-4,13,-5,-2}, n=6;
    int b[] = {-6,2,4,-7,5,3,2,-1,6,-9,10,-2}, m=12;
    printf("a 序列的最大连续子序列的和:%ld\n", maxSubSum4(a,0,n-1));
    printf("b 序列的最大连续子序列的和:%ld\n", maxSubSum4(b,0,m-1));
}
```

【算法分析】 设求解序列 $a[0..n-1]$最大连续子序列和的执行时间为 $T(n)$，第(1)、(2)两种情况的执行时间为 $T(n/2)$，第(3)种情况的执行时间为 $O(n)$，所以得到以下递推式：

$$T(n)=1 \qquad \text{当 } n=1 \text{ 时}$$
$$T(n)=2T(n/2)+n \qquad \text{当 } n>1 \text{ 时}$$

容易推出 $T(n)=O(n\log_2 n)$。

思考题：给定一个有 $n(n\geqslant 1)$个整数的序列，可能含有负整数，求出其中最大连续子序列的积，是否采用上述求最大连续子序列和的方法？

思考题解析：结论是不可以！例如，$a[0..5]=\{-2,3,2,4,1,-5\}$，显然最大连续子序列的积$=(-2)\times 3\times 2\times 4\times(-5)=240$。如果采用上述分治法，mid$=(0+5)/2=2$，划分为 $a[0..2]$和 $a[3..5]$左、右两个部分。递归求出左部分$(-2,3,2)$的最大连续子序列积为 $3\times 2=6$，递归求出右部分$(4,1,-5)$的最大连续子序列积为 $4\times 1=4$，再求出以 $a[\text{mid}]=2$ 为中心的最大连续子序列积为 $3\times 2\times 4\times 1=24$，最终结果为 $\max\{6,4,24\}=24$。

为什么求最大连续子序列积不能采用上述分治法求解，而求最大连续子序列和可以呢？这是因为这两个问题都是求最优解，采用分治法求最优解需要满足最优性原理，即整个问题的最优解由各个子问题的最优解构成，显然求最大连续子序列和问题满足最优性原理，而求最大连续子序列积并不满足最优性原理。例如，当 $x>0$、$y<0$ 时有 $x+y\leqslant x$、$x\times y\leqslant x$；当 $x<0$、$y<0$ 时有 $x+y\leqslant x$，而 $x\times y\geqslant x$。

扫一扫

视频讲解

3.4.2 求解棋盘覆盖问题

【问题描述】 有一个 $2^k\times 2^k(k>0)$的棋盘，恰好有一个方格与其他方格不同，称之为特殊方格。现在要用如图 3.10 所示的 L 形骨牌覆盖除了特殊方格以外的其他全部方格，骨牌可以任意旋转，并且任何两个骨牌不能重叠。请给出一种覆盖方法。

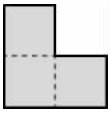

图 3.10 L 形的骨牌

【问题求解】 棋盘中的方格数$=2^k\times 2^k=4^k$，覆盖使用的 L 形骨牌个数$=(4^k-1)/3$。采用的方法是将棋盘划分为大小相同的 4 个象限，根据特殊方格的位置(dr,dc)，在中间位置放置一个合适的 L 形骨牌。例如，如图 3.11(a)所示，特殊方格在左上角象限中，在中间放置一个覆

盖其他 3 个象限中各一个方格的 L 形骨牌。图 3.11(b)～图 3.11(d)是特殊方格在其他象限中放置 L 形骨牌的情况。

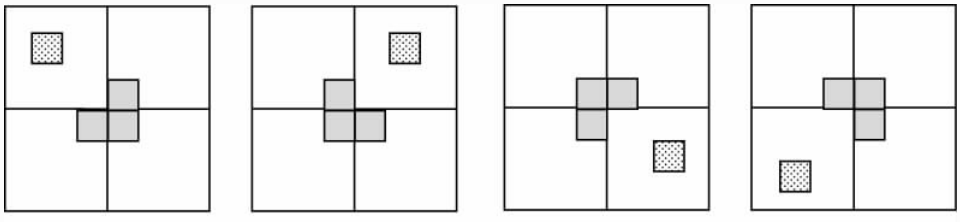

图 3.11　放置一个 L 形骨牌

这样每个象限和包含特殊方格的象限类似，都需要少覆盖一个方格，也与整个问题类似，所以采用分治法求解，将原问题分解为 4 个子问题。

用(tr,tc)表示一个象限左上角方格的坐标，(dr,dc)是特殊方格所在的坐标，size 是棋盘的行数和列数。用二维数组 board 存放覆盖方案，用全局变量 tile 表示 L 形骨牌的编号(从整数 1 开始)，board 中 3 个相同的整数表示一个 L 形骨牌。

对应的完整程序如下：

```
#include<stdio.h>
#define MAX 1025
//问题表示
int k;                                          //棋盘大小
int x,y;                                        //特殊方格的位置
//求解问题表示
int board[MAX][MAX];
int tile=1;                                     //L 形骨牌的编号,从 1 开始
void ChessBoard(int tr,int tc,int dr,int dc,int size)
{   if(size==1) return;                         //递归出口
    int t=tile++;                               //取一个 L 形骨牌,其牌号为 tile
    int s=size/2;                               //分割棋盘
    //考虑左上角象限
    if(dr<tr+s && dc<tc+s)                      //特殊方格在此象限中
        ChessBoard(tr,tc,dr,dc,s);
    else                                        //此象限中无特殊方格
    {   board[tr+s-1][tc+s-1]=t;                //用 t 号 L 形骨牌覆盖右下角
        ChessBoard(tr,tc,tr+s-1,tc+s-1,s);      //将右下角作为特殊方格继续处理该象限
    }
    //考虑右上角象限
    if(dr<tr+s && dc>=tc+s)
        ChessBoard(tr,tc+s,dr,dc,s);            //特殊方格在此象限中
    else                                        //此象限中无特殊方格
    {   board[tr+s-1][tc+s]=t;                  //用 t 号 L 形骨牌覆盖左下角
        ChessBoard(tr,tc+s,tr+s-1,tc+s,s);      //将左下角作为特殊方格继续处理该象限
    }
    //处理左下角象限
```

```
    if(dr>=tr+s && dc<tc+s)                    //特殊方格在此象限中
        ChessBoard(tr+s,tc,dr,dc,s);
    else                                       //此象限中无特殊方格
    {   board[tr+s][tc+s-1]=t;                 //用t号L形骨牌覆盖右上角
        ChessBoard(tr+s,tc,tr+s,tc+s-1,s);     //将右上角作为特殊方格继续处理该象限
    }
    //处理右下角象限
    if(dr>=tr+s && dc>=tc+s)                   //特殊方格在此象限中
        ChessBoard(tr+s,tc+s,dr,dc,s);
    else                                       //此象限中无特殊方格
    {   board[tr+s][tc+s]=t;                   //用t号L形骨牌覆盖左上角
        ChessBoard(tr+s,tc+s,tr+s,tc+s,s);     //将左上角作为特殊方格继续处理该象限
    }
}
void main()
{   k=3;
    x=1; y=2;
    int size=1<<k;
    ChessBoard(0, 0, x, y, size);
    for(int i=0; i<size; i++)                  //输出覆盖方案
    {   for(int j=0; j<size;j++)
            printf("%4d",board[i][j]);
        printf("\n");
    }
}
```

上述程序的执行结果如图 3.12 所示，这里 $k=3$，其中值相同的 3 个方格为一个 L 形骨牌，值为 0 的方格是特殊方格。

3	3	4	4	8	8	9	9
3	2	0	4	8	7	7	9
5	2	2	6	10	10	7	11
5	5	6	6	1	10	11	11
13	13	14	1	1	18	19	19
13	12	14	14	18	18	17	19
15	12	12	16	20	17	17	21
15	15	16	16	20	20	21	21

图 3.12　一种棋盘覆盖方案

【算法分析】 用 $T(k)$ 表示 $2^k \times 2^k (k \geqslant 0)$ 的棋盘问题的求解时间，有：

$$T(k)=1 \quad \text{当 } k=0$$
$$T(k)=4T(k-1) \quad \text{当 } k>0$$

求出 $T(k)=O(4^k)$。

3.4.3 求解循环日程安排问题

扫一扫

视频讲解

【问题描述】 设有 $n=2^k$ 个选手要进行网球循环赛，设计一个满足以下要求的比赛日程表：

（1）每个选手必须与其他 $n-1$ 个选手各赛一次。

（2）每个选手一天只能赛一次。

（3）循环赛在 $n-1$ 天之内结束。

【问题求解】 按问题要求可将比赛日程表设计成一个 n 行 $n-1$ 列的二维表，其中第 i 行、第 j 列表示和第 i 个选手在第 j 天比赛的选手。

假设 n 位选手被顺序编号为 1、2、…、$n(2^k)$。当 $k=1$、2、3 时比赛日程表如图 3.13 所示，其中第 1 列是增加的，取值为 1～n 对应各位选手，这样比赛日程表变成一个 n 行 n 列的二维表。

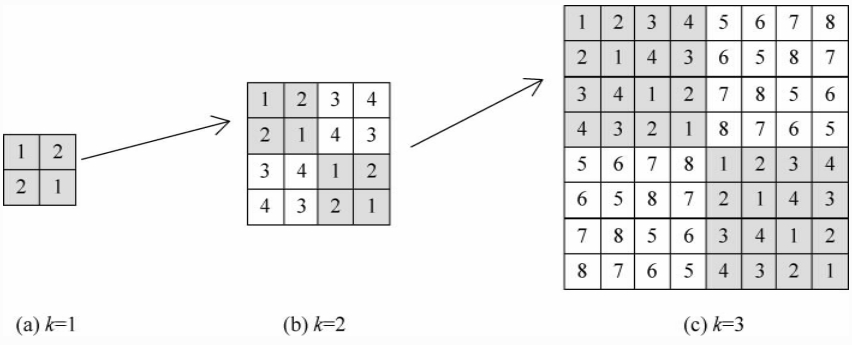

图 3.13 $k=1$～3 的比赛日程表

从中可以看出规律，$k=1$ 只有两个选手时比赛安排十分简单，而 $k=2$ 时可以基于 $k=1$ 的结果进行安排，$k=3$ 时可以基于 $k=2$ 的结果进行安排。

看一看 $k=3$（即 8 个选手）的比赛日程表，右下角（4 行 4 列）的值等于左上角的值，左下角（4 行 4 列）的值等于右上角的值。

$k=3$ 的左上角（4 行 4 列）的值等于 $k=2$（即 4 个选手）的比赛日程表。

$k=3$ 的左下角（4 行 4 列）的值等于 $k=3$ 的左上角对应元素加上数字 4。

因此，采用分治策略可以将所有的选手分为两半，2^k 个选手的比赛日程表就可以通过为 2^{k-1} 个选手设计的比赛日程来决定。将 $n=2^k$ 问题划分为 4 个部分。

（1）左上角：左上角为 2^{k-1} 个选手在前半程的比赛日程（$k=1$ 时直接给出，否则上一轮求出的就是 2^{k-1} 个选手的比赛日程）。

（2）左下角：左下角为另 2^{k-1} 个选手在前半程的比赛日程，由左上角加 2^{k-1} 得到，例如 2^2 个选手比赛，左下角由左上角直接加 $2(2^{k-1})$ 得到，2^3 个选手比赛，左下角由左上角直接加 $4(2^{k-1})$ 得到。

（3）右上角：将左下角直接复制到右上角得到另 2^{k-1} 个选手在后半程的比赛日程。

（4）右下角：将左上角直接复制到右下角得到 2^{k-1} 个选手在后半程的比赛日程。

对应的完整程序如下：

```
#include <stdio.h>
#define MAX 101
//问题表示
int k;
//求解结果表示
int a[MAX][MAX];                    //存放比赛日程表(行、列下标为0的元素不用)
void Plan(int k)
{   int i,j,n,t,temp;
    n=2;                            //n从2^1=2开始
    a[1][1]=1; a[1][2]=2;           //求解两个选手的比赛日程,得到左上角元素
    a[2][1]=2; a[2][2]=1;
    for (t=1;t<k;t++)               //迭代处理,依次处理2^2(t=1)、…、2^k(t=k-1)个选手
    {   temp=n;                     //temp=2^t
        n=n*2;                      //n=2^(t+1)
        for (i=temp+1;i<=n;i++ )    //填左下角元素
            for (j=1; j<=temp; j++)
                a[i][j]=a[i-temp][j]+temp;      //左下角元素和左上角元素的对应关系
        for (i=1; i<=temp; i++)                 //填右上角元素
            for (j=temp+1; j<=n; j++)
                a[i][j]=a[i+temp][(j+temp)% n];
        for (i=temp+1; i<=n; i++)               //填右下角元素
            for (j=temp+1; j<=n; j++)
                a[i][j]=a[i-temp][j-temp];
    }
}
void main()
{   k=3;
    int n=1<<k;                     //n等于2的k次方,即n=2^k
    Plan(k);                        //产生n个选手的比赛日程表
    for(int i=1; i<=n; i++)         //输出比赛日程表
    {   for(int j=1; j<=n; j++)
            printf("%4d",a[i][j]);
        printf("\n");
    }
}
```

这里 $k=3$，执行程序的输出结果如图 3.13(c)所示。

【算法分析】 用 $T(k)$表示 2^k 个选手网球循环赛问题的求解时间，有：

$$T(k)=1 \quad \text{当 } k=1$$
$$T(k)=4T(k-1) \quad \text{当 } k>1$$

求出 $T(k)=O(4^k)$。

3.5 求解大整数乘法和矩阵乘法问题

3.5.1 求解大整数乘法问题

【问题描述】 设 X 和 Y 都是 n(为了简单,假设 n 为 2 的幂,且 X、Y 均为正数)位的二进制整数,现在要计算它们的乘积 $X\times Y$。当位数 n 很大时可以用传统方法来设计一个计算乘积 $X\times Y$ 的算法,但是这样做计算步骤太多,显得效率较低,此时可以采用分治法来设计一个更有效的大整数乘积算法。

【问题求解】 先将 n 位的二进制整数 X 和 Y 各分为两段,每段的长为 $n/2$ 位,如图 3.14 所示。

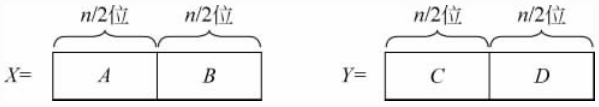

图 3.14 大整数 X 和 Y 的分段

由此,$X=A\times 2^{n/2}+B$,$Y=C\times 2^{n/2}+D$。这样,X 和 Y 的乘积如下:

$$\begin{aligned} X\times Y&=(A\times 2^{n/2}+B)\times(C\times 2^{n/2}+D)\\ &=A\times C\times 2^{n}+(A\times D+C\times B)\times 2^{n/2}+B\times D \end{aligned}$$

如果这样计算 $X\times Y$,则必须进行 4 次 $n/2$ 位整数的乘法($A\times C$、$A\times D$、$B\times C$ 和 $B\times D$),以及 3 次不超过 n 位的整数加法,此外还要做两次移位(分别对应乘 2^n 和乘 $2^{n/2}$)。这些加法和移位共用 $O(n)$ 步运算。设 $T(n)$ 是两个 n 位整数相乘所需的运算总数,则有以下递推式:

$$\begin{aligned} &T(n)=O(1) && \text{当 } n=1 \text{ 时}\\ &T(n)=4T(n/2)+O(n) && \text{当 } n>1 \text{ 时} \end{aligned}$$

由此可得 $T(n)=O(n^2)$。

这种分治法求解 $X\times Y$ 对应的完整程序如下(注意当 n 很大时必须用整型数组来存放 X 和 Y 的各位):

```
#include <stdio.h>
#include <math.h>
#define MAXN 20                          //最多的位数
void Left(int A[],int B[],int n)         //取 A 的左边(高位)n/2 位
{   int i;
    for (i=0;i<MAXN;i++)
        B[i]=0;
    for (i=n/2;i<=n;i++)
        B[i-n/2]=A[i];
}
void Right(int A[],int B[],int n)        //取 A 的右边(低位)n/2 位
```

```
{   int i;
    for (i=0;i<MAXN;i++)
        B[i]=0;
    for (i=0;i<n/2;i++)
        B[i]=A[i];
    B[i]='\0';
}
long Trans2to10(int A[])                    //二进制数转换成十进制数
{   int i;
    long s=A[0],x=1;
    for (i=1;i<MAXN;i++)
    {   x=2*x;
        s+=A[i]*x;
    }
    return s;
}
void Trans10to2(int x,int A[])              //将十进数转换成二进制数
{   int i,j=0;
    while (x>0)
    {   A[j]=x%2;j++;
        x=x/2;
    }
    for (i=j;i<MAXN;i++)
        A[i]=0;
}
void disp(int A[])                          //从高位到低位输出二进制数 A
{   int i;
    for (i=MAXN-1;i>=0;i--)
        printf("%d",A[i]);
    printf("\n");
}
void MULT(int X[],int Y[],int Z[],int n)    //求 Z=X*Y
{   int i;
    long e,e1,e2,e3,e4;
    int A[MAXN],B[MAXN],C[MAXN],D[MAXN];
    int m1[MAXN],m2[MAXN],m3[MAXN],m4[MAXN];
    for (i=0;i<MAXN;i++)                    //Z 初始化为 0
        Z[i]=0;
    if (n==1)                               //递归出口
    {   if (X[0]==1 && Y[0]==1)Z[0]=1;
        else Z[0]=0;
    }
    else
    {   Left(X,A,n);                        //A 取 X 的左边 n/2 位
        Right(X,B,n);                       //B 取 X 的右边 n/2 位
        Left(Y,C,n);                        //C 取 Y 的左边 n/2 位
        Right(Y,D,n);                       //D 取 Y 的右边 n/2 位
        MULT(A,C,m1,n/2);                   //m1=AC
        MULT(A,D,m2,n/2);                   //m2=AD
        MULT(B,C,m3,n/2);                   //m3=BC
```

```
        MULT(B,D,m4,n/2);                    //m4=BD
        e1=Trans2to10(m1);                   //将 m1 转换成十进制数 e1
        e2=Trans2to10(m2);                   //将 m2 转换成十进制数 e2
        e3=Trans2to10(m3);                   //将 m3 转换成十进制数 e3
        e4=Trans2to10(m4);                   //将 m4 转换成十进制数 e4
        e=e1*(int)pow(2,n)+(e2+e3)*(int)pow(2,n/2)+e4;
        Trans10to2(e,Z);                     //将 e 转换成二进制数 Z
    }
}
void trans(char a[],int n,int A[])           //将字符串 a 转换为整数数组 A
{   int i;
    for (i=0;i<n;i++)
        A[i]=int(a[n-1-i]-'0');
    for (i=n;i<MAXN;i++)
        A[i]=0;
}
void main()
{   long e;
    char a[]="10101100";                     //两个参与运算的二进制数
    char b[]="10010011";
    int X[MAXN],Y[MAXN],Z[MAXN];
    int n=8;
    trans(a,n,X);                            //将 a 转换成整数数组 X
    trans(b,n,Y);                            //将 b 转换成整数数组 Y
    printf("X:"); disp(X);                   //输出 X
    printf("Y:"); disp(Y);                   //输出 Y
    printf("Z=X*Y\n");
    MULT(X,Y,Z,n);                           //求 Z=X*Y
    printf("Z:"); disp(Z);                   //输出 Z
    e=Trans2to10(Z);                         //将 Z 转换成十进制数 e
    printf("Z 对应的十进制数:%ld\n",e);
    printf("验证正确性:\n");
    long x,y,z;
    x=Trans2to10(X);                         //将 X 转换成十进制数 x
    y=Trans2to10(Y);                         //将 X 转换成十进制数 y
    printf("X 对应的十进制数 x:%ld\n",x);
    printf("Y 对应的十进制数 y:%ld\n",y);
    printf("z=x*y\n");
    z=x*y;                                   //求 z=x*y
    printf("求解结果 z:%d\n",z);
}
```

本程序的执行结果如下：

```
X:00000000000010101100
Y:00000000000010010011
Z=X*Y
Z:00000110001011000100
Z 对应的十进制数:25284
```

```
验证正确性:
X 对应的十进制数 x:172
Y 对应的十进制数 y:147
z=x*y
求解结果 z:25284
```

【算法改进】 上述算法计算 X 和 Y 的乘积并不比小学生的方法更有效，要想减少算法的计算复杂性，必须减少乘法次数，为此把 $X\times Y$ 写成另一种形式：

$$X\times Y=A\times C\times 2^n+[(A-B)\times(D-C)+A\times C+B\times D]\times 2^{n/2}+B\times D$$

虽然该式看起来比前式复杂一些，但它仅需做 3 次 $n/2$ 位整数的乘法（$A\times C$、$B\times D$ 和 $(A-B)\times(D-C)$），6 次加、减法和两次移位，由此可以推出 $T(n)=O(n^{\log_2 3})=O(n^{1.59})$。

3.5.2 求解矩阵乘法问题

【问题描述】 对于两个 $n\times n$ 的矩阵 $\boldsymbol{A}$ 和 $\boldsymbol{B}$，计算 $\boldsymbol{C}=\boldsymbol{A}\times\boldsymbol{B}$。

【问题求解】 常用的计算公式是 $C_{ij}=\sum_{k=1}^{n}A_{ik}B_{kj}$，对应算法的时间复杂度为 $O(n^3)$。

那么是否存在更有效的算法呢？假设 $n=2^k$，考虑采用分治法思路，当 $n\geqslant 2$ 时将 $\boldsymbol{A}$、$\boldsymbol{B}$ 分成 4 个 $n/2\times n/2$ 的矩阵：

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

利用块矩阵的乘法，矩阵 $\boldsymbol{C}$ 可表示为：

$$\boldsymbol{C}=\begin{pmatrix}A_{11}B_{11}+A_{12}B_{21}&A_{11}B_{12}+A_{12}B_{22}\\A_{21}B_{11}+A_{22}B_{21}&A_{21}B_{12}+A_{22}B_{22}\end{pmatrix}$$

因此原问题可以划分成计算 8 个子问题的乘积问题，两个 $n\times n$ 矩阵乘积的计算量是两个 $n/2\times n/2$ 矩阵乘积计算量的 8 倍，再加上 $n/2\times n/2$ 阶矩阵相加的 4 倍，后者最多需要 $O(n^2)$，因此有：

$$\begin{aligned}&T(n)=O(1) && \text{当 } n=1 \text{ 时}\\&T(n)=8T(n/2)+O(n^2) && \text{当 } n>1 \text{ 时}\end{aligned}$$

可以推导出 $T(n)=O(n^3)$。也就是说，它跟前面介绍的两个矩阵直接相乘的计算量没有什么差别。那么是否可以算得更快呢？

Strassen 通过研究分析提出了 Strassen 算法，其思路如下。

要计算矩阵乘积：$\boldsymbol{C}=\begin{pmatrix}A_{11}&A_{12}\\A_{21}&A_{22}\end{pmatrix}\begin{pmatrix}B_{11}&B_{12}\\B_{21}&B_{22}\end{pmatrix}$

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：

$$\begin{aligned}d_1&=(A_{11}+A_{22})(B_{11}+B_{22})\\d_2&=(A_{21}+A_{22})B_{11}\end{aligned}$$

$$d_3=A_{11}(B_{12}-B_{22})$$
$$d_4=A_{22}(B_{21}-B_{11})$$
$$d_5=(A_{11}+A_{12})B_{22}$$
$$d_6=(A_{21}-A_{11})(B_{11}+B_{12})$$
$$d_7=(A_{12}-A_{22})(B_{21}+B_{22})$$

【算法分析】 由上面可知，两个 $n\times n$ 矩阵乘积的计算量是两个 $n/2\times n/2$ 矩阵乘积计算量的 7 倍，再加上它们进行加或减运算的 18 倍，加减运算共需要 $O(n^2)$，因此有：

$$T(n)=O(1) \qquad \text{当 } n=1 \text{ 时}$$
$$T(n)=7T(n/2)+O(n^2) \qquad \text{当 } n>1 \text{ 时}$$

可以推导出 $T(n)=O(n^{\log_2 7})=O(n^{2.81})$，因此 Strassen 算法的效率更高。

3.6 并行计算简介

3.6.1 并行计算概述

传统计算机是串行结构，每一时刻只能按一条指令对一个数据进行操作，在传统计算机上设计的算法称为串行算法。并行算法是用多台处理器联合求解问题的方法和步骤，其执行过程是将给定的问题首先分解成若干个尽量相互独立的子问题，然后使用多台计算机同时求解它，从而最终求得原问题的解。

为利用并行计算，通常计算问题表现出以下特征：

(1) 将工作分离成离散部分有助于同时解决。例如，对于分治法设计的串行算法，可以将各个独立的子问题并行求解，最后合并成整个问题的解，从而转化为并行算法。

(2) 随时并及时地执行多个程序指令。

(3) 多计算资源下解决问题的耗时要少于单个计算资源下的耗时。

3.6.2 并行计算模型

并行计算模型通常指从并行算法的设计和分析出发，将各种并行计算机(至少某一类并行计算机)的基本特征抽象出来，形成一个抽象的计算模型。从更广的意义上说，并行计算模型为并行计算提供了硬件和软件界面，在该界面的约定下，并行系统硬件设计者和软件设计者可以开发对并行性的支持机制，从而提高系统的性能。

并行算法设计是基于并行计算模型的，下面简要介绍目前常见的两种并行计算模型。

1. PRAM 模型

PRAM(Parallel Random Access Machine，随机存取并行机器)模型也称为共享存储的 SIMD(单指令流多数据流)模型，是一种抽象的并行计算模型，它是从串行的 RAM 模型直接发展起来的。在这种模型中，假定有一个无限大容量的共享存储器，并且有多个功能相同的处理器，且它们都具有简单的算术运算和逻辑判断功能，在任意时刻各个处理器可以访问

共享存储单元。

2. BSP 模型

BSP(Bulk Synchronous Parallel,整体同步并行)模型是分布存储的 MIMD(多指令流多数据流)计算模型,由哈佛大学的 Viliant 和牛津大学的 Bill McColl 提出。

一台 BSP 计算机由 n 个处理器/存储器(结点)组成,通过通信网络进行互联,如图 3.15 所示。

一个 BSP 程序有 n 个进程,每个进程驻留在一个结点上,程序按严格的超步(可以理解为并行计算中子问题的求解)顺序执行,如图 3.16 所示。超步间采用路障同步,每个超步分成以下有序的 3 个部分。

(1) 计算:一个或多个处理器执行若干个局部计算操作,操作的所有数据只能是局部存储器中的数据。一个进程的计算与其他进程无关。

(2) 通信:处理器之间相互交换数据,通信总是以点对点的方式进行。

(3) 同步:确保通信过程中交换的数据被传送到目的处理器上,并使一个超步中的计算和通信操作全部完成后才能开始下一个超步中的任何动作。

BSP 模型总的执行时间等于各超步执行时间之和。

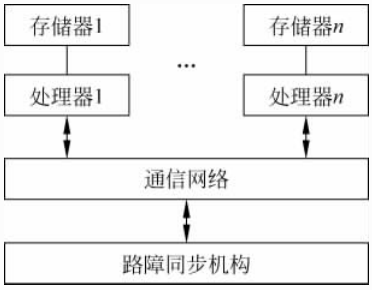

图 3.15 BSP 模型

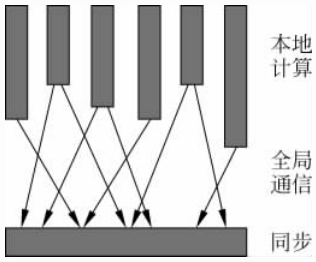

图 3.16 BSP 的一个超步

3.6.3 快速排序的并行算法

基于 BSP 模型,快速排序算法并行化的一个简单思想是对每次划分后所得到的两个序列分别使用两个处理器完成递归排序。

例如对一个长为 n 的序列首先划分得到两个长为 $n/2$ 的序列,将其交给两个处理器分别处理;然后进一步划分得到 4 个长为 $n/4$ 的序列,再分别交给 4 个处理器处理;如此递归下去最终得到排好序的序列。当然这里说的是理想的划分情况,如果划分步骤不能达到平均分配的目的,那么排序的效率会相对较差。

以下算法描述了使用 2^m 个处理器完成对 n 个输入数据(即 $a[0..n-1]$)排序的并行算法:

```
void ParaQuickSort(int a[],int i,int j,int m,int id)
{   if((j-i<=k) ||(m=0))                    //若排序数据个数足够少或 m=0
        Pid执行 QuickSort(a,i,j);           //在 Pid处理器上直接执行传统快速排序算法
    else
    {   Pid执行 r=Partition(a,i,j);         //在 Pid处理器上执行一趟划分
```

```
            Pid 发送 a[r+1,m-1]数据到 Pid+2^(m-1) ;
            ParaQuickSort(a,i,r-1,m-1,id);
            ParaQuickSort(a,r+1,j,m-1,id+2^(m-1));
            Pid+2^(m-1) 发送 a[r+1,m-1]到 Pid ;
        }
    }
    void main()
    {
        ParaQuickSort(data,0,n-1,m,0)
    }
```

在最好情况下该并行算法形成一个高度为$\lceil \log_2 n \rceil$的排序树，其计算时间复杂度为$O(n)$。和串行算法一样，在最坏情况下时间复杂度降为$O(n^2)$。正常情况下该算法的平均时间复杂度为$O(n)$。

3.7 练习题

1. 分治法的设计思想是将一个难以直接解决的大问题分割成规模较小的子问题，分别解决子问题，最后将子问题的解组合起来形成原问题的解，这要求原问题和子问题(　　)。

A. 问题规模相同，问题性质相同　　B. 问题规模相同，问题性质不同

C. 问题规模不同，问题性质相同　　D. 问题规模不同，问题性质不同

2. 在寻找n个元素中第k小的元素的问题中，如采用快速排序算法思想，运用分治法对n个元素进行划分，如何选择划分基准？下面(　　)答案最合理。

A. 随机选择一个元素作为划分基准

B. 取子序列的第一个元素作为划分基准

C. 用中位数的中位数方法寻找划分基准

D. 以上皆可行，但不同方法的算法复杂度上界可能不同

3. 对于下列二分查找算法，正确的是(　　)。

A.

```
int binarySearch(int a[], int n, int x)
{   int low=0, high=n-1;
    while(low<=high)
    {   int mid=(low+high)/2;
        if(x==a[mid]) return mid;
        if(x>a[mid]) low=mid;
            else high=mid;
    }
    return -1;
}
```

B.

```
int binarySearch(int a[], int n, int x)
{   int low=0, high=n-1;
    while(low+1!=high)
    {   int mid=(low+high)/2;
        if(x>=a[mid]) low=mid;
            else high=mid;
    }
    if(x==a[low]) return low;
    else return -1;
}
```

C.

```
int binarySearch(int a[], int n, int x)
{   int low=0, high=n-1;
    while(low<high-1)
    {   int mid=(low+high)/2;
        if(x<a[mid])
            high=mid;
        else low=mid;
    }
    if(x==a[low]) return low;
    else return -1;
}
```

D.

```
int binarySearch(int a[], int n, int x)
{   if(n>0 && x>= a[0])
    {   int low = 0, high = n-1;
        while(low<high)
        {   int mid=(low+high+1)/2;
            if(x<a[mid])
                high=mid-1;
            else low=mid;
        }
        if(x==a[low]) return low;
    }
    return -1;
}
```

4. 快速排序算法是根据分治策略来设计的，简述其基本思想。

5. 假设含有 n 个元素的待排序数据 a 恰好是递减排列的，说明调用 QuickSort(a,0,$n-1$)递增排序的时间复杂度为 $O(n^2)$。

6. 以下哪些算法采用分治策略：

(1) 堆排序算法；

(2) 二路归并排序算法；

(3) 折半查找算法；

(4) 顺序查找算法。

7. 适合并行计算的问题通常表现出哪些特征？

8. 设有两个复数 $x=a+bi$ 和 $y=c+di$。复数乘积 xy 可以使用 4 次乘法来完成，即 $xy=(ac-bd)+(ad+bc)i$。设计一个仅用 3 次乘法来计算乘积 xy 的方法。

9. 有 4 个数组 a、b、c 和 d，都已经排好序，说明找出这 4 个数组的交集的方法。

10. 设计一个算法，采用分治法求一个整数序列中的最大和最小元素。

11. 设计一个算法，采用分治法求 x^n。

12. 假设二叉树采用二叉链存储结构进行存储，设计一个算法采用分治法求一棵二叉树 bt 的高度。

13. 假设二叉树采用二叉链存储结构进行存储，设计一个算法采用分治法求一棵二叉树 bt 中度为 2 的结点个数。

14. 有一种二叉排序树，其定义为空树是一棵二叉排序树，若不空，左子树中的所有结点值小于根结点值，右子树中的所有结点值大于根结点值，并且左、右子树都是二叉排序树。现在该二叉排序树采用二叉链存储，采用分治法设计查找值为 x 的结点地址，并分析算法的最好平均时间复杂度。

15. 设有 n 个互不相同的整数，按递增顺序存放在数组 $a[0..n-1]$中，若存在一个下标 $i(0\leqslant i<n)$，使得 $a[i]=i$，设计一个算法以 $O(\log_2 n)$时间找到这个下标 i。

16. 请模仿二分查找过程设计一个三分查找算法，分析其时间复杂度。

17. 对于大于 1 的正整数 n，可以分解为 $n=x_1\times x_2\times\cdots\times x_m$，其中 $x_i\geqslant 2$。例如，$n=12$ 时有 8 种不同的分解式，即 $12=12$，$12=6\times 2$，$12=4\times 3$，$12=3\times 4$，$12=3\times 2\times 2$，$12=2\times 6$，$12=2\times 3\times 2$，$12=2\times 2\times 3$，设计一个算法求 n 的不同分解式的个数。

18. 设计一个基于 BSP 模型的并行算法，假设有 p 台处理器，计算整数数组 $a[0..n-1]$ 的所有元素之和，并分析算法的时间复杂度。

3.8 上机实验题

实验 1. 求解查找假币问题

编写一个实验程序查找假币问题。有 $n(n>3)$个硬币，其中有一个假币，且假币较轻，采用天平称重方式找到这个假币，并给出操作步骤。

实验 2. 求解众数问题

给定一个整数序列，每个元素出现的次数称为重数，重数最大的元素称为众数。编写一个实验程序对递增有序序列 a 求众数。例如 $S=\{1,2,2,2,3,5\}$，多重集 S 的众数是 2，其重数为 3。

实验 3. 求解逆序数问题

给定一个整数数组 $A=(a_0,a_1,\cdots,a_{n-1})$，若 $i<j$ 且 $a_i>a_j$，则$<a_i,a_j>$就为一个逆序对。例如数组(3,1,4,5,2)的逆序对有$<3,1>$、$<3,2>$、$<4,2>$、$<5,2>$。编写一个实验程

序采用分治法求 A 中逆序对的个数，即逆序数。

实验 4. 求解半数集问题

给定一个自然数 n，由 n 开始可以依次产生半数集 set(n)中的数如下：

(1) $n \in$ set(n)。

(2) 在 n 的左边加上一个自然数，但该自然数不能超过最近添加的数的一半。

(3) 按此规则进行处理，直到不能再添加自然数为止。

例如，set(6)={6,16,26,126,36,136}，半数集 set(6)中有 6 个元素。编写一个实验程序求给定 n 时对应半数集中元素的个数。

实验 5. 求解一个整数数组划分为两个子数组问题

已知由 $n(n \geqslant 2)$个正整数构成的集合 $A=\{a_k\}(0 \leqslant k<n)$，将其划分为两个不相交的子集 A_1 和 A_2，元素个数分别是 n_1 和 n_2，A_1 和 A_2 中的元素之和分别为 S_1 和 S_2。设计一个尽可能高效的划分算法，满足 $|n_1-n_2|$ 最小且 $|S_1-S_2|$ 最大，算法返回 $|S_1-S_2|$ 的结果。

3.9 在线编程题

在线编程题 1. 求解满足条件的元素对个数问题

【问题描述】 给定 N 个整数 A_i 以及一个正整数 C，问其中有多少对 i、j 满足 $A_i-A_j=C$。

输入描述：第 1 行输入两个空格隔开的整数 N 和 C，第 2～N+1 行每行包含一个整数 A_i。

输出描述：输出一个数表示答案。

输入样例：

```
5 3
2
1
4
2
5
```

样例输出：

```
3
```

在线编程题 2. 求解查找最后一个小于等于指定数的元素问题

【问题描述】 给定一个长度为 n 的单调递增的正整数序列，即序列中的每一个数都比前一个数大，有 m 个询问，每次询问一个 x，问序列中最后一个小于等于 x 的数是什么？

输入描述：给定一个长度为 n 的单调递增的正整数序列，即序列中的每一个数都比前一个数大，有 m 个询问，每次询问一个 x。

输出描述：输出共 m 行，表示序列中最后一个小于等于 x 的数是多少。如果没有，输

出－1。

输入样例：

```
5 3
1 2 3 4 6
5
1
3
```

样例输出：

```
4
1
3
```

数据范围限制：$1 \leqslant n、m \leqslant 100\,000$，序列中的元素和 x 都不超过 10^6。

在线编程题 3. 求解递增序列中与 *x* 最接近的元素问题

【问题描述】 在一个非降序列中查找与给定值 x 最接近的元素。

输入描述：第 1 行包含一个整数 n，为非降序列长度（$1 \leqslant n \leqslant 100\,000$）；第 2 行包含 n 个整数，为非降序列的各个元素，所有元素的大小均在 0～1 000 000 000 范围内；第 3 行包含一个整数 m，为要询问的给定值个数（$1 \leqslant m \leqslant 10\,000$）；接下来 m 行，每行一个整数，为要询问最接近元素的给定值，所有给定值的大小均在 0～1 000 000 000 范围内。

输出描述：输出共 m 行，每行一个整数，为最接近相应给定值的元素值，并保持输入顺序。若有多个元素值满足条件，输出最小的一个。

输入样例：

```
3
2 5 8
2
10
5
```

样例输出：

```
8
5
```

在线编程题 4. 求解按“最多排序”到“最少排序”的顺序排列问题

【问题描述】 一个序列中的“未排序”的度量是相对于彼此顺序不一致的条目对的数量，例如，在字母序列“DAABEC”中该度量为 5，因为 D 大于其右边的 4 个字母，E 大于其右边的 1 个字母。该度量称为该序列的逆序数。序列“AACEDGG”只有一个逆序对（E 和 D），它几乎被排好序了，而序列“ZWQM”有 6 个逆序对，它是未排序的，恰好是反序。

需要对若干个 DNA 序列（仅包含 4 个字母 A、C、G 和 T 的字符串）分类，注意是分类而不是按字母顺序排列，是按照“最多排序”到“最小排序”的顺序排列，所有 DNA 序列的长度

都相同。

输入描述：第1行包含两个整数，$n(0<n\leqslant 50)$表示字符串长度，$m(0<m\leqslant 100)$表示字符串个数；后面是m行，每行包含一个长度为n的字符串。

输出描述：按“最多排序”到“最小排序”的顺序输出所有字符串。若两个字符串的逆序对个数相同，按原始顺序输出它们。

输入样例：

```
10 6
AACATGAAGG
TTTTGGCCAA
TTTGGCCAAA
GATCAGATTT
CCCGGGGGGA
ATCGATGCAT
```

样例输出：

```
CCCGGGGGGA
AACATGAAGG
GATCAGATTT
ATCGATGCAT
TTTTGGCCAA
TTTGGCCAAA
```