算法设计与分析

分治法

讲师：杨垠晖 博士

本讲内容

3.1 分治法概述

3.2 求解排序问题

3.3 求解查找问题

3.4 求解组合问题

3.5 求解大整数乘法和矩阵乘法问题

3.6 并行计算简介

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（比如说规模*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做**分治法**。

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同问题。

（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。

3.1.2 分治法的求解过程

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：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：若子问题规模较小而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：将各个子问题的解合并为原问题的解。

分治法的一般的算法设计框架如下：

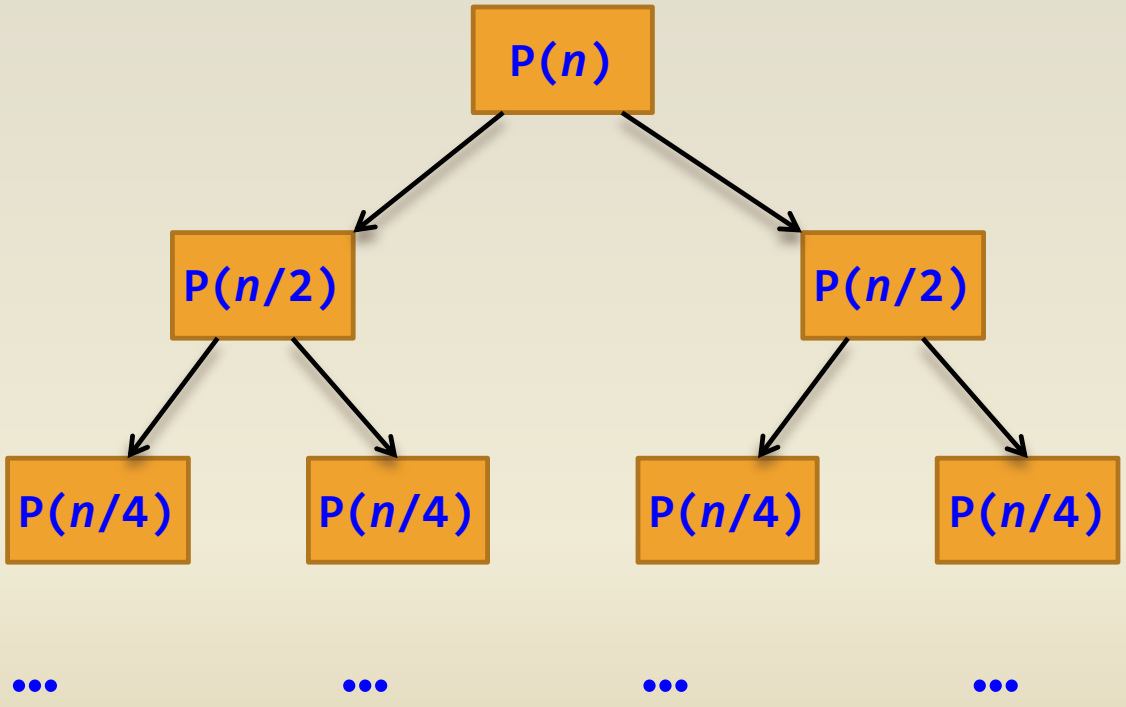
```
divide-and-conquer(P)
{  if |P|≤n0 return adhoc(P);
   将P分解为较小的子问题 P1, P2, …, Pk;
   for(i=1;i<=k;i++)                    //循环处理k次
      yi=divide-and-conquer(Pi);        //递归解决Pi
    return merge(y1, y2, …, yk);        //合并子问题
}
```

根据分治法的分割原则，原问题应该分为多少个子问题才较适宜？各个子问题的规模应该怎样才为适当？

这些问题很难予以肯定的回答。但人们从大量实践中发现，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4.1所示，这种使子问题规模大致相等的做法是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。

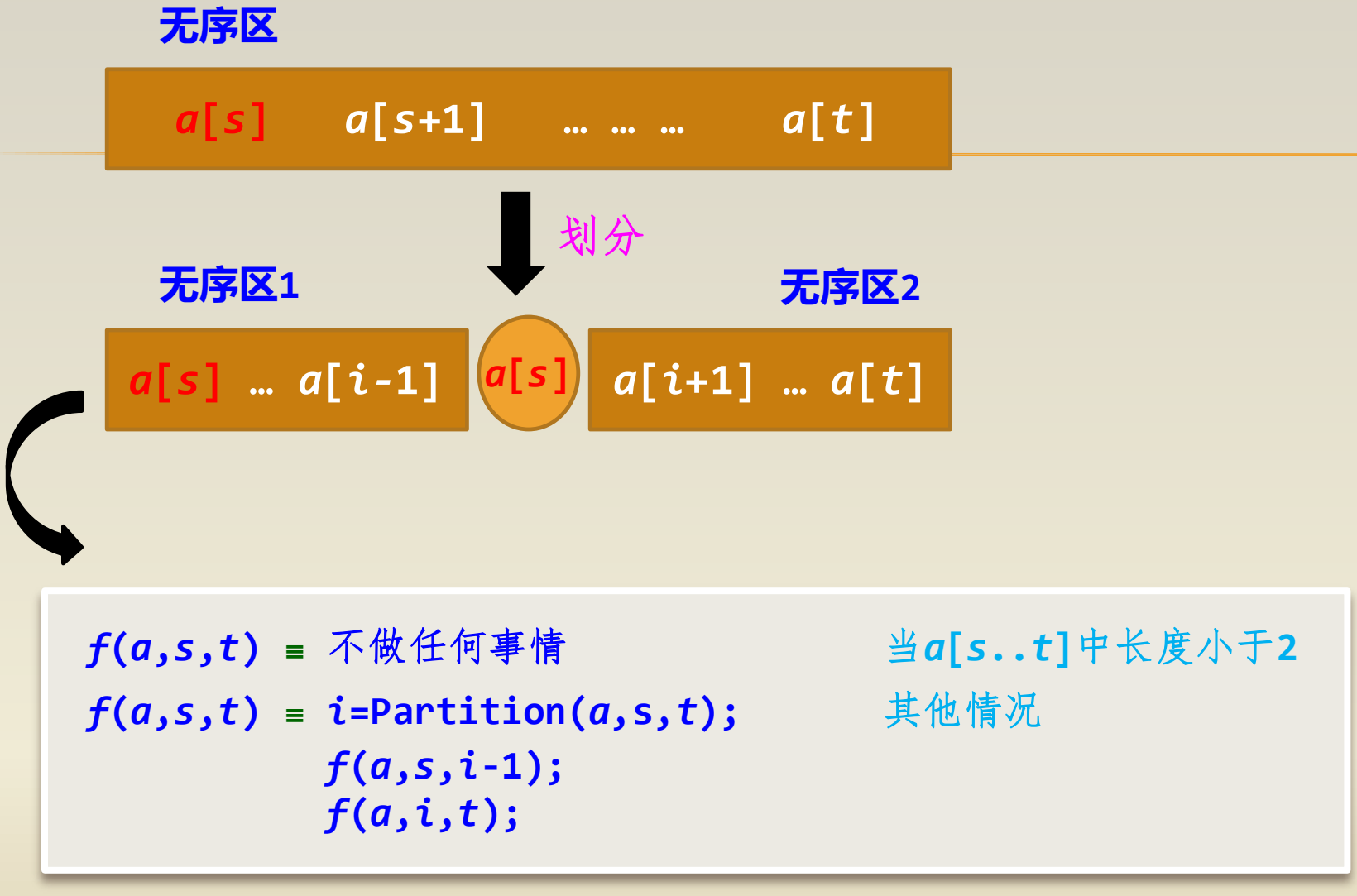


3.2 求解排序问题

3.2.1 快速排序

基本思想：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，这个过程称作划分。

然后对两个子序列分别重复上述过程，直至每个子序列内只有一个记录或空为止。




```
f(a,s,t) ≡ 不做任何事情                    当a[s..t]中长度小于2
f(a,s,t) ≡ i=Partition(a,s,t);             其他情况
           f(a,s,i-1);
           f(a,i,t);
```

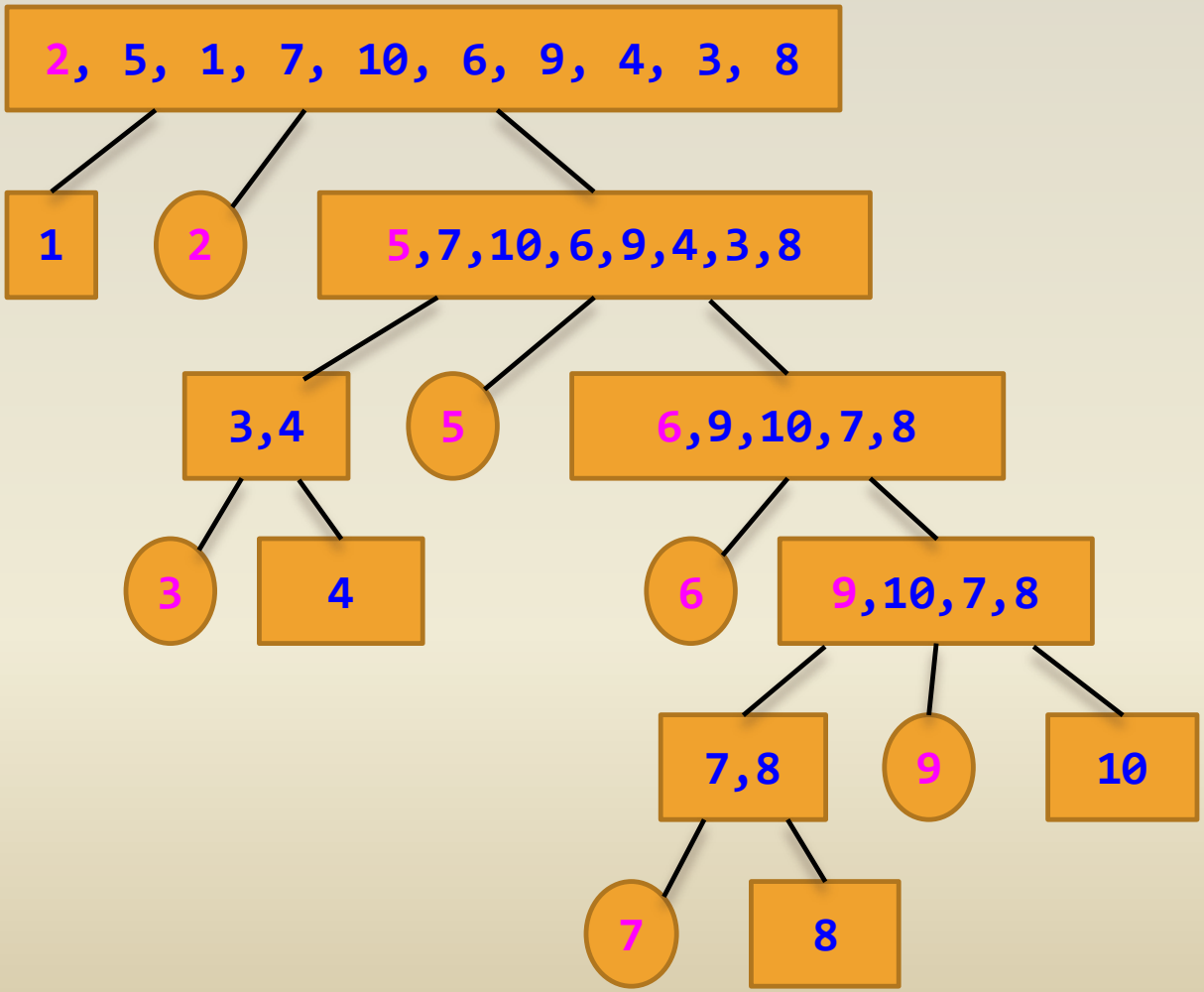
分治策略：

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。

② 求解子问题：若子序列的长度为0或为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，其快速排序过程如下图所示。



快速排序算法：

```
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，直至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;
}
```

快速排序算法：

```
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2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
   }
}
```

【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个记录进行快速排序的过程构成一棵递归树，在这样的递归树中，每一层至多对n个记录进行划分，所花时间为$O(n)$。

当初始排序数据正序或反序时，此时的递归树高度为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记录个数大致相等，此时的递归树高度为$\log_2 n$，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$O(n\log_2 n)$。

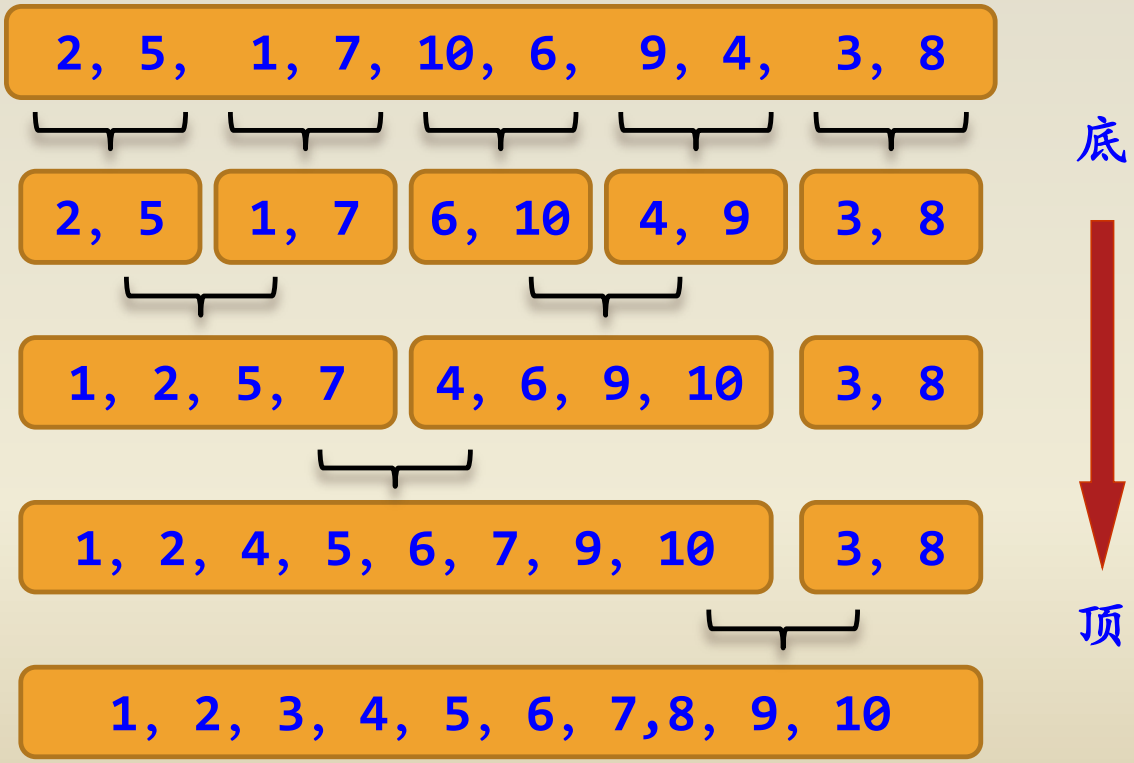
2.2.2 归并排序

归并排序的基本思想是：首先将$a[0..n-1]$看成是n个长度为1的有序表，将相邻的k（$k\geq 2$）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。

若$k=2$，即归并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。

1．自底向上的二路归并排序算法

例如，对于{2，5，1，7，10，6，9，4，3，8}序列，其排序过程如下图所示，图中方括号内是一个有序子序列。



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长度的若干子序列。

② 求解子问题：将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。

```
void Merge(int a[], int low, int mid, int high)
//a[low..mid]和a[mid+1..high]→a[low..high]
{  int *tmpa;
   int i=low, j=mid+1, k=0;
   tmpa=(int *)malloc((high-low+1)*sizeof(int));
   while (i<=mid && j<=high)
     if (a[i]<=a[j])              //将第1子表中的元素放入tmpa中
      {  tmpa[k]=a[i];  i++; k++; }
      else                        //将第2子表中的元素放入tmpa中
      {  tmpa[k]=a[j];  j++; k++; }
   while (i<=mid)                 //将第1子表余下部分复制到tmpa
   {  tmpa[k]=a[i]; i++; k++; }
   while (j<=high)                //将第2子表余下部分复制到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tmpa所占内存空间
}
```

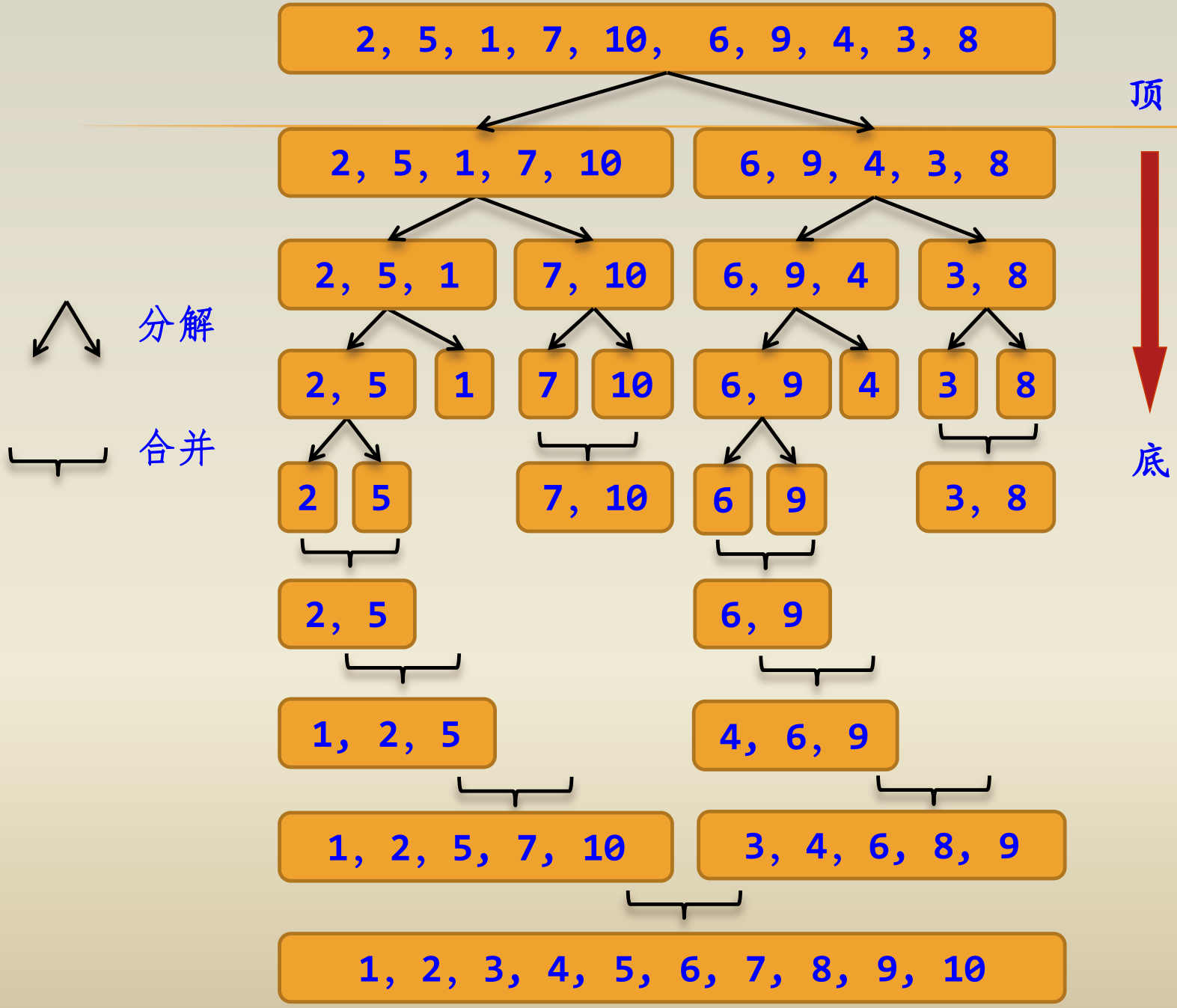
```
void MergePass(int a[], int length, int n)
//一趟二路归并排序
{  int i;
   for (i=0;i+2*length-1<n;i=i+2*length)    //归并length长的两相邻子表
        Merge(a, i, i+length-1, i+2*length-1);

   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);
}
```

【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

例如，对于{2，5，1，7，10，6，9，4，3，8}序列，说明其自顶向下的二路归并排序的过程。


2, 5, 1, 7, 10, 6, 9, 4, 3, 8
顶
2, 5, 1, 7, 10
6, 9, 4, 3, 8
2, 5, 1
7, 10
6, 9, 4
3, 8
分解
2, 5
1
7
10
6, 9
4
3
8
合并
底
2
5
7, 10
6
9
3, 8
2, 5
6, 9
1, 2, 5
4, 6, 9
1, 2, 5, 7, 10
3, 4, 6, 8, 9
1, 2, 3, 4, 5, 6, 7, 8, 9, 10

设归并排序的当前区间是a[low..high]，则递归归并的两个步骤如下：

① 分解：将序列a[low..high]一分为二，即求mid=(low+high)/2；递归地对两个子序列a[low..mid]和a[mid+1..high]进行继续分解。其终结条件是子序列长度为1（因为一个元素的子表一定是有序表）。

② 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);       //将两子序列合并，见前面的算法
    }
}
```

递归出口为序列长度为1或者0!

【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)的执行时间为$O(n)$，所以得到以下递推式：

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

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*元素的无序序列，求这个序列中最大和次大的两个不同的元素。

例如：（2，5，1，4，6，3），最大元素为6，次大元素为5。

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

（1）a[low.high]中只有一个元素：则max1=a[low]，max2=-INF（-∞）（要求它们是不同的元素）。

（2）a[low.high]中只有两个元素：则max1=MAX{a[low]，a[high]}，max2=MIN{a[low]，a[high]}。

（3）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]左右两个区间（注意左区间包含a[mid]元素）。

求出左区间最大元素lmax1和次大元素lmax2，求出右区间最大元素rmax1和次大元素rmax2。

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

```
void solve(int a[],int low,int high,int &max1,int &max2)
{   if (low==high)                  //区间只有一个元素
    {   max1=a[low];   max2=-INF;  }
    else if (low==high-1)           //区间只有两个元素
    {   max1=max(a[low],a[high]); max2=min(a[low],a[high]); }
    else                            //区间有两个以上元素
    {   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为：

$T(1)=T(2)=1$

$T(n)=2T(n/2)+1$　　//合并的时间为O(1)

可以推导出$T(n)=O(n)$。

3.3.2 折半查找

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=⌊(low+high)/2⌋；然后将待查的k值与a[mid].key比较：

（1）若k==a[mid]，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必在位于右子表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。

算法实现：

```
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
            return BinSearch(a, low, mid-1, k);
        else                       //当a[mid]<k时
            return BinSearch(a, mid+1, high, k);
   }
   else return -1;                 //若当前查找区间没有元素时返回-1
}
```

折半查找-你真的会了吗?

据说90%的程序员编写的折半查找程序都有bug!

- 当有序数组中存在相同元素*x*时，如何查找最左边的*x* ？
- 当有序数组中存在相同元素*x*时，如何查找最右边的*x* ？
- 当有序数组中存在相同元素*x*时，如何求*x*的个数 ？

【算法分析】折半查找算法的主要时间花费在元素比较上，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：

$C(n)=1$	当$n=1$
$C(n)\leq 1+C(\lfloor n/2 \rfloor)$	当$n\geq 2$

由此得到：$C(n)\leq \lfloor \log_2 n \rfloor+1$

折半查找的主要时间花在元素比较上，所以算法的时间复杂度为$O(\log_2 n)$。

3.3.3 寻找一个序列中第k小元素

【问题描述】对于给定的含有n元素的无序序列，求这个序列中第k（1≤k≤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]。

采用类似于快速排序的思想。

该元素的下标为$k-1$

对于序列$a[s..t]$，在其中查找第k小元素的过程如下：

将$a[s]$作为基准划分，其对应下标为i。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。

算法实现：

```
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,tmp;
   if (s<t)
   {  tmp=a[s];
      while (i!=j)                  //从区间两端交替向中间扫描，直至i=j为止
      {  while (j>i && a[j]>=tmp) j--;
         a[i]=a[j];                 //将a[j]前移到a[i]的位置
         while (i<j && a[i]<=tmp) i++;
         a[j]=a[i];                 //将a[i]后移到a[j]的位置
      }
      a[i]=tmp;                     //将基准归位

      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];
}
```

【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。

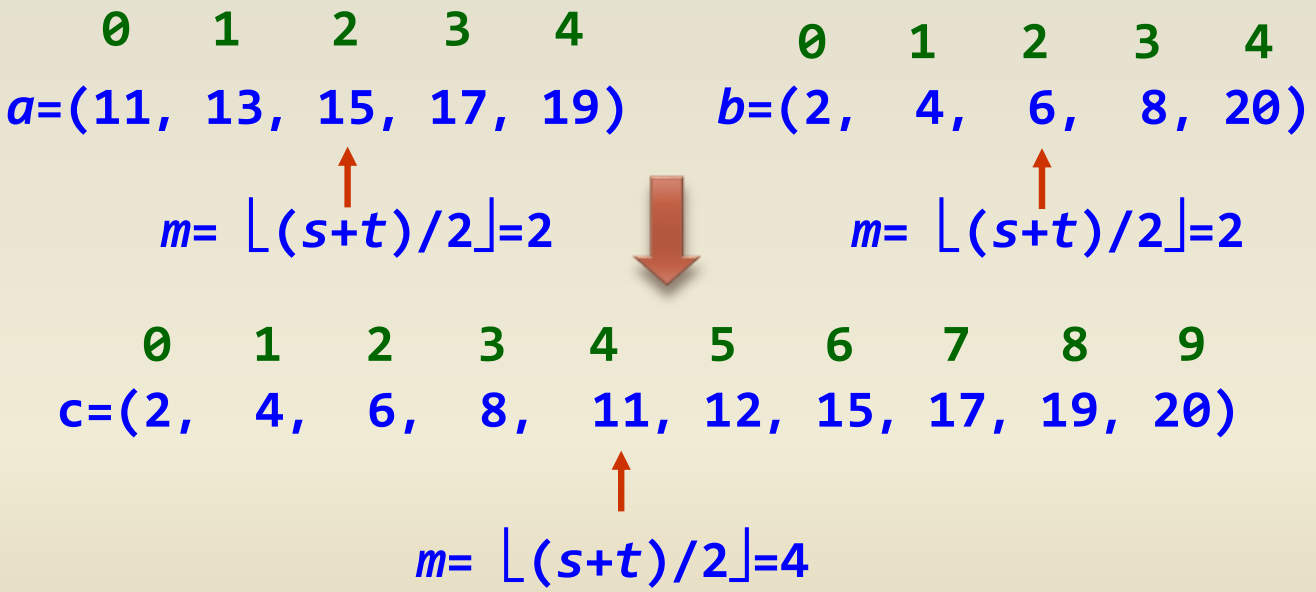
设计一个算法求给定的两个有序序列的中位数。

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。

a=(11，13，15，17，19) b=(2，4，6，8，20)

c=(2，4，6，8，11，12，15，17，19，20)

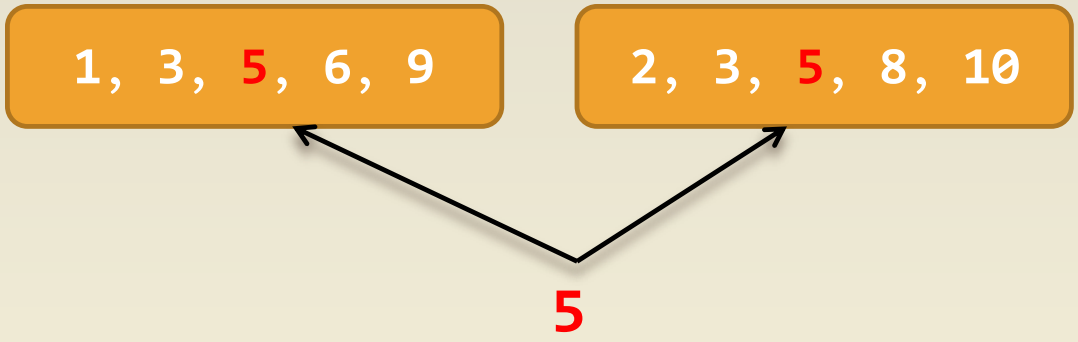
【问题求解】对于含有n个元素的有序序列$a[s..t]$，当n为奇数时，中位数是出现在$m=\lfloor(s+t)/2\rfloor$处；当n为偶数时，中位数下标有$m=\lfloor(s+t)/2\rfloor$（下中位）和$m=\lfloor(s+t)/2\rfloor+1$（上中位）两个。为了简单，仅考虑中位数为$m=\lfloor(s+t)/2\rfloor$处。



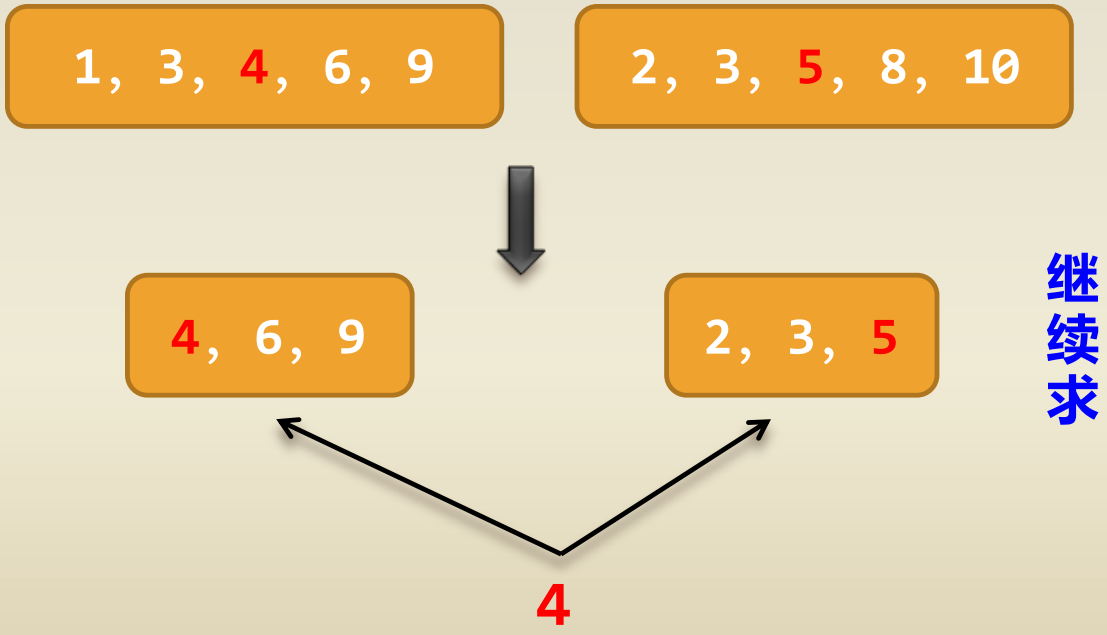
采用二分法求含有n个有序元素的序列a、b的中位数的过程如下：

分别求出a、b的中位数a[m1]和b[m2]：

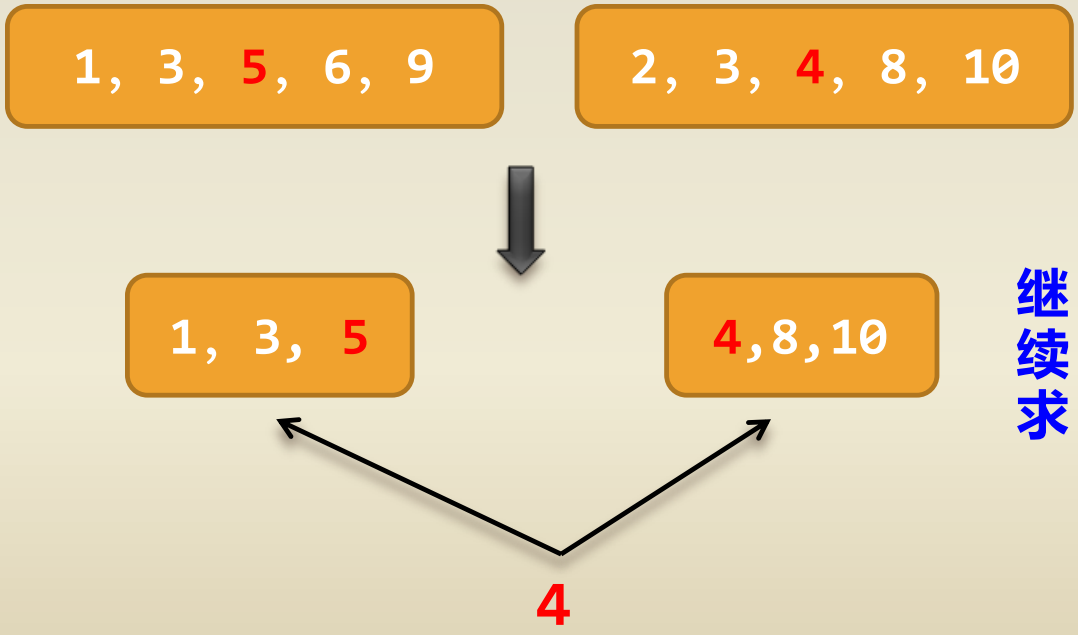
① 若a[m1]=b[m2]，则a[m1]或b[m2]即为所求中位数，算法结束。



② 若a[m1]<b[m2]，则舍弃序列a中前半部分（较小的一半），同时舍弃序列b中后半部分（较大的一半）要求舍弃的长度相等。



③ 若a[m1]>b[m2]，则舍弃序列a中后半部分（较大的一半），同时舍弃序列b中前半部分（较小的一半），要求舍弃的长度相等。



```
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);
       }
    }
}
```

```
// 求a[s…t]序列的前半子序列
void prepart(int& s, int& t)
{
  int m = (s + t) / 2;
  t = m;
}

// 求a[s…t]序列的后半子序列
void postpart(int& s, int& t)
{
  int m = (s + t) / 2;
  if((s+t)%2==0)   //序列中有奇数个元素
    s = m;
  else
    s = m + 1;     //序列中有偶数个元素
}
```

【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$T(n)$，显然有以下递归式：

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

容易推出，$T(n)=O(n)$。

3.4 求解组合问题

3.4.1 求解最大连续子序列和问题

【问题描述】给定一个有n（n≥1）个整数的序列，要求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

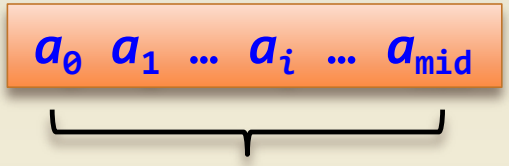
序列（-6，2，4，-7，5，3，2，-1，6，-9，10，-2）的最大子序列和为16。

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。

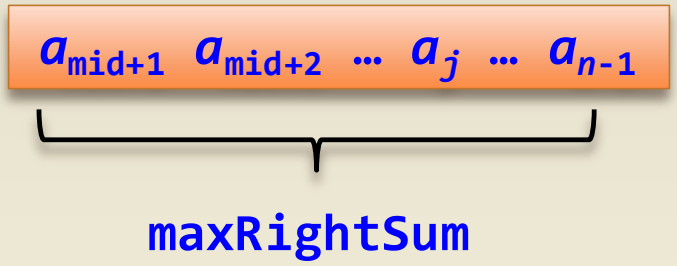
【问题求解】对于含有n个整数的序列$a[0..n-1]$，若$n=1$，表示该序列仅含一个元素，如果该元素大于0，则返回该元素；否则返回0。

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3个地方。

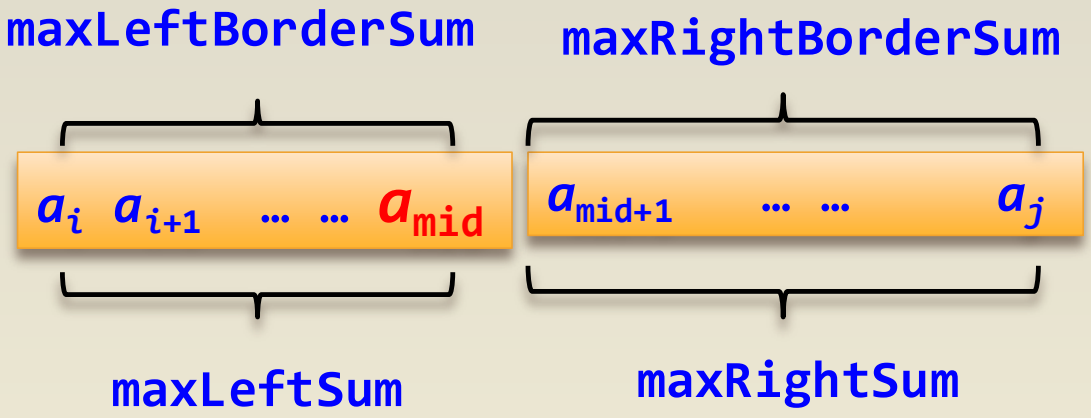
（1）该子序列完全落在左半部即$a[0..mid]$中。采用递归求出其最大连续子序列和maxLeftSum。



（2）该子序列完全落在右半部即a[mid+1..n-1]中。采用递归求出其最大连续子序列和maxRightSum。



（3）该子序列跨越序列a的中部而占据左右两部分。



结果: max3(maxLeftSum,
maxRightSum,
maxLeftBorderSum+maxRightBorderSum)

```
long maxSubSum(int a[], int left, int right)
//求a[left..high]序列中最大连续子序列和
{  int i, j;
   long maxLeftSum, maxRightSum;
   long maxLeftBorderSum, leftBorderSum;
   long maxRightBorderSum, rightBorderSum;

   if (left==right)              //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
    maxRightSum=maxSubSum(a, mid+1, right);          //求右边
    maxLeftBorderSum=0, leftBorderSum=0;
    for (i=mid;i>=left;i--)                          //求出以左边加上a[mid]元素
    {  leftBorderSum+=a[i];                          //构成的序列的最大和
       if (leftBorderSum>maxLeftBorderSum)
            maxLeftBorderSum=leftBorderSum;
    }
    maxRightBorderSum=0, rightBorderSum=0;
    for (j=mid+1;j<=right;j++)                       //求出a[mid]右边元素
    {  rightBorderSum+=a[j];                         //构成的序列的最大和
       if (rightBorderSum>maxRightBorderSum)
            maxRightBorderSum=rightBorderSum;
    }
    return max3(maxLeftSum, maxRightSum,
                maxLeftBorderSum+maxRightBorderSum);
}
```

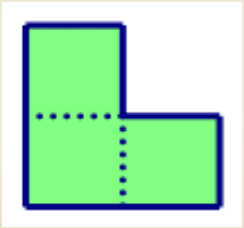
【算法分析】设求解序列$a[0..n-1]$最大连续子序列和的执行时间为$T(n)$，第（1）、（2）两种情况的执行时间为$T(n/2)$，第（3）种情况的执行时间为$O(n)$，所以得到以下递推式：

$T(n)=1$	当$n=1$
$T(n)=2T(n/2)+n$	当$n>1$

容易推出，$T(n)=O(n\log_2 n)$。

3.4.2 求解棋盘覆盖问题

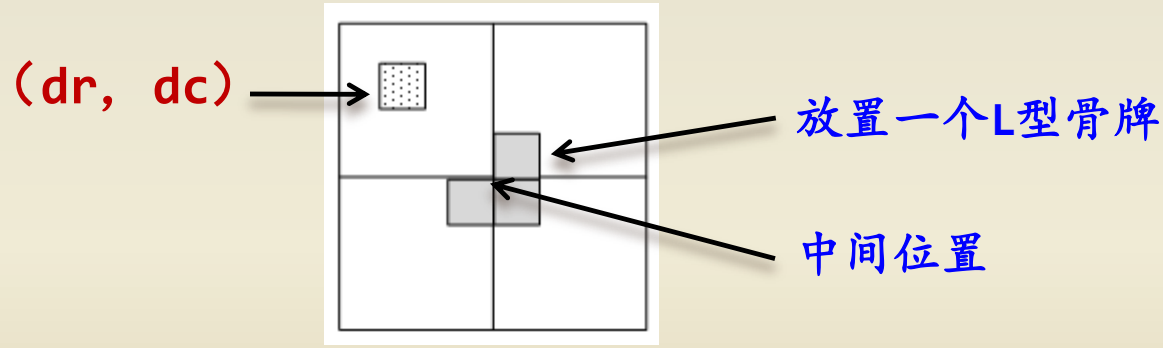
【问题描述】有一个$2^k \times 2^k$（$k>0$）的棋盘，恰好有一个方格与其他方格不同，称之为特殊方格。现在要用如下的L型骨牌覆盖除了特殊方格外的其他全部方格，骨牌可以任意旋转，并且任何两个骨牌不能重叠。请给出一种覆盖方法。



【问题求解】棋盘中的方格数=$2^k \times 2^k = 4^k$，覆盖使用的L型骨牌个数=$(4^k-1)/3$。

采用的方法是：将棋盘划分为4个大小相同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型骨牌。

例如，如下图所示，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型骨牌。



特殊方格在左上角象限

其他情况类似！

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。

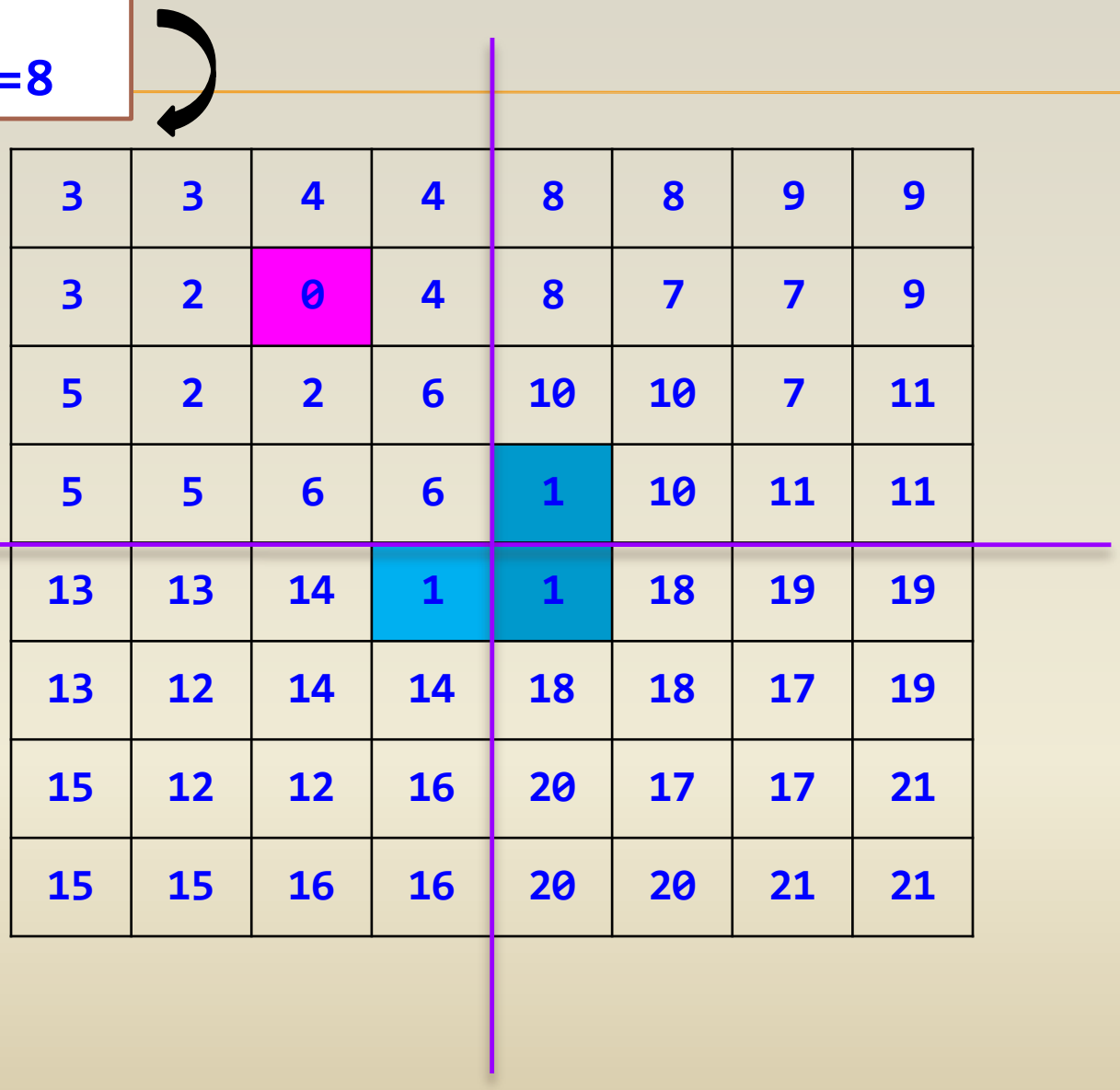
用二维数组board存放覆盖方案，用tile全局变量表示L型骨牌的编号（从整数1开始），board中3个相同的整数表示一个L型骨牌。

```
#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;
```

```
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型骨，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型骨牌覆盖右下角
        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型骨牌覆盖左下角
        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型骨牌覆盖右上角
        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型骨牌覆盖左上角
        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
    }
}
```

$k=3$,
$n=2^3=8$



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

$k=3$,
$n=2^3=8$


❶左上
角象限
❷右上
角象限
❸左下
角象限
❹右下
角象限
3 3 4 4 8 8 9 9
3 2 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

3.4.3 求解循环日程安排问题

【问题描述】设有$n=2^k$个选手要进行网球循环赛，要求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。

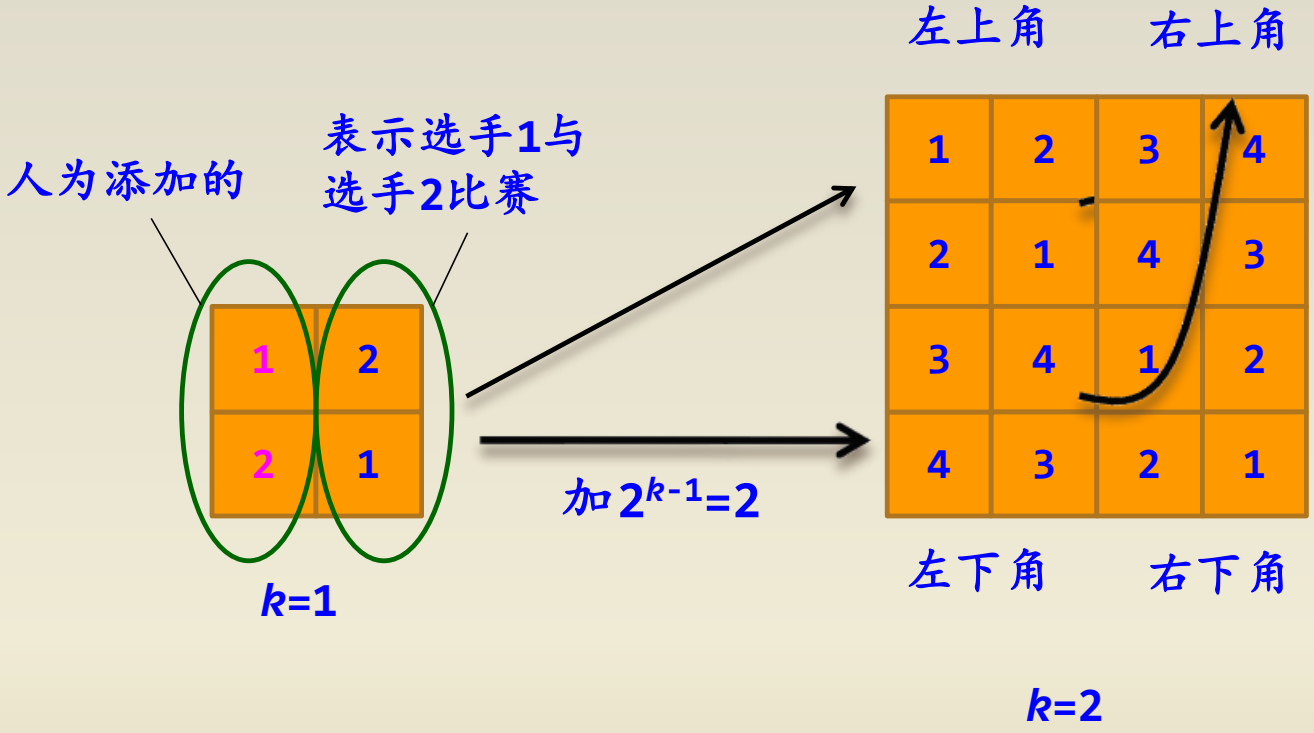
（2）每个选手一天只能赛一次。

（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）。

由k=1创建k=2的过程


左上角
右上角
人为添加的
表示选手1与
选手2比赛
1
2
2
1
加$2^{k-1}=2$
1
2
3
4
2
1
4
3
3
4
1
2
4
3
2
1
左下角
右下角
k=1
k=2

由k=2创建k=3的过程

1	2	3	4
2	1	4	3
3	4	1	2
4	3	2	1

$k=2$

加 $2^{k-1}=4$

左上角　　右上角

1	2	3	4	5	6	7	8
2	1	4	3	6	5	8	7
3	4	1	2	7	8	5	6
4	3	2	1	8	7	6	5
5	6	7	8	1	2	3	4
6	5	8	7	2	1	4	3
7	8	5	6	3	4	1	2
8	7	6	5	4	3	2	1

左下角　　右下角

$k=3$

将$n=2^k$问题划分为4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。

```
#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2个选手比赛日程,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;         //产生左下角元素
        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];
    }
}
```

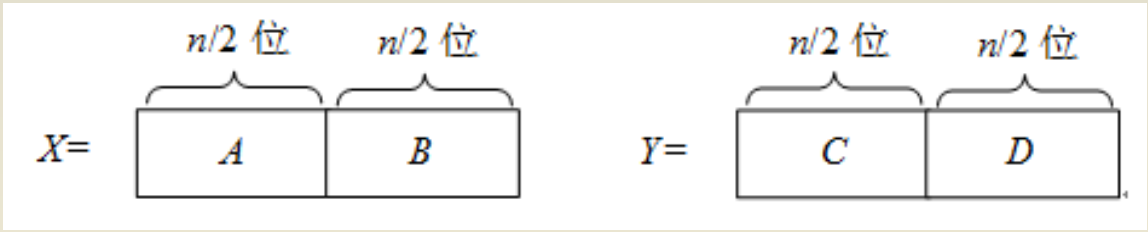
3.5 求解大整数乘法和矩阵乘法问题

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***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***Y*的算法，但是这样做计算步骤太多，显得效率较低。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，如下图所示。



由此，$X=A*2^{n/2}+B$，$Y=C*2^{n/2}+D$。这样，X和Y的乘积为：

$$X*Y=(A*2^{n/2}+B)*(C*2^{n/2}+D)=A*C*2^{n}+(A*D+C*B)*2^{n/2}+B*D$$

如果这样计算$X*Y$，则必须进行4次$n/2$位整数的乘法（$A*C$、$A*D$、$B*C$和$B*D$），以及3次不超过n位的整数加法，此外还要做2次移位（分别对应乘2^n和乘$2^{n/2}$）。所有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：

$$T(n)=O(1) \quad \text{当} n=1$$
$$T(n)=4T(n/2)+O(n) \quad \text{当} n>1$$

由此可得$T(n)=O(n^2)$。

采用分治法，把$X*Y$写成另一种形式：

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

虽然该式看起来比前式复杂些，但它仅需做3次$n/2$位整数的乘法（$A*C$、$B*D$和$(A-B)*(D-C)$），6次加、减法和2次移位。由此可以推出：

$$T(n) = O(n^{1.59})$$

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A和B，计算$C=A\times B$。

【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$O(n^3)$。

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$n\geq 2$时，将A、B分成4个$n/2\times n/2$的矩阵：

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

利用块矩阵的乘法，矩阵C可表示为

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

因此，原问题可以划分成计算8个子问题的乘积问题，因此，两个$n\times n$矩阵乘积的计算量是2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：

$T(n)=O(1)$	当$n=1$
$T(n)=8T(n/2)+O(n^2)$	当$n>1$

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。是否可以算得更快呢？

Strassen通过研究分析，提出了Strassen算法。要计算矩阵乘积：

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

只需要计算

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

其中：

$d_1=(A_{11}+A_{22})(B_{11}+B_{22})$

$d_2=(A_{21}+A_{22})B_{11}$

$d_3=A_{11}(B_{12}-B_{22})$

$d_4=A_{22}(B_{21}-B_{11})$

$d_5=(A_{11}+A_{12})B_{22}$

$d_6=(A_{21}-A_{11})(B_{11}+B_{12})$

$d_7=(A_{12}-A_{22})(B_{21}+B_{22})$

7个$n/2\times n/2$矩阵乘积
加减运算共需要$O(n^2)$

由上面可知，两个$n\times n$矩阵乘积的计算量是2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：

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

可以推导出$T(n)=O(n^{2.81})$，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，通常计算问题表现为以下特征：

（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。

1. PRAM模型

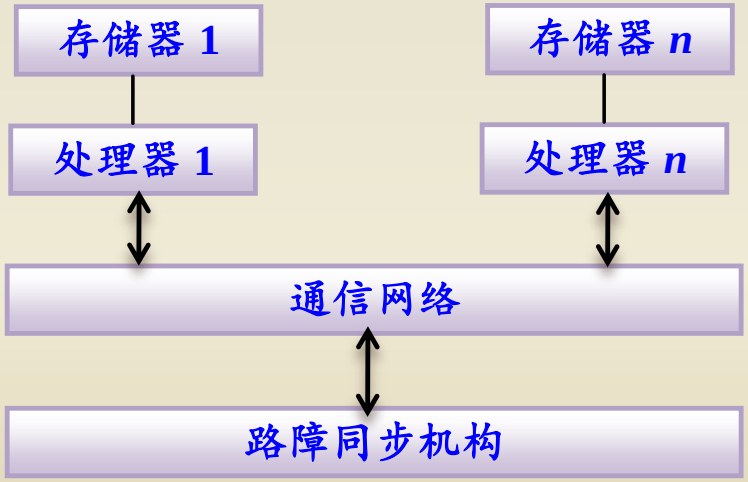
PRAM（Parallel Random Access Machine，随机存取并行机器）模型，也称为共享存储的SIMD（单指令流多数据流）模型，是一种抽象的并行计算模型，它是从串行的RAM模型直接发展起来的。

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。

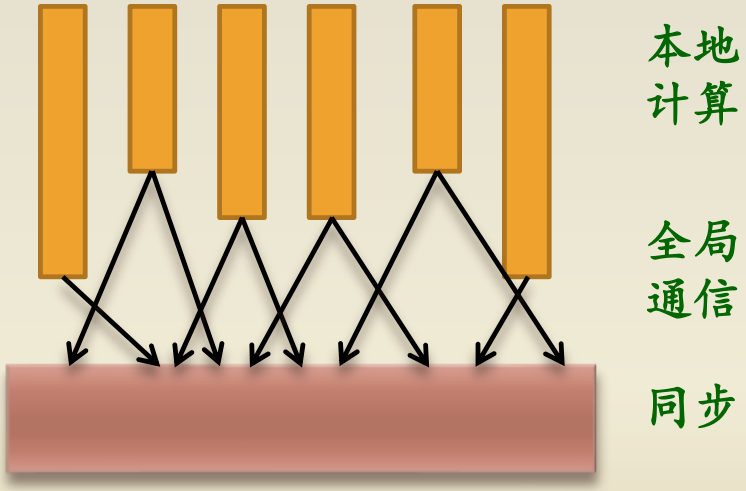
2．BSP模型

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

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



一个BSP程序有*n*个进程，每个驻留在一个节点上，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。



超步间采用路障同步，每个超步分成如下有序的三个部分：

① 计算：一个或多个处理器执行若干局部计算操作，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：处理器之间的相互交换数据，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必须全部完成之后，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。

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，对每次划分过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；而后进一步划分得到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序好的序列。当然这里举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a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 P_{id+2^{m-1}} ;
        ParaQuickSort(a, i, r-1, m-1, id);
        ParaQuickSort(a, r+1, j, m-1, id+2^{m-1});
        P_{id+2^{m-1}} 发送a[r+1, m-1]到Pid;
    }
}
```

在最好的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$O(n)$。

同串行快速排序算法一样，并行快速排序算法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$O(n)$。

阿里巴巴面试题

[单选题]

给定的一个长度为N的字符串str,查找长度为P(P<N)的字符串在str中的出现次数.下面的说法正确的是()

A. 不存在比最坏时间复杂度O(NP)好的算法

B. 不存在比最坏时间复杂度O(N^2)好的算法

C. 不存在比最坏时间复杂度O(P^2)好的算法

D. 存在最坏时间复杂度为O(N+P)的算法

E. 存在最坏时间复杂度为O(log(N+P))的算法

F. 以上都不对

str= “$a_1a_2...a_N$”
sstr= “$b_1b_2...b_P$ ”

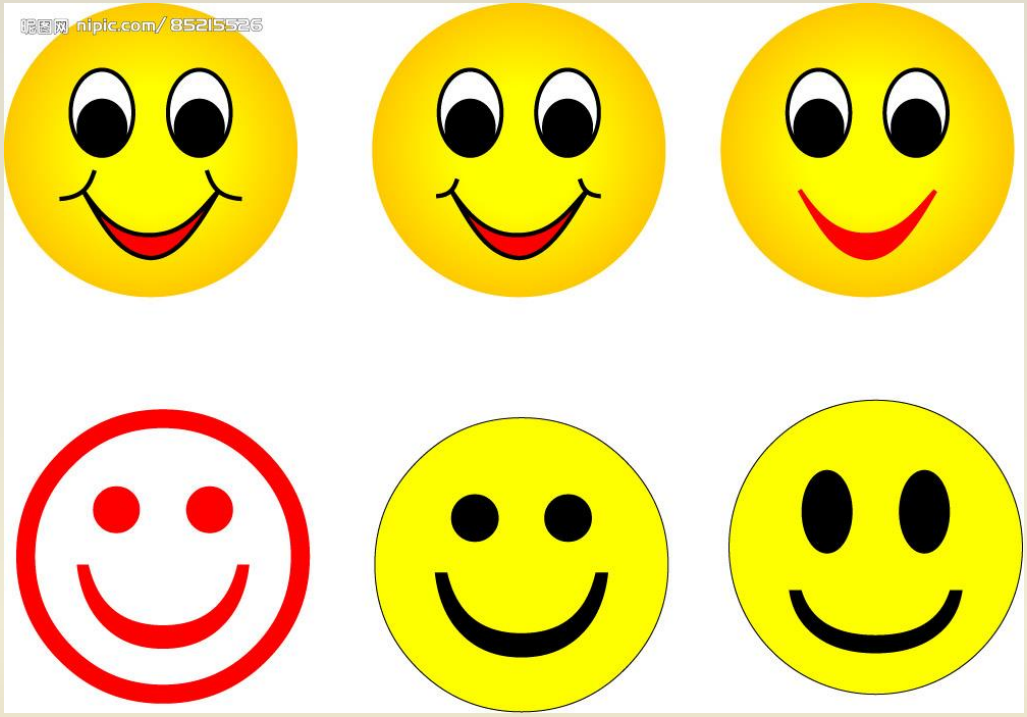
“$a_1a_2...a_P$” , ” $b_1b_2...b_P$ ”

“$a_2...a_{P+1}$” , ” $b_1b_2...b_P$ ”

…

“$a_Pa_2...a_{N-P+1}$” , ” $b_1b_2...b_P$ ”

并行任务


nipic.com/ 85215526