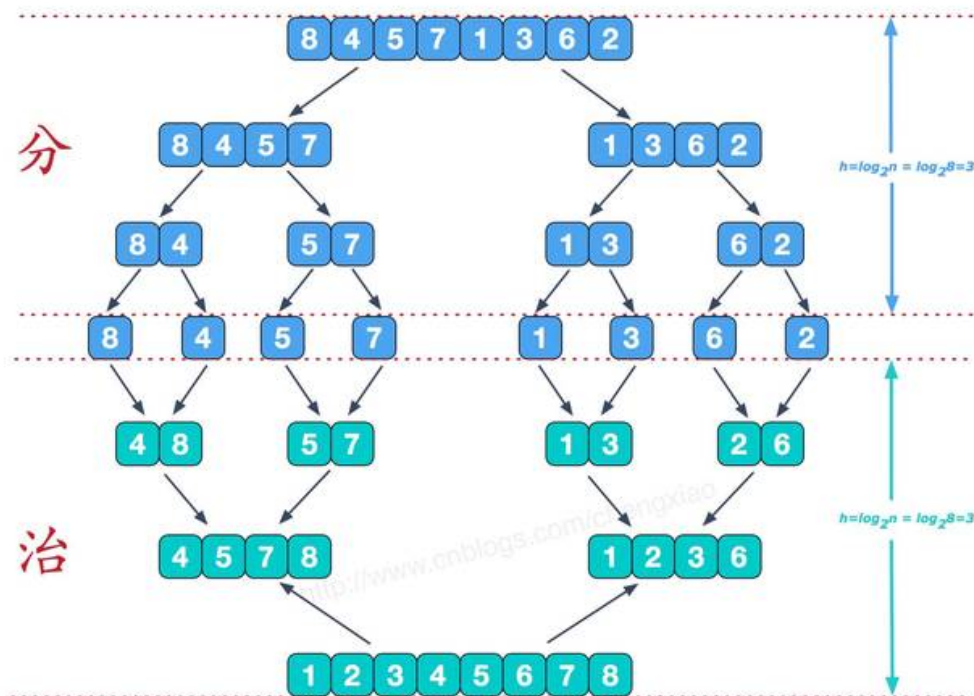


凡治众如治寡，分数是也
——《孙子兵法》

分治



分治法

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

3.1 分治法概述

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。**

3.1 分治法概述

分治法的特征

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 分治法概述

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。

3.1 分治法概述

分治法算法设计框架

divide-and-conquer(P)

{ if $|P| \leq n_0$ return adhoc(P);

/*分解为较小的子问题 P_1, P_2, \dots, P_k */

divide P into smaller substances P_1, P_2, \dots, P_k ;

for($i=1; i \leq k; i++$)

//循环处理k次

$y_i = \text{divide-and-conquer}(P_i)$;

//递归求解各子问题 P_i

return merge(y_1, y_2, \dots, y_k); //合并子问题解为原问题的解

}; */

3.1 分治法概述

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？

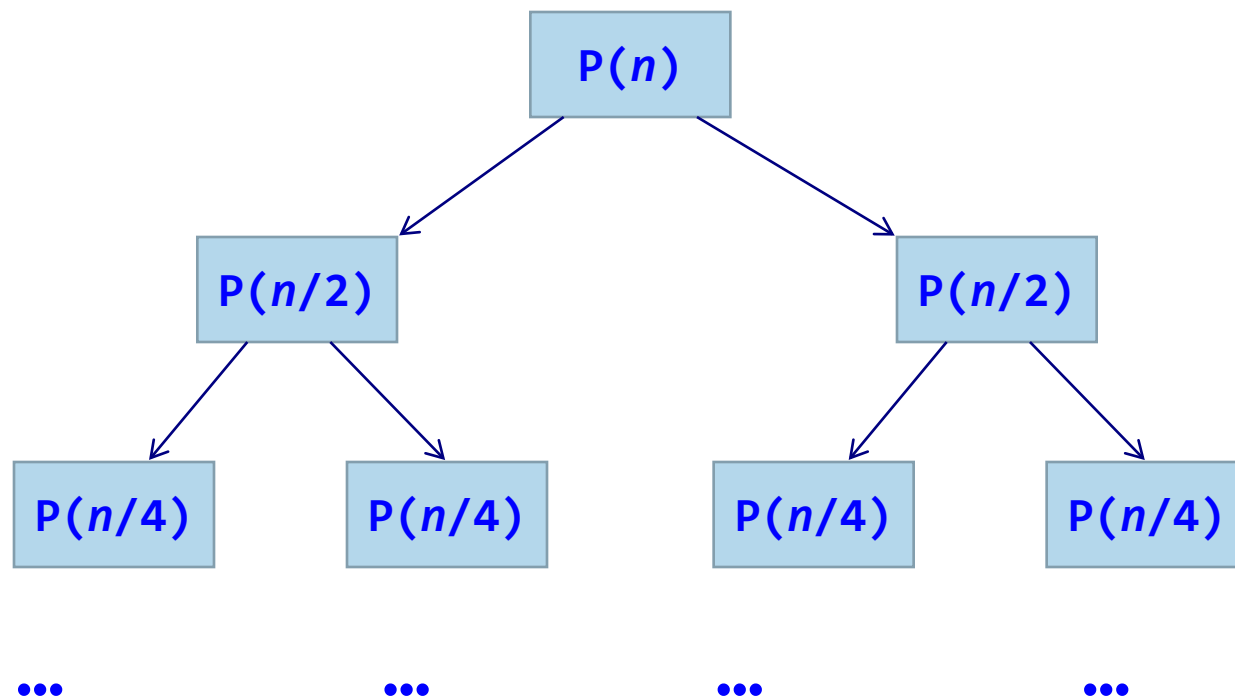
这些问题很难予以肯定的回答。但人们从大量实践中发现，在用分治法设计算法时，最好使子问题的规模大致相同。换句话说，将一个问题分成大小相等的 k 个子问题的处理方法是行之有效的。

当 $k=1$ 时称为减治法。如： $f(n)=n*f(n-1)$

3.1 分治法概述

分治法----二分法

许多问题可以取 $k=2$ ，称为**二分法**，如图所示，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 分治法求解排序问题

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。

3.2 分治法求解排序问题

快速排序

无序区



无序区1



无序区2



$f(a, s, t) \equiv$ 不做任何事情

$f(a, s, t) \equiv$ $i = \text{Partition}(a, s, t);$
 $f(a, s, i-1);$
 $f(a, i+1, t);$

当 $a[s..t]$ 中长度小于2
其他情况

3.2 分治法求解排序问题

快速排序

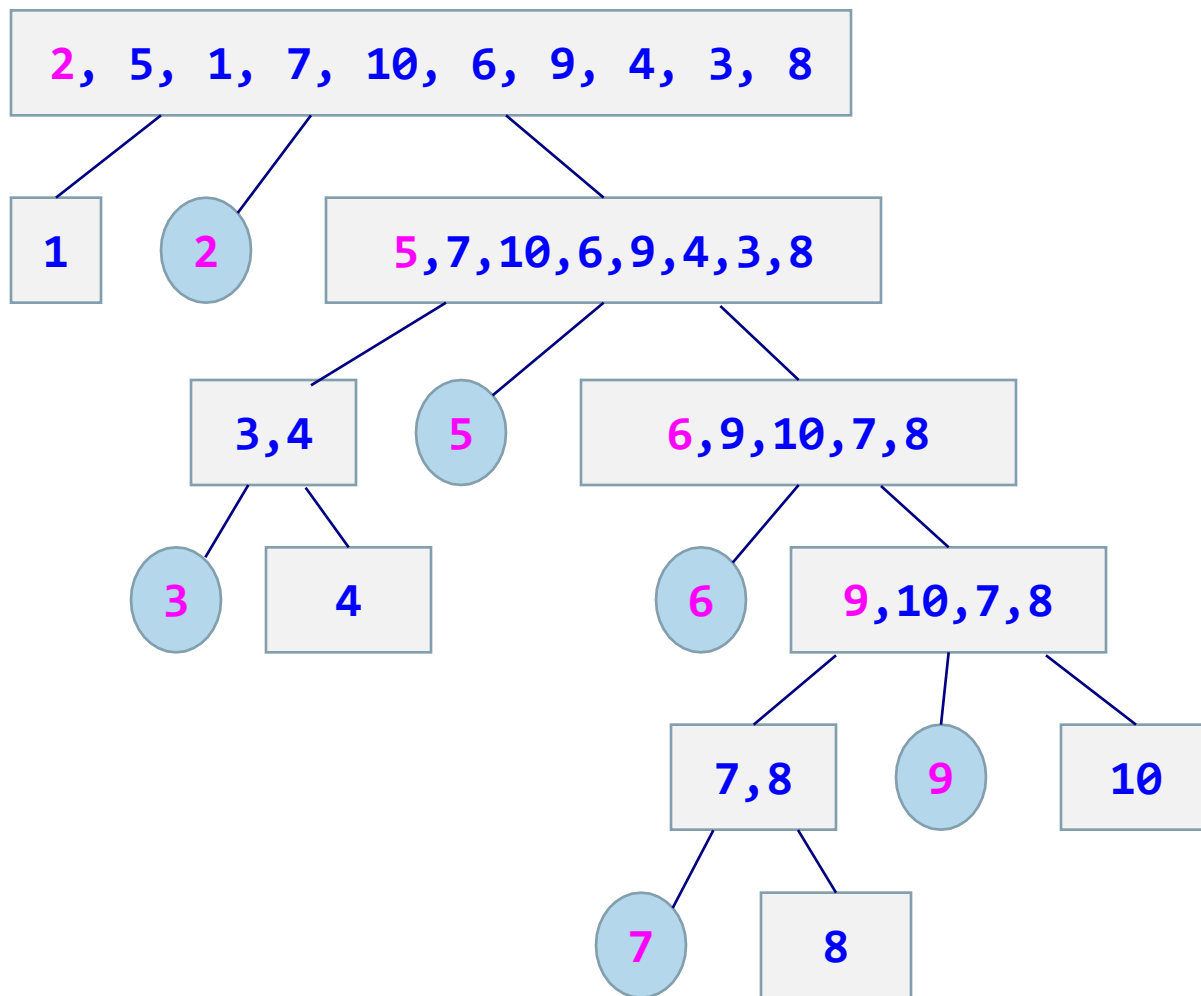
分治策略：

- 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，则它是有顺序的，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中 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。

3.2 分治法求解排序问题

快速排序

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（没有画出空的子序列）。



3.2 分治法求解排序问题

快速排序

快速排序——一次划分算法：

```
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;
}
```

3.2 分治法求解排序问题

快速排序

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
    }
}
```

3.2 分治法求解排序问题

快速排序

【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 $O(n \log_2 n)$ 。

练习

已知由 n ($n \geq 2$) 个正整数构成的集合 $A = \{a_k\}$ ($0 \leq k < n$)，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。要求：

- (1) 给出算法的基本设计思想。
- (2) 根据设计思想，采用C、C++描述算法，关键之处给出注释。
- (3) 说明你所设计算法的时间复杂度和空间复杂度。

练习

 **思路：** 将A递增排序，前 $\lfloor n/2 \rfloor$ 个元素放在 A_1 中，其他放在 A_2 中



将最小的 $\lfloor n/2 \rfloor$ 个元素放在 A_1 中，其他放在 A_2 中



查找第 $n/2$ 小的元素

递归快速排序

```
int Partition(int a[], int low, int high) //以a[low]为基准划分
{
    int i=low, j=high;
    int povit=a[low];

    while (i<j)
    {
        while (i<j && a[j]>=povit)
            j--;
        a[i]=a[j];
        while (i<j && a[i]<=povit)
            i++;
        a[j]=a[i];
    }

    a[i]=povit;
    return i;
}
```

```
int Solution(int a[], int n)                                //求解
{  int low=0, high=n-1;
   bool flag=true;
   while (flag)
   {  int i=Partition(a, low, high);
      if (i==n/2-1)                                         //基准a[i]为第n/2的元素
         flag=false;
      else if (n/2-1>i)                                     //在右区间查找
         low=i+1;
      else
         high=i-1;                                         //在左区间查找
   }
}
```

```
int s1=0, s2=0;
for (int i=0; i<n/2; i++)
    s1+=a[i];
for (int j=n/2; j<n; j++)
    s2+=a[j];
return s2-s1;
}
```

3.2 分治法求解排序问题

归并排序

【归并排序的基本思想】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。

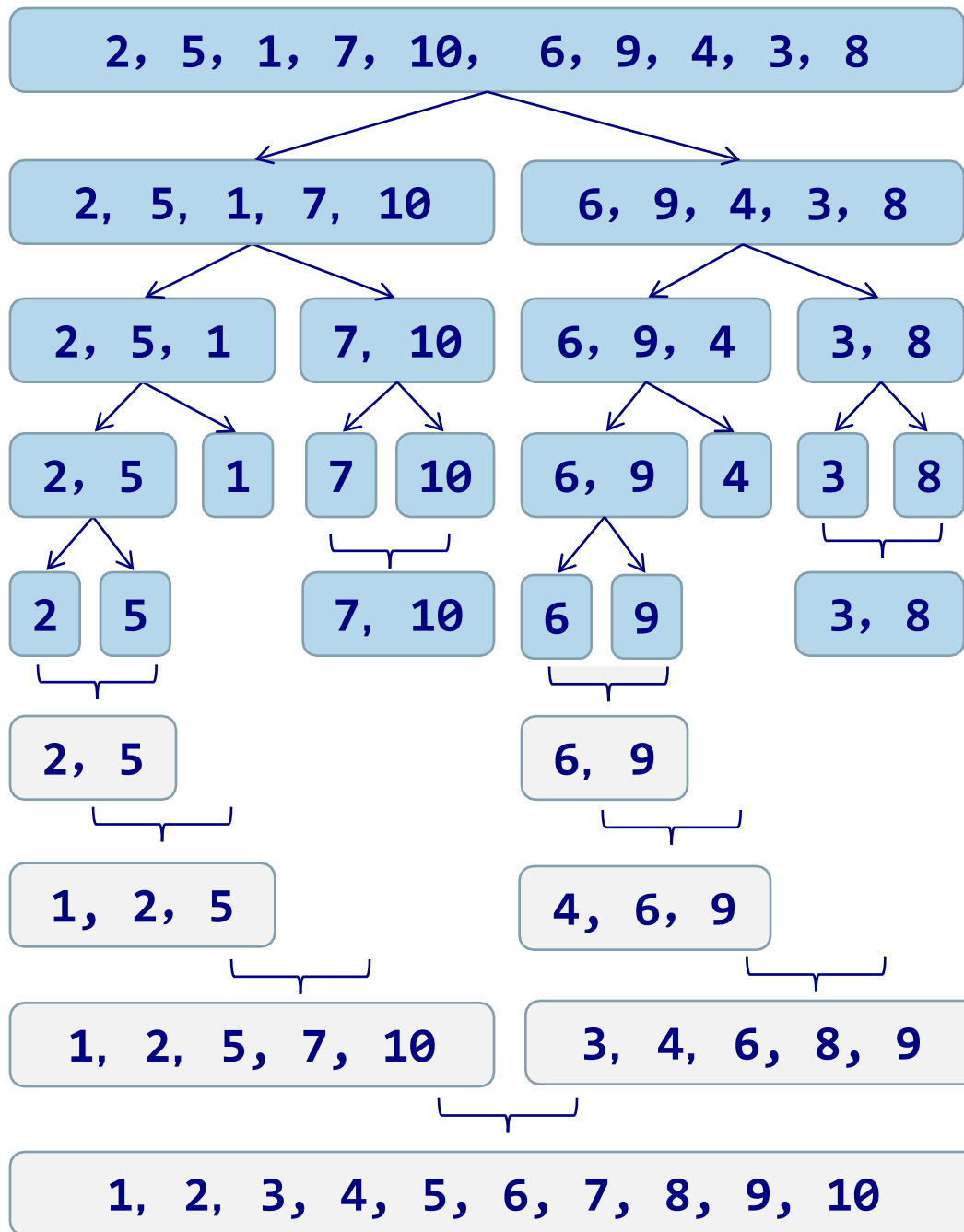
- 自顶向下（递归）
- 自底向上(非递归)

示例

归并排序-
自顶向下

分解
合并

顶
下



3.2 分治法求解排序问题

归并排序

自顶向下的二路归并排序算法：设归并排序的当前区间是 $a[\text{low}..\text{high}]$ ，则递归归并排序的两个步骤如下：

- ① **分解**：将序列 $a[\text{low}..\text{high}]$ 一分为二，即求 $\text{mid} = (\text{low} + \text{high}) / 2$ ；递归地对两个子序列 $a[\text{low}..\text{mid}]$ 和 $a[\text{mid}+1..\text{high}]$ 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 **求解子问题**：排序两个子序列 $a[\text{low}..\text{mid}]$ 和 $a[\text{mid}+1..\text{high}]$ 。
- ③ **合并**：与分解过程相反，将已排序的两个子序列 $a[\text{low}..\text{mid}]$ 和 $a[\text{mid}+1..\text{high}]$ 归并为一个有序序列 $a[\text{low}..\text{high}]$ 。

```
void merge(int a[], int low, int mid, int high)
//a[low..mid], a[mid+1..high]分别有序，将其归并为新有序序列
{
    i=low, j=mid+1, k=1;
    tmpa=(int *)malloc((high-low+1)*sizeof(int)+1);
    while (i<=mid && j<=high)
        if (a[i]<=a[j]) { tmpa[k]=a[i]; i++; k++; }
        else { tmpa[k]=a[j]; j++; k++; }
    while (i<=mid) { tmpa[k]=a[i]; i++; k++; }
    while (j<=high) { tmpa[k]=a[j]; j++; k++; }
    for (k=1, i=low; i<=high; k++, i++) a[i]=tmpa[k];
    free(tmpa);
}
```

3.2 分治法求解排序问题

归并排序

```
void mergeSort(int a[], int low, int high) {  
    if (low < high)                                //子序列有两个或以上元素  
    {  
        mid = (low + high) / 2;  
        mergeSort(a, low, mid);  
        mergeSort(a, mid + 1, high);  
        merge(a, low, mid, high); //O(n)  
    }  
}
```

$$T(n) = 1$$

当 $n = 1$

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

当 $n > 1$

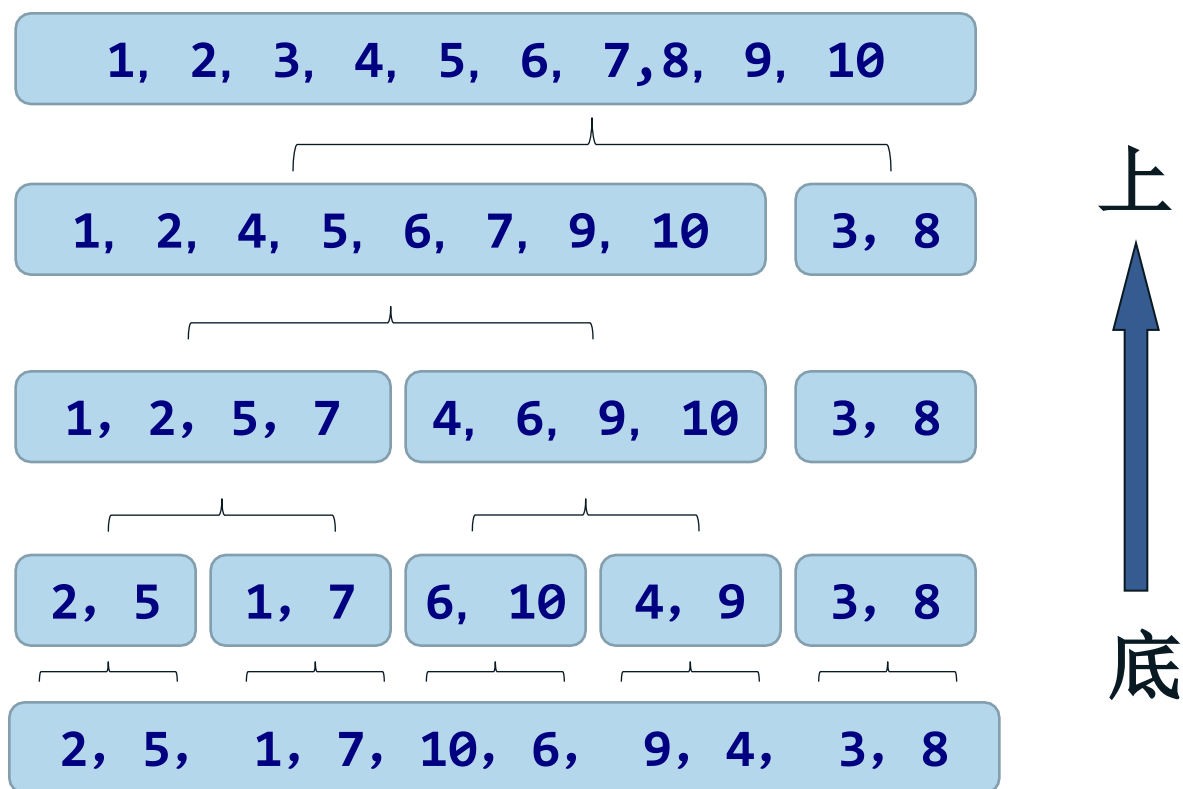
$$T(n) = O(n \log_2 n)$$

$$S(n) = O(n)$$

3.2 分治法求解排序问题

归并排序

归并排序-自底向上



3.2 分治法求解排序问题

归并排序

自底向上的二路归并排序的分治策略如下：

循环 $\lceil \log_2 n \rceil$ 次， len 依次取1、2、4、8...、 $\log_2 n$ 。每次执行以下步骤：

- ① **分解**：将原序列分解成长度为 len 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中 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。

3.2 分治法求解排序问题

归并排序 → 自底向上

```
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
}
```

3.2 分治法求解排序问题

归并排序

将相邻的长度为length的有序表合并成一个有序表

```
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
```

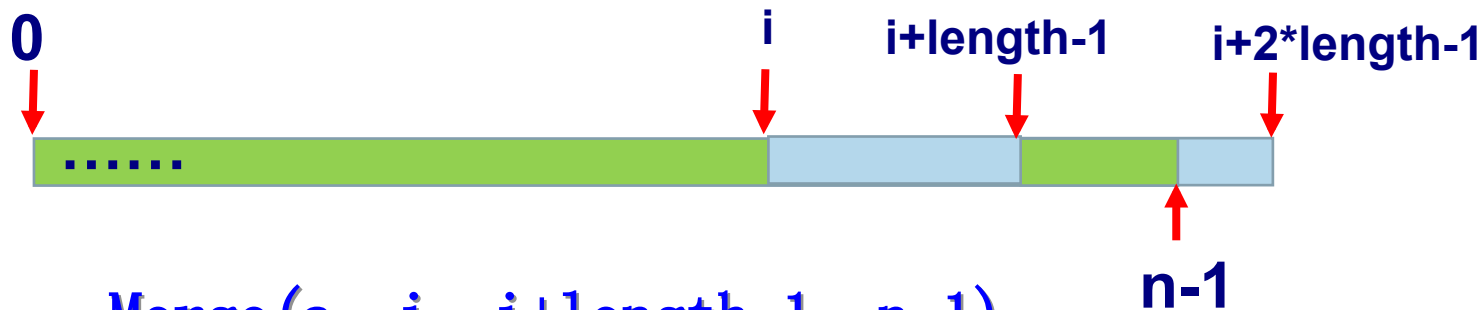
```
        Merge(a, i, i+length-1, n-1); /*归并这两个子表;
```

```
        否则余下一个子表，此次不参与归并*/
```

```
}
```



Merge(a , i , $i + \text{length} - 1$, $i + 2 * \text{length} - 1$)



Merge(a , i , $i + \text{length} - 1$, $n - 1$)



此时，剩下一个子表，
则不做合并操作。

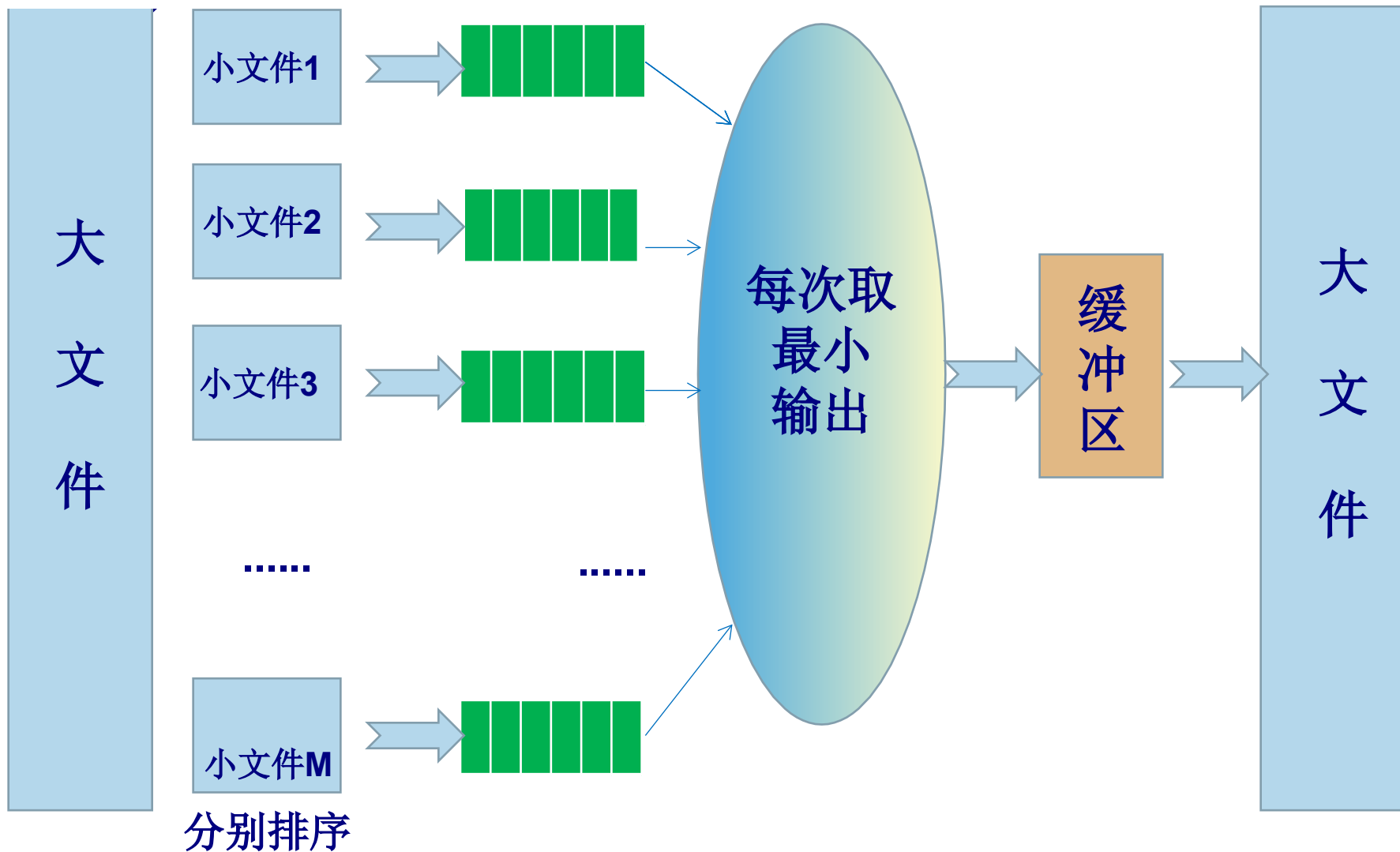
3.2 分治法求解排序问题

归并排序

```
void MergeSort(int a[], int n)           //自底向上的二路归并算法
{
    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 $O(n \log_2 n)$ 。

应用于外部排序



3.3 分治法求解查找问题

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。

3.3 分治法求解查找问题

查找最大和次大元素

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

- (1) $a[\text{low}..\text{high}]$ 中只有一个元素：则 $\text{max1}=a[\text{low}]$ ， $\text{max2}=-\text{INF}$ ($-\infty$) (要求它们是不同的元素)。
- (2) $a[\text{low}..\text{high}]$ 中只有两个元素：则 $\text{max1}=\text{MAX}\{a[\text{low}], a[\text{high}]\}$ ， $\text{max2}=\text{MIN}\{a[\text{low}], a[\text{high}]\}$ 。
- (3) $a[\text{low}..\text{high}]$ 中有两个以上元素：按中间位置 $\text{mid}=(\text{low}+\text{high})/2$ 划分为 $a[\text{low}..\text{mid}]$ 和 $a[\text{mid}+1..\text{high}]$ 左右两个区间（注意左区间包含 $a[\text{mid}]$ 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 $\text{lmax1} > \text{rmax1}$ ，则 $\text{max1}=\text{lmax1}$ ， $\text{max2}=\text{MAX}\{\text{lmax2}, \text{rmax1}\}$ ；否则 $\text{max1}=\text{rmax1}$ ， $\text{max2}=\text{MAX}\{\text{lmax1}, \text{rmax2}\}$ 。

3.3 分治法求解查找问题

查找最大和次大元素

【算法描述】

```
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lmax1和lmax2
        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
        }
    }
}
```

3.3 分治法求解查找问题

查找最大和次大元素

【算法分析】 对于 $\text{solve}(a, 0, n-1, \text{max1}, \text{max2})$ 调用, 其比较次数的递推式为:

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

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

可以推导出 $T(n)=O(n)$ 。

3.3 分治法求解查找问题

查找最大和最小元素

```
void solve(int a[], int low, int high, int *max, int *min)
{
    if (low==high)                //区间只有一个元素
    {
        *max=a[low];    *min=a[low];
    }
    else if (low==high-1)         //区间只有两个元素
    {
        *max=max(a[low], a[high]); *min=min(a[low], a[high]);
    }
    else                          //区间有两个以上元素
    {
        mid=(low+high)/2;
        int lmax, lmin; int rmax, rmin;
        solve(a, low, mid, &lmax, &lmin);           //左区间求lmax和lmin
        solve(a, mid+1, high, &rmax, &rmin);        //右区间求rmax和rmin
        if (lmax>rmax) *max=lmax;
        else *max=rmax;
        if (lmin<rmin) *min=lmin;
        else *min=rmin;
    }
}
```

3.3 分治法求解查找问题

查找最大和最小元素

算法分析：

(1) 列出比较次数的递归方程。

$$T(n) = \begin{cases} 0, & n = 1 \\ 1, & n = 2 \\ 2T(\frac{n}{2}) + 2, & n > 2 \end{cases}$$

3.3 分治法求解查找问题

查找最大和最小元素

(2). 求解递归方程。

当 $n > 2$ 时

$$T(n) = 2T\left(\frac{n}{2}\right) + 2 = 2[2T\left(\frac{n}{2^2}\right) + 2] + 2 = 2^{k-1}T\left(\frac{n}{2^{k-1}}\right) + 2^k - 2$$

令 $n=2^k$, 则

$$2^{k-1} + 2^{k-2} + \dots + 2 = \frac{2 - 2^{k-1} * 2}{1 - 2}$$

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

因为 $T(2) = 1$

所以

$$T(n) = \frac{n}{2}T(2) + n - 2 = \frac{3}{2}n - 2$$

3.3 分治法求解查找问题

折半查找

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

(1) 若 $k = a[\text{mid}]$ ，则查找成功并返回该元素的物理下标；

(2) 若 $k < a[\text{mid}]$ ，则由表的有序性可知 $a[\text{mid}..\text{high}]$ 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 $a[\text{low}..\text{mid}-1]$ 中，故新的查找区间是左子表 $a[\text{low}..\text{mid}-1]$ ；

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

下一次查找是针对新的查找区间进行的。

3.3 分治法求解查找问题

折半查找

【算法实现】

```
int BinSearch(int a[], int low, int high, int k)
//拆半查找算法
{   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
}
```


3.3 分治法求解查找问题

折半查找

【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：

$$C(n)=1 \quad \text{当 } n=1$$

$$C(n) \leq 1 + C(\lfloor n/2 \rfloor) \quad \text{当 } n \geq 2$$

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 分治法求解查找问题

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 ($1 \leq k \leq n$) 小的元素。

【问题求解】 假设无序序列存放在 $a[1..n]$ 中，若将 a 递增排序，则第 k 小的元素为 $a[k]$ 。

采用类似于快速排序的思想。

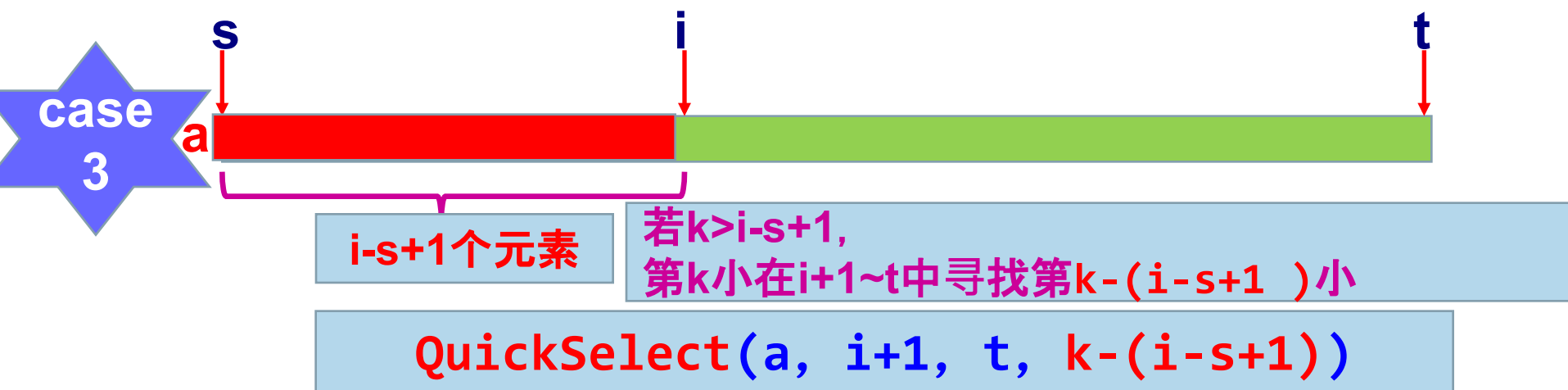
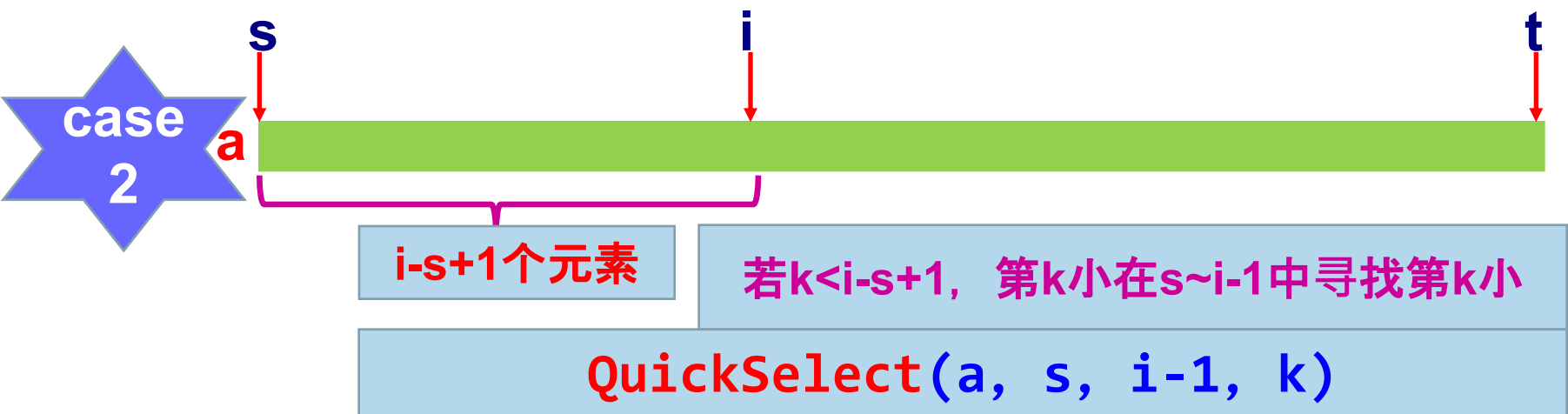
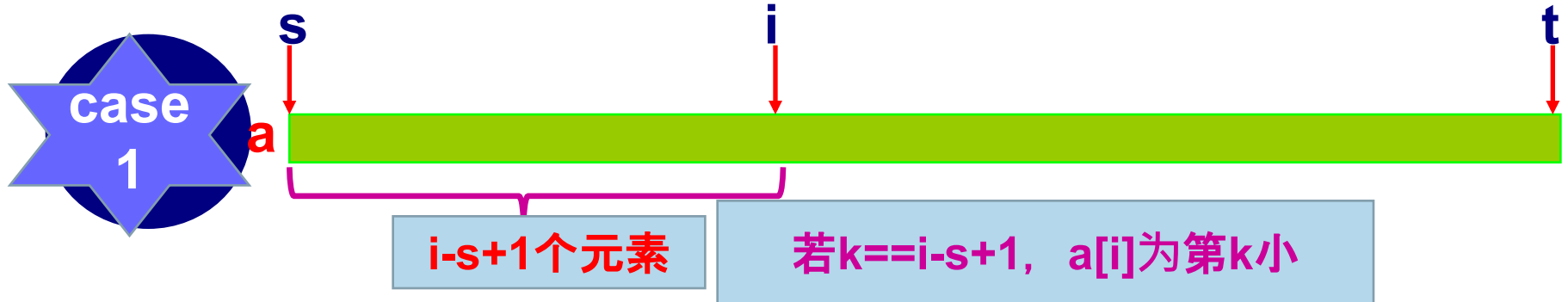
3.3 分治法求解查找问题

寻找一个序列中第 k 小元素

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(即最小下标位置的元素为数轴)进行一次划分，对应的划分位置下标为 i 。3种情况：

- 若 $k=i-s+1$ ， $a[i]$ 即为所求，返回 $a[i]$ 。
- 若 $k < i-s+1$ ，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 $k > i-s+1$ ，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。



3.3 分治法求解查找问题

寻找一个序列中第 k 小元素

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;

        if (k==i-s+1) return a[i];
        else if (k<i-s+1) return QuickSelect(a, s, i-1, k);
        //在左区间中递归查找
        else return QuickSelect(a, i+1, t, k-(i-s+1));
        //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];
}
```

3.3 分治法求解查找问题

寻找一个序列中第 k 小元素

【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 $O(n^2)$ 。

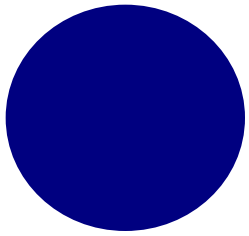
在平均情况下该算法的时间复杂度为 $O(n)$ 。

3.3 分治法求解查找问题

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。



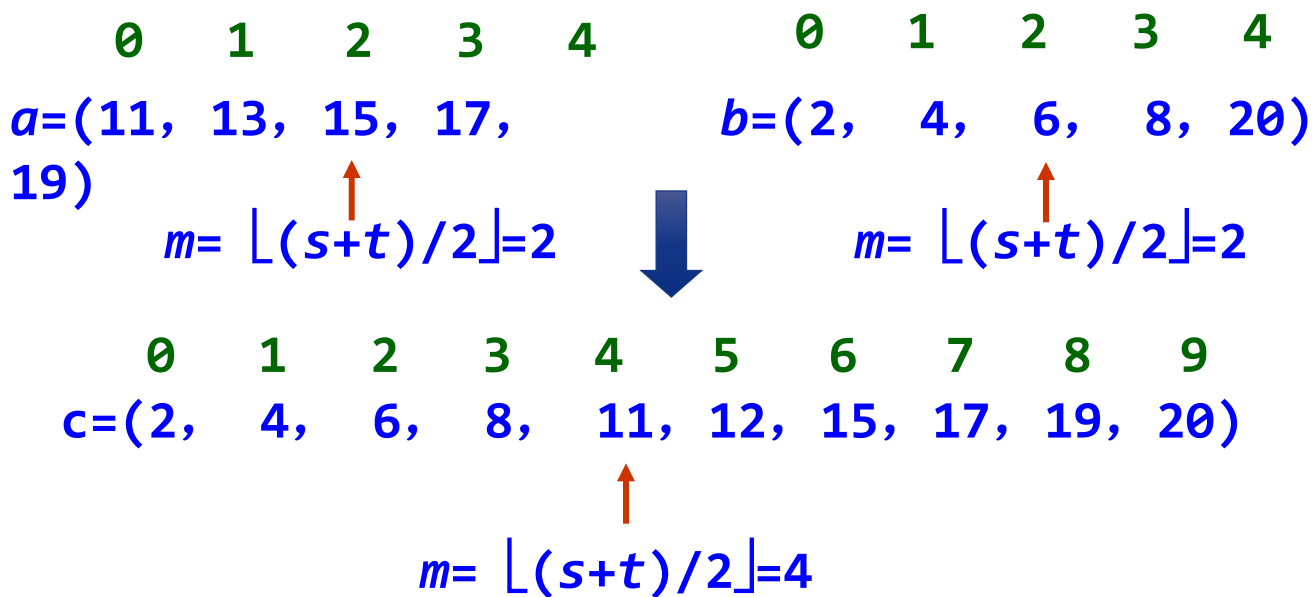
例如，若序列 $a=(11, 13, 15, 17, 19)$ ，其中位数是15，若 $b=(2, 4, 6, 8, 20)$ ，其中位数为6。两个等长有序序列的中位数是含它们所有元素的有序序列的中位数，例如 a 、 b 两个有序序列的中位数为11。

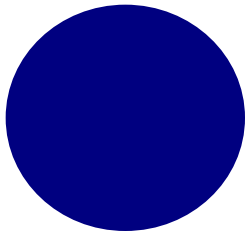
$a=(11, 13, 15, 17, 19)$ $b=(2, 4, 6, 8, 20)$



$c=(2, 4, 6, 8, 11, 12, 15, 17, 19, 20)$

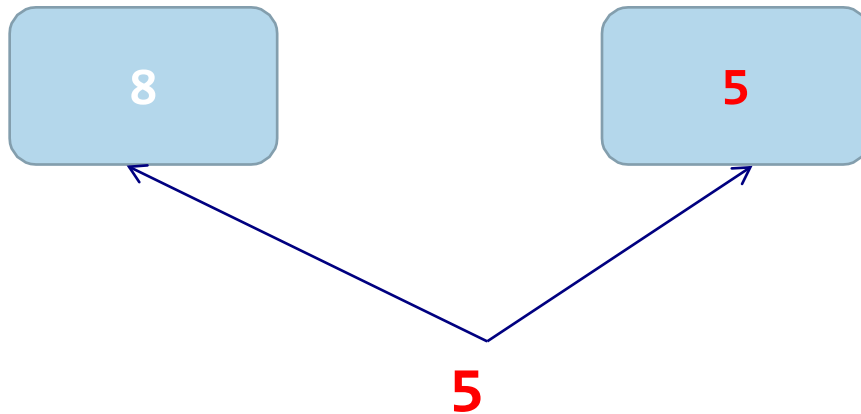
【问题求解】 对于含有 n 个元素的有序序列 $a[s..t]$ ，当 n 为奇数时，中位数是出现在 $m = \lfloor (s+t)/2 \rfloor$ 处；当 n 为偶数时，中位数下标有 $m = \lfloor (s+t)/2 \rfloor$ （下中位）和 $m = \lfloor (s+t)/2 \rfloor + 1$ （上中位）两个。为了简单，仅考虑中位数为 $m = \lfloor (s+t)/2 \rfloor$ 处。

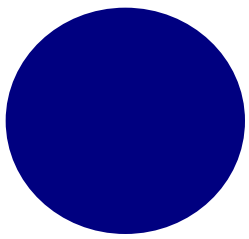




采用二分法求含有 n 个有序元素的序列 a 、 b 的中位数的过程如下：

- 若 $n=1$ ，较小者为中位数。

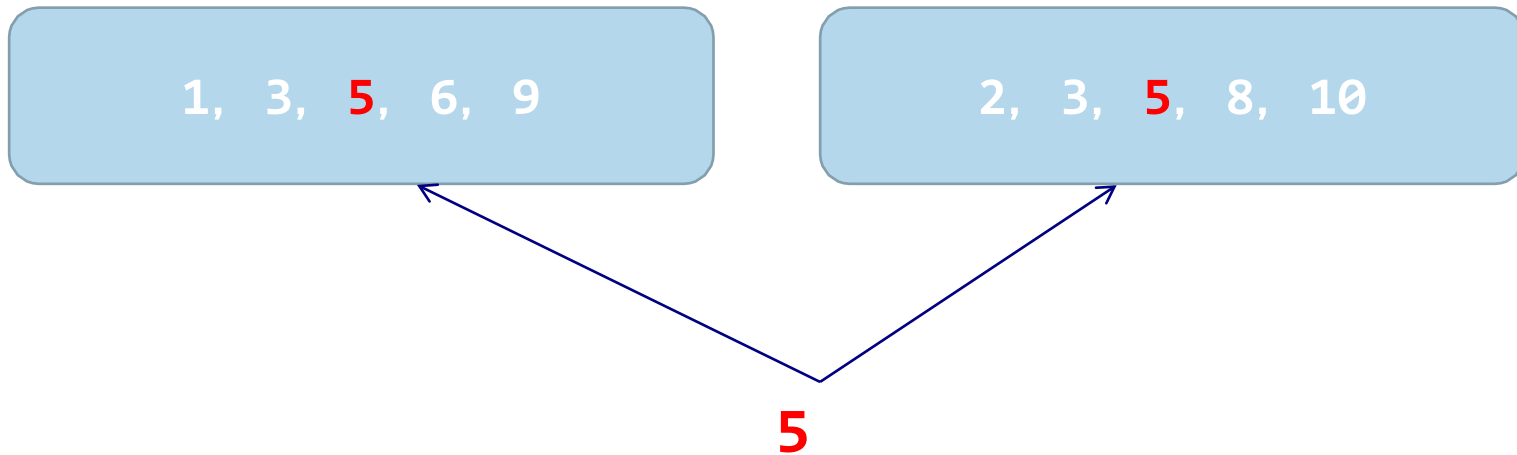




其他(else): 又分为3种情况:

分别求出 a 、 b 的中位数 $a[m1]$ 和 $b[m2]$:

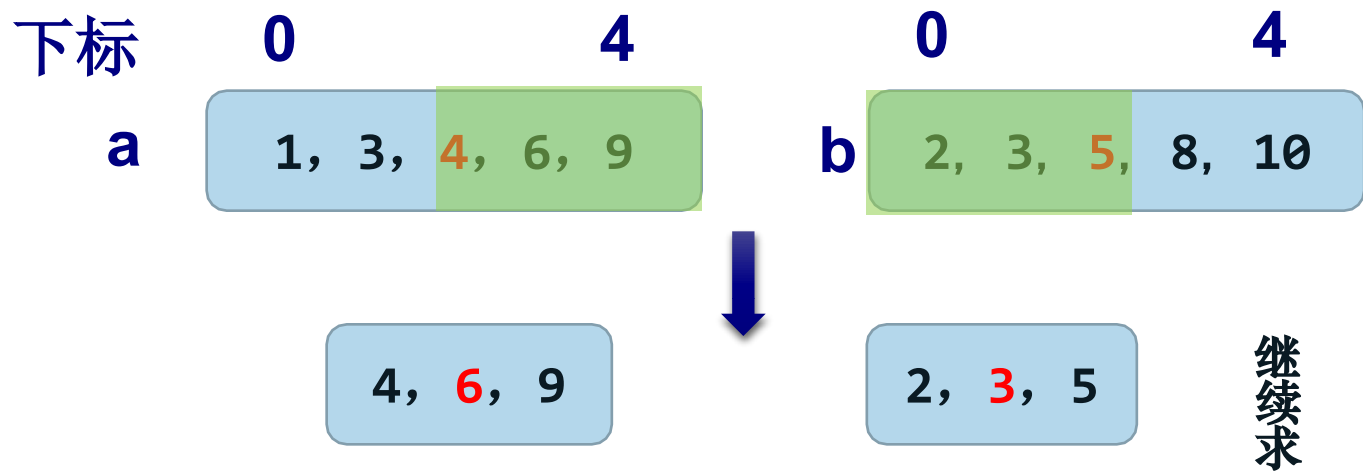
① 若 $a[m1] = b[m2]$, 则 $a[m1]$ 或 $b[m2]$ 即为所求中位数, 算法结束。



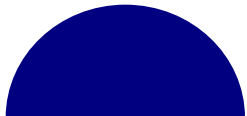


② 若 $a[m1] < b[m2]$ ，则舍弃序列a中前半部分（较小的一半），同时舍弃序列b中后半部分（较大的一半）**要求舍弃的长度相等（因为要转换为同类型的子问题（同为n个元素的有序表））。**

case1:（序列元素个数n为奇数）下面示例中， $n=5$

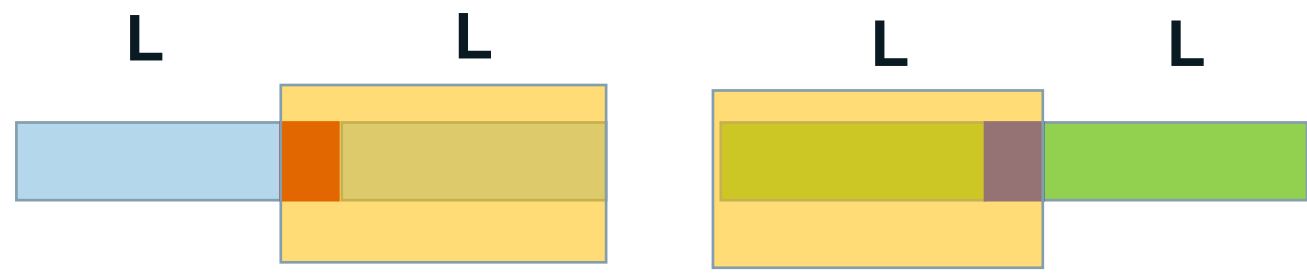


.....



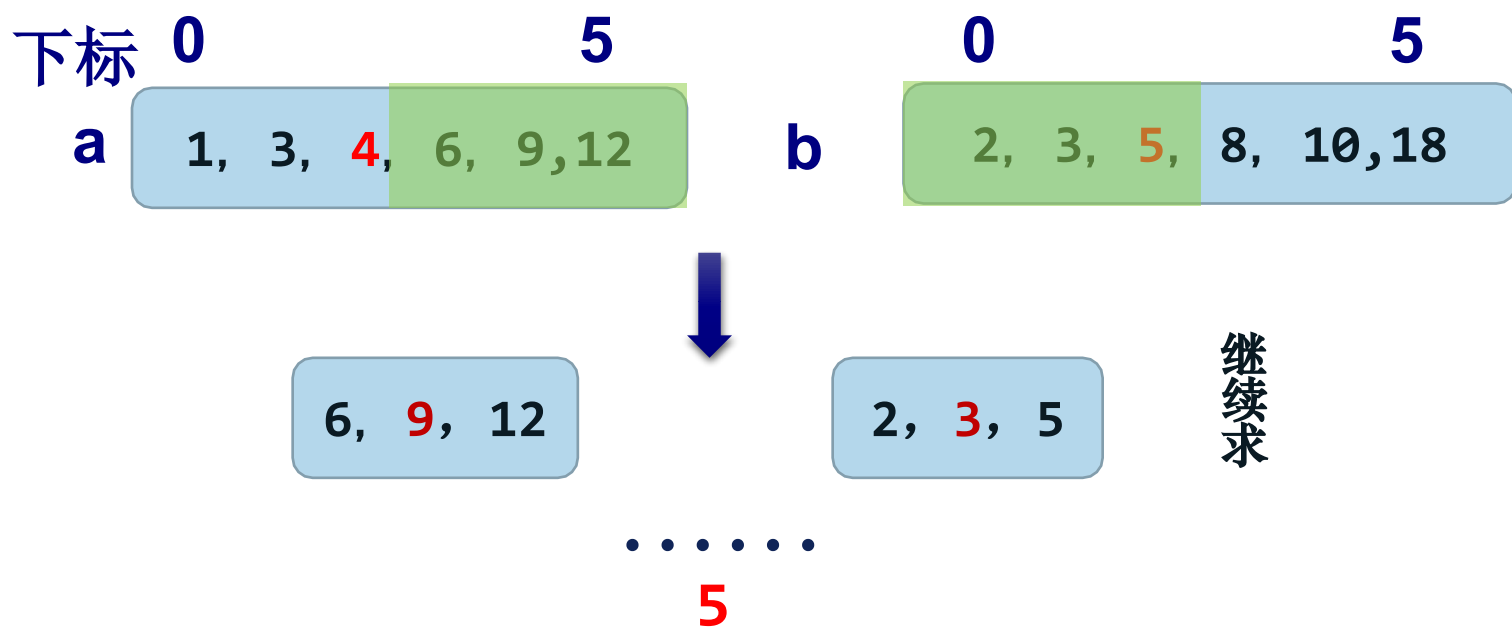
② 若 $a[m1] < b[m2]$ ，则舍弃序列a中前半部分（较小的一半），同时舍弃序列b中后半部分（较大的一半）要求舍弃的长度相等（因为要转换为同类型的子问题（同为 n 个元素的有序表））。

case1:（序列元素个数 n 为奇数）



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（因为要转换为同类型的子问题：即：同为n个元素的有序表）

case2:（序列元素个数n为偶数）下面示例中， $n=6$ 。n为偶数时，我们取中用的是下取整。这样在舍弃子表的时候，a的左子表元素个数少于右子表，故a的左子表要连其中间位置的元素一并舍弃。对于b，它需要舍弃右子表，故不需要连带舍去其中间位置的元素。这样就能保证子问题与原问题相同，仍然为两个长度相等的有序表。



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（因为要转换为同类型的子问题：即：同为 n 个元素的有序表）

case2:（序列元素个数 n 为偶数）下面示例中， $n=6$ 。 n 为偶数时，我们取中用的是下取整。这样在舍弃子表的时候，a的左子表元素个数少于右子表，故a的左子表要连其中间位置的元素一并舍弃。对于b，它需要舍弃右子表，故不需要连带舍去其中间位置的元素。这样就能保证子问题与原问题相同，仍然为两个长度相等的有序表。



对于 $a[s1..t1]$ 和 $b[s2..t2]$;取中间分别为 $a[m1]$ 和 $b[m2]$

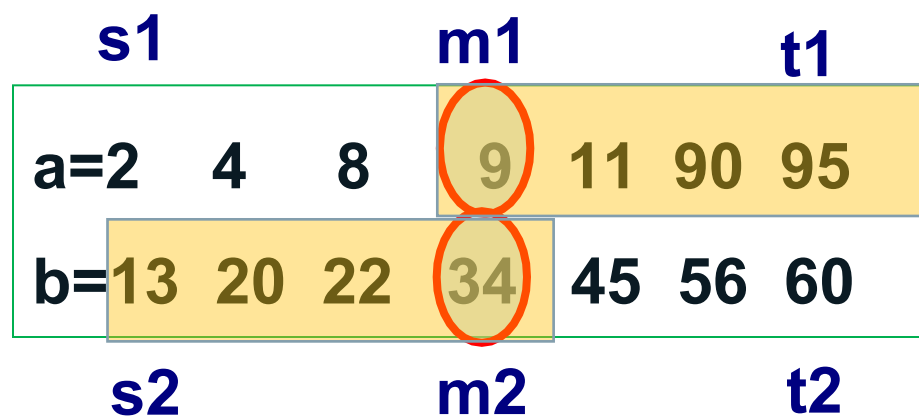
②若 $a[m1] < b[m2]$ 规律:

取 a 后半部分: 奇数个元素, $a[m1...t1]$

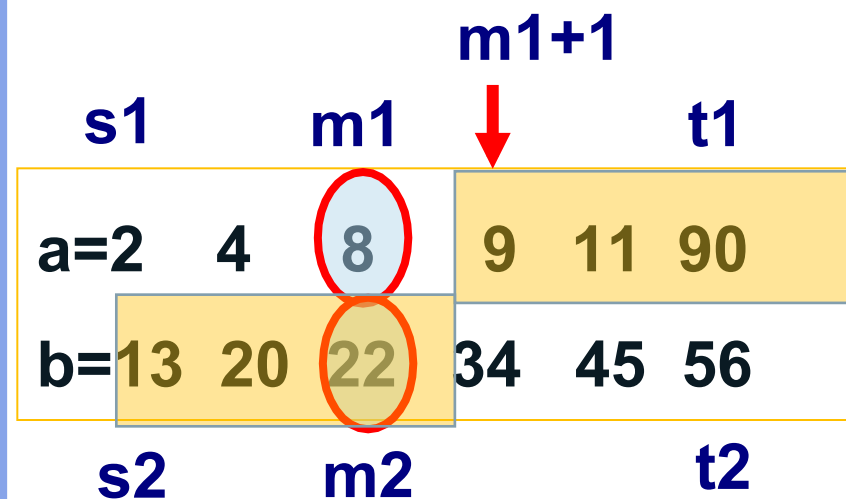
偶数个元素, $a[m1+1..t1]$

取 b 前半部分: $b[s2..m2]$

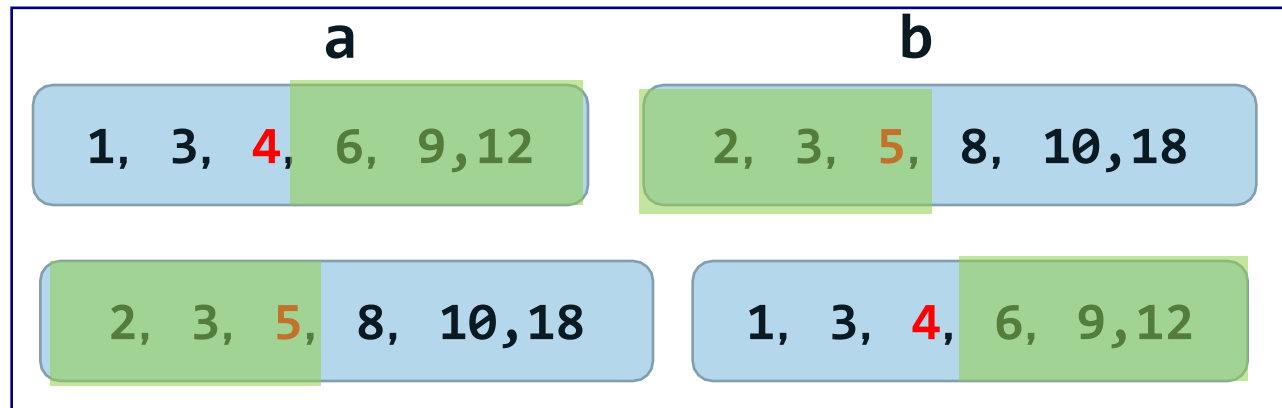
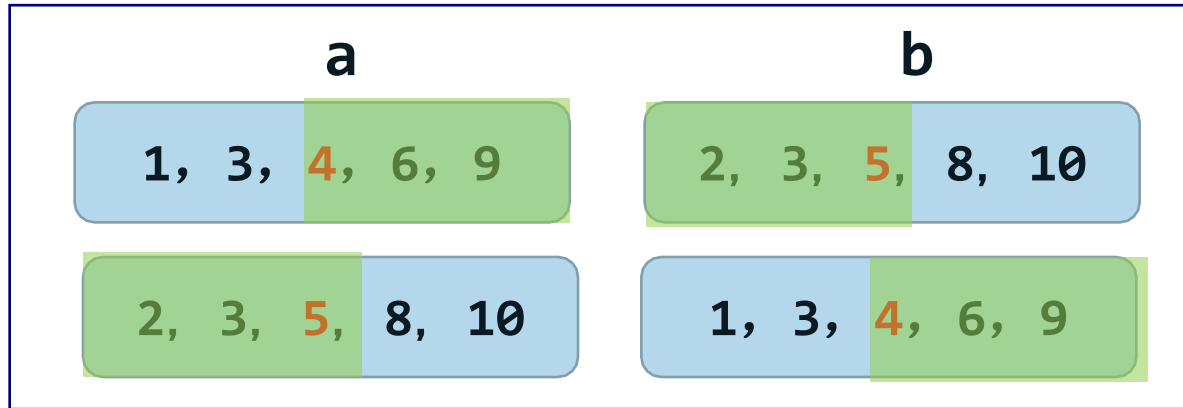
n为奇数



n为偶数



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。舍弃一半即 $\lfloor n/2 \rfloor$ 个元素。将情况②中的调换位置即可。



对于 $a[s1..t1]$ 和 $b[s2..t2]$;取中间分别为 $a[m1]$ 和 $b[m2]$

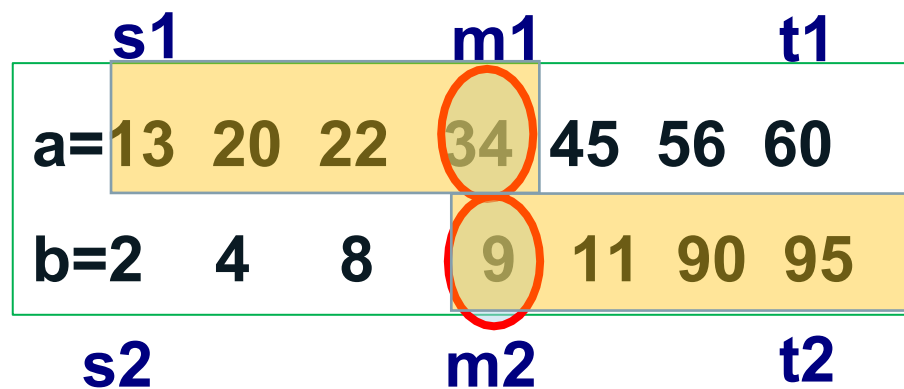
③若 $a[m1] > b[m2]$ 规律:

取a前半部分: $a[s1..m1]$

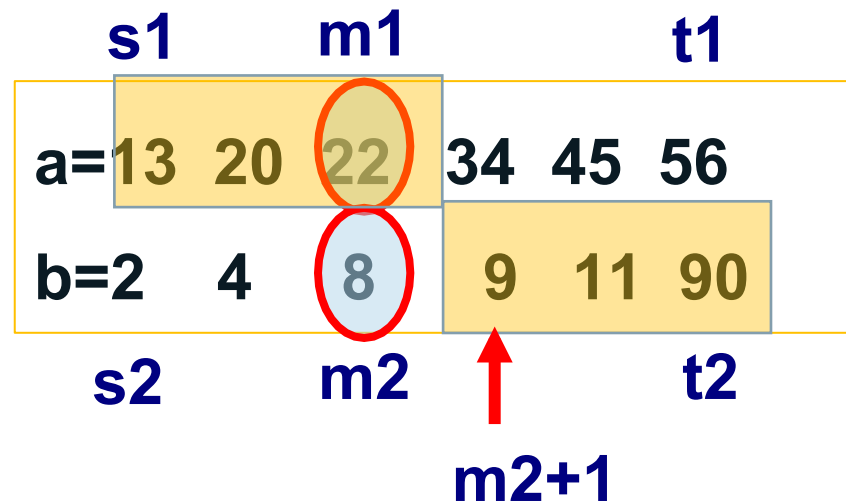
取b后半部分: 奇数个元素, 后半部分为 $b[m2..t2]$

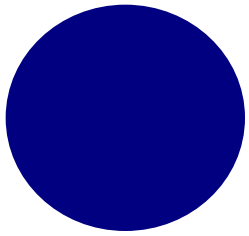
偶数个元素, 后半部分为 $b[m2+1..t2]$

n为奇数



n为偶数





13 20 22 **34** 45 56 60

2 4 8 **9** 11 90 95

13 **20** 22 34

9 **11** 90 95

13 20

90 95

20

90

20

2 4 8 9 11 13 **20** 22 34 45 56 60 90 95

```

int midNum(int a[],int s1,int e1,int b[],int s2,int e2)
{
    //求两个有序序列a[s1..e1]和b[s2..e2]的中位数
    int m1,m2;
    if (s1==e1 && s2==e2)          //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+e1)/2;                //求a的中位数位置
        m2=(s2+e2)/2;                //求b的中位数位置
        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
        {
            if((s1+e1)%2==0) s1=m1; //a取后半部分，奇数个元素
            else s1=m1+1;           //偶数个元素

            e2=m2;                   //b取前半部分
            return midNum(a,s1,e1,b,s2,e2);
        }
        else                          //当a[m1]>b[m2]时
        {
            e1=m1;                   //a取前半部分
            if((s2+e2)%2==0) s2=m2; //b取后半部分，奇数个元素
            else s2=m2+1;           //偶数个元素

            return midNum(a,s1,e1,b,s2,e2);
        }
    }
}

```

3.3 分治法求解组合问题

求解最大连续子序列和问题

【问题描述】 给定一个有 n ($n \geq 1$) 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（长度为0的子序列），如果小于0，其结果为0。

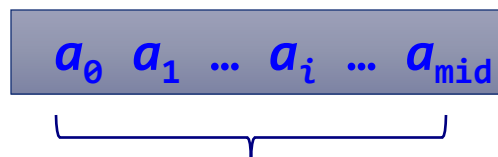
3.3 分治法求解组合问题

求解最大连续子序列和问题

【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 $\text{mid}=\lfloor (n-1)/2 \rfloor$ ，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 $a[0..\text{mid}]$ 中。采用递归求出其最大连续子序列和 maxLeftSum 。



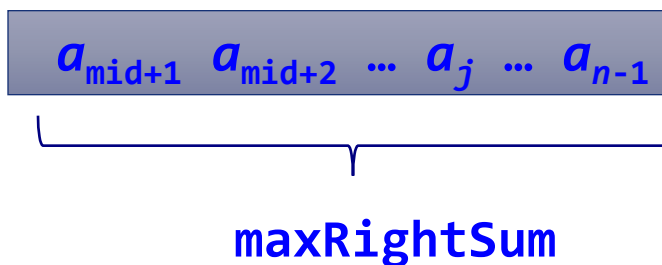
maxLeftSum

3.3 分治法求解组合问题

求解最大连续子序列和问题

【问题求解】

(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 。



求解最大连续子序列和问题

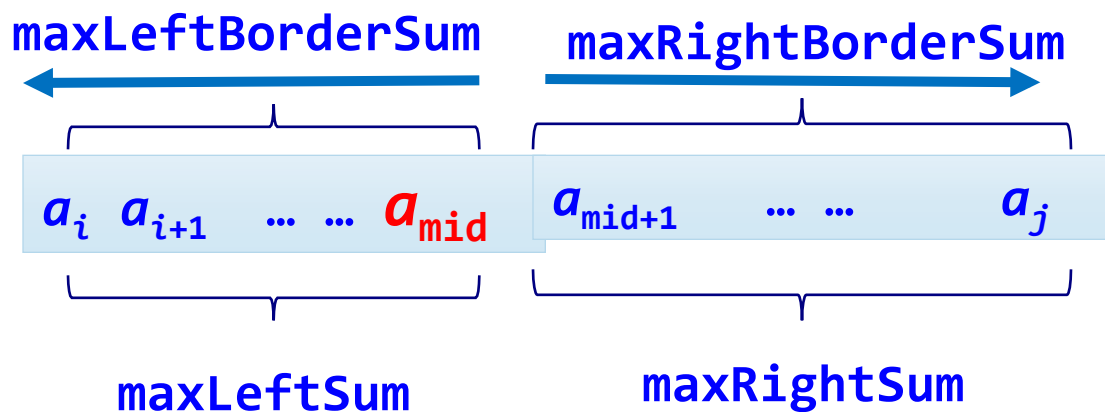
$\text{max3}(8, 6, 14) = 14$

3.3 分治法求解组合问题

求解最大连续子序列和问题

【问题求解】

考虑该子序列跨越序列 a_{mid} 元素的情况



结果: $\text{max3}(\text{maxLeftSum}, \text{maxRightSum}, \text{maxLeftBorderSum} + \text{maxRightBorderSum})$

3.3 分治法求解组合问题

求解最大连续子序列和问题

【算法描述】

```
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;
    }
}
```

3.3 分治法求解组合问题

求解最大连续子序列和问题

```
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);
}
```

3.3 分治法求解组合问题

求解最大连续子序列和问题

【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：

$$T(n)=1 \quad \text{当 } n=1$$

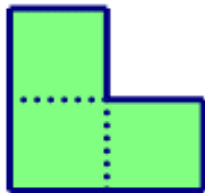
$$T(n)=2T(n/2)+n \quad \text{当 } n>1$$

容易推出， $T(n)=O(n\log_2 n)$ 。

3.3 分治法求解组合问题

求解棋盘覆盖

【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。



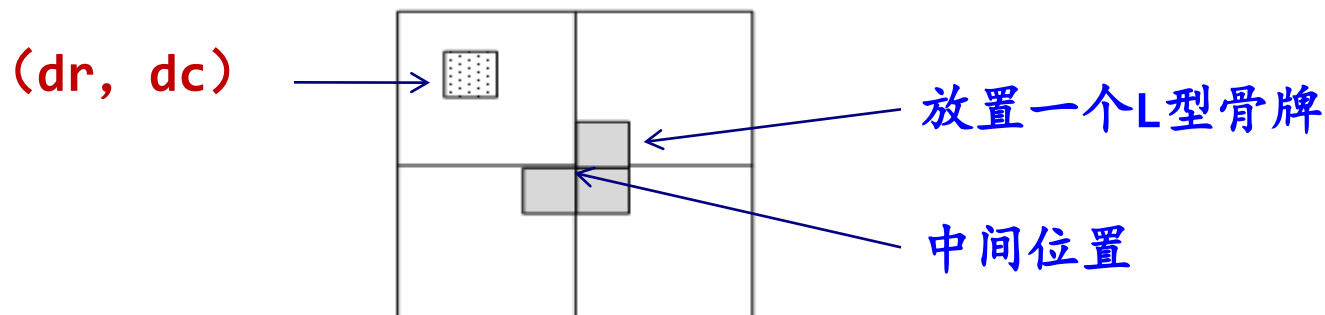
3.3 分治法求解组合问题

求解棋盘覆盖

【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！

棋盘左上角位置

(tr, dc)

特殊方格位置

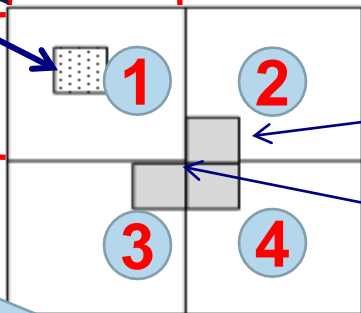
(dr, dc)

size

Size/2

Size/2

size



放置一个L型骨牌

中间位置

Size

特殊方格位置

初始问题:

将
特殊方格
用L型骨牌

问题:

- 1: 如何确定棋盘尺寸size
- 2: 如何确定棋盘左上角坐标 (tr, tc)
- 3: 如何确定特殊位置坐标?
 (dr, dc)

将一个size/2×size/2的棋盘, 其中特殊方格在 (dr, dc) 的棋盘用L型骨牌进行覆盖。

将一个size/2×size/2的棋盘, 其中特殊方格在左上角的棋盘用L型骨牌进行覆盖。

将一个size/2×size/2的棋盘, 其中特殊方格在右上角的棋盘用L型骨牌进行覆盖。

将一个size/2×size/2的棋盘, 其中特殊方格在左下角的棋盘用L型骨牌进行覆盖。

4

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



左上角
象限

右上角
象限

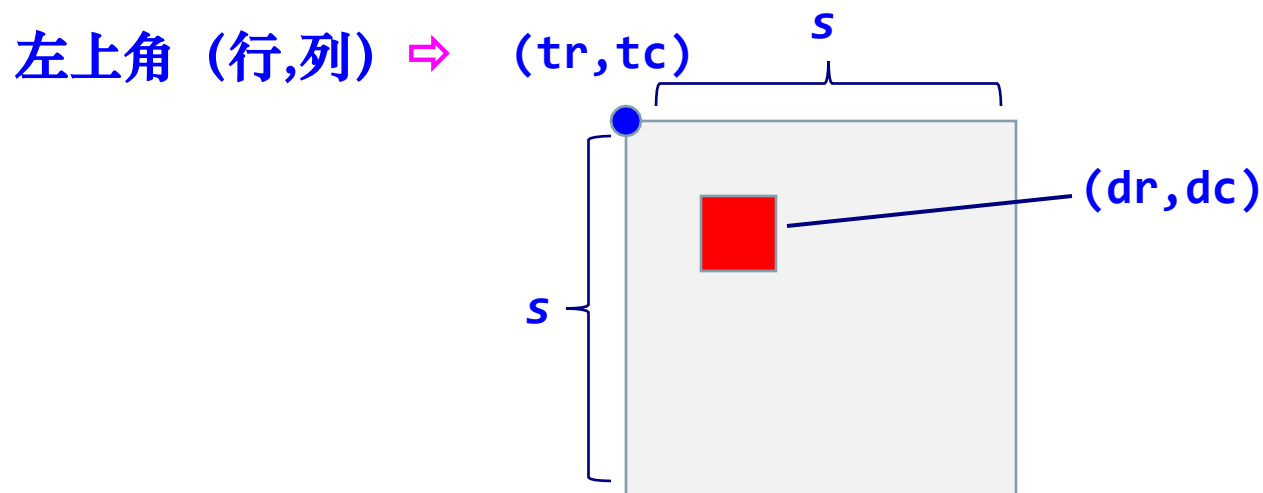
左下角
象限

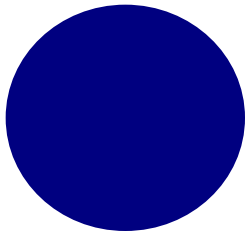
右下角
象限

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

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;
```

```
int x,y;
```

```
//棋盘大小
```

```
//特殊方格的位置
```

```
//求解问题表示
```

```
int board[MAX][MAX];
```

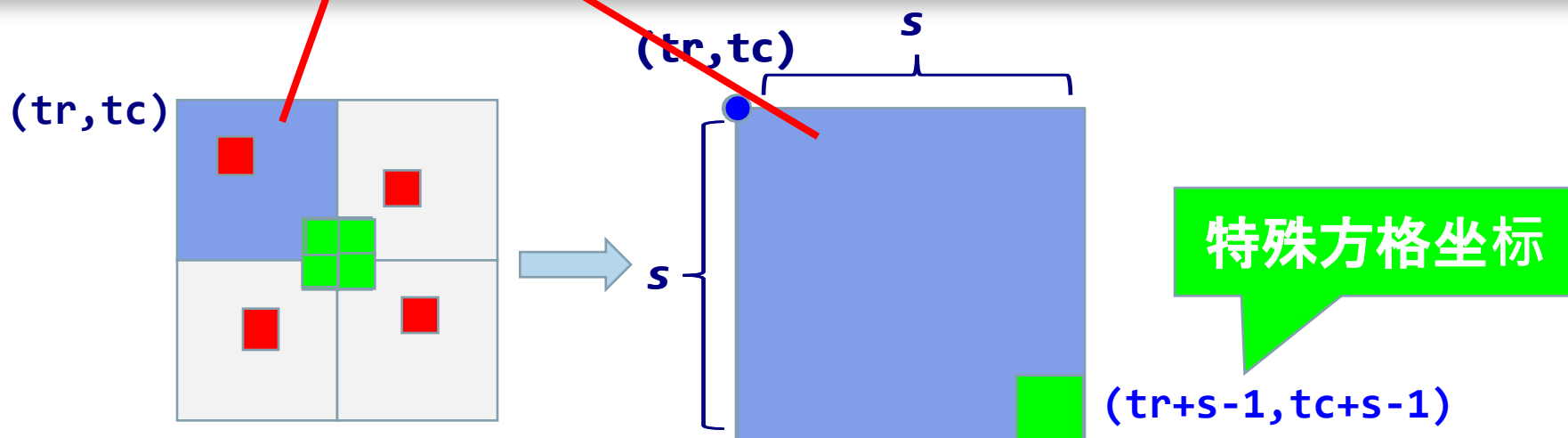
```
int tile=1;
```

```

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
    { //用t号L型骨牌覆盖右下角
        board[tr+s-1][tc+s-1]=t;
        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);
```

```
else
```

```
{ board[tr+s-1][tc+s]=t;
```

```
    ChessBoard(tr,tc+s,tr+s-1,tc+s,s);
```

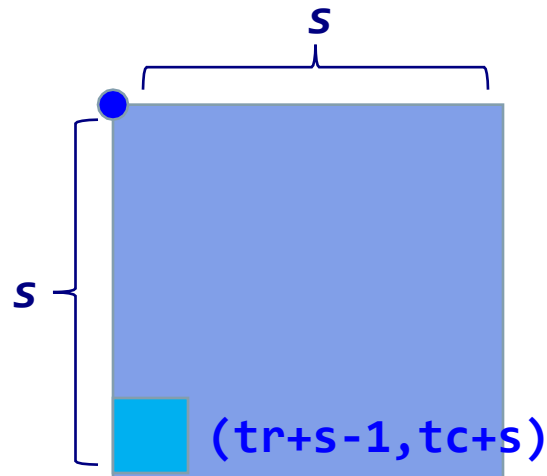
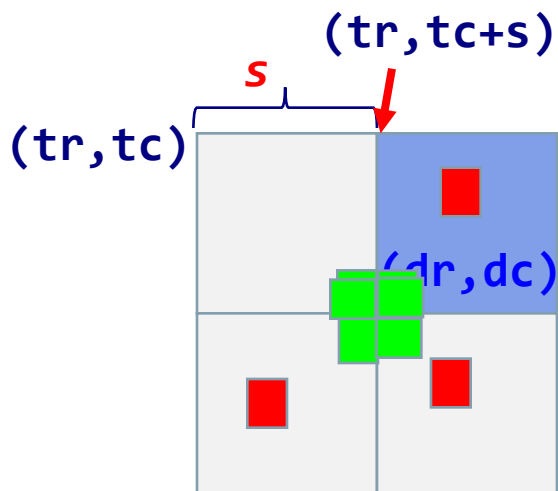
```
    //将左下角作为特殊方格继续处理该象限
```

```
}
```

//特殊方格在此象限中

//此象限中无特殊方格

//用t号L型骨牌覆盖左下角



//处理左下角象限

```
if(dr >= tr+s && dc < tc+s)
```

```
    ChessBoard(tr+s,tc,dr,dc,s);
```

```
else
```

```
{    board[tr+s][tc+s-1]=t;
```

```
    ChessBoard(tr+s,tc,tr+s,tc+s-1,s);
```

```
}
```

//特殊方格在此象限中

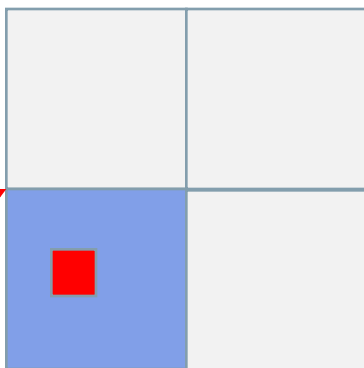
//此象限中无特殊方格

//用t号L型骨牌覆盖右上角

//将右上角作为特殊方格继续处理该象限

(tr,tc)

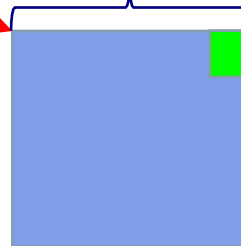
(tr+s,tc)



s

(tr+s,tc)

(tr+s,tc+s-1)



//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

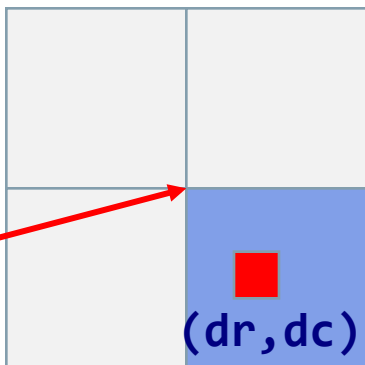
```
    ChessBoard(tr+s,tc+s,tr+s,tc+s,s);
```

//将左上角作为特殊方格继续处理该象限

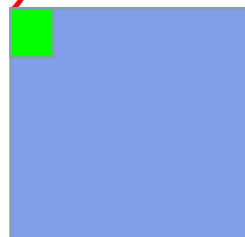
```
}
```

```
}
```

(tr,tc)



(tr+s,tc+s)



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



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

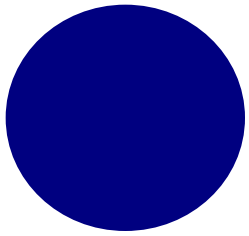


3.5 求解大整数乘法和矩阵乘法问题

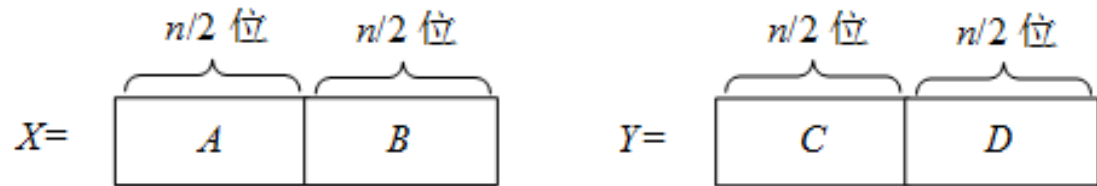
3.5.1 求解大整数乘法问题

【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：

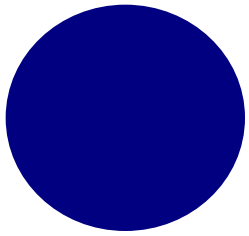
$$T(n)=O(1)$$

当 $n=1$

$$T(n)=4T(n/2)+O(n)$$

当 $n>1$

由此可得 $T(n)=O(n^2)$ 。

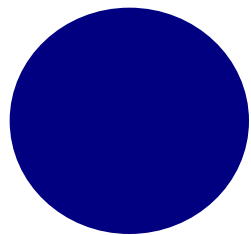


采用分治法，把 $X*Y$ 写成另一种形式：

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

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：

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



图片网 tupic.com/ 85215525

