三 、基本的算法策略

3、分而治之算法

- 3.1 分治算法框架
- 3.2 二分法
- 3.3 其它分治方法

3.1 分治算法框架

- 1. 算法设计思想
  - 分治法的过程是将整个问题分解成若干个小问题后分而治之。如果分解得到的子问题相对来说还太大，则可反复使用分治策略将这些子问题分成更小的同类型子问题，直至产生出方便求解的子问题，必要时逐步合并这些子问题的解，从而得到问题的解。
  - 分治法的基本步骤在每一层递归上都有三个步骤：
    - （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）解决：若子问题规模较小而容易被解决则直接解，否则再继续分解为更小的子问题；
      （3）合并：将已求解的各个子问题的解，逐步合并为原问题的解。

3.1 分治算法框架

- 1．算法设计思想
  - 有时问题分解后，不必求解所有的子问题，也就不必作第三步的操作。比如折半查找，在判别出问题的解在某一个子问题中后，其它的子问题就不必求解了。分治法的这类应用，又称为“减治法”。
  - 多数问题需要所有子问题的解，并由子问题的解，使用恰当的方法合并成为整个问题的解，比如归并排序，就是不断将子问题中已排好序的解合并成较大规模的有序子集。

3.1 分治算法框架

- 2．适合用分治法策略的问题
  - 当求解一个输入规模为n且值相当大的问题时，用蛮力策略效率一般得不到保证。若问题能满足以下几个条件，就能用分治法来提高解决问题的效率。
    - （1）能将这n个数据分解成k个不同子集，且得到k个子集是可以独立求解的子问题，其中1＜k≤n；
    - （2）分解所得到的子问题与原问题具有相似的结构，便于利用递归或循环机制；
    - （3）在求出这些子问题的解之后，就可以推解出原问题的解。

3.1 分治算法框架

- 分治法的一般的算法设计模式如下：

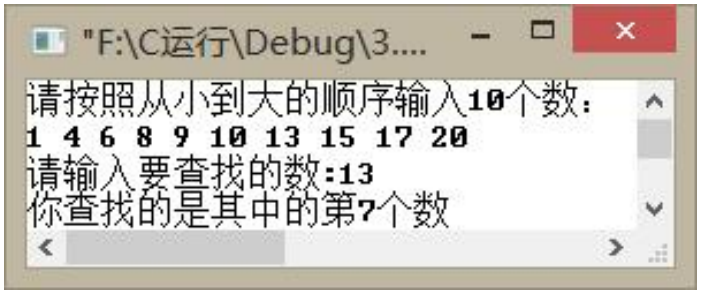
```
Divide(int  n)    //n为问题规模
{
   if (n≤n0)      //n0 为可解子问题的规模,当问题Pi的规模不超过n0时
   {    解子问题;
        return(子问题的解); }
   for (i=1 ;i<=k;i++)          //分解为较小子问题p1,p2,……pk
        yi=Divide (Pi);         //递归解决Pi
   T =MERGE(y1,y2,...,yk);    //合并子问题，在一些问题中不需要这一步
   return(T);
}
```

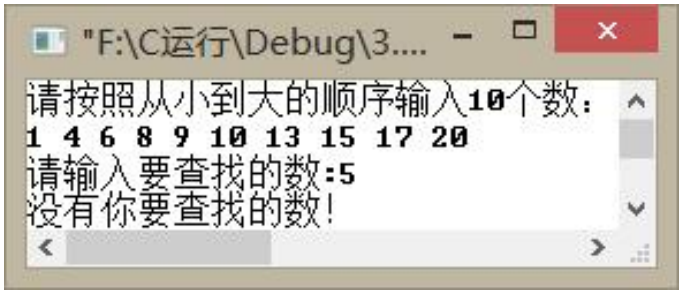
3.2 二分法

- 在算法设计中每次一个问题分解成的子问题个数一般是固定的，每个子问题的规模也是平均分配的。当每次都将问题分解为原问题规模的一半时，称为二分法。二分法是分治法较常用的分解策略，数据结构课程中的折半查找、归并排序等算法都是采用此策略实现的。

3.2 二分法

- 例：折半查找
  - 先求位于升序查找区间正中的对象的下标mid，用其与给定值key比较:
  - a[mid]= key ，查找成功；
  - a[mid]> key ，把查找区间缩小到表的前半部分，再继续进行折半查找；
  - a[mid]< key ，把查找区间缩小到表的后半部分，再继续进行折半查找。
  - 每比较一次，查找区间缩小一半。如果查找区间已缩小到一个对象，仍未找到想要查找的对象，则查找失败。





3.2 二分法

- 算法实现：

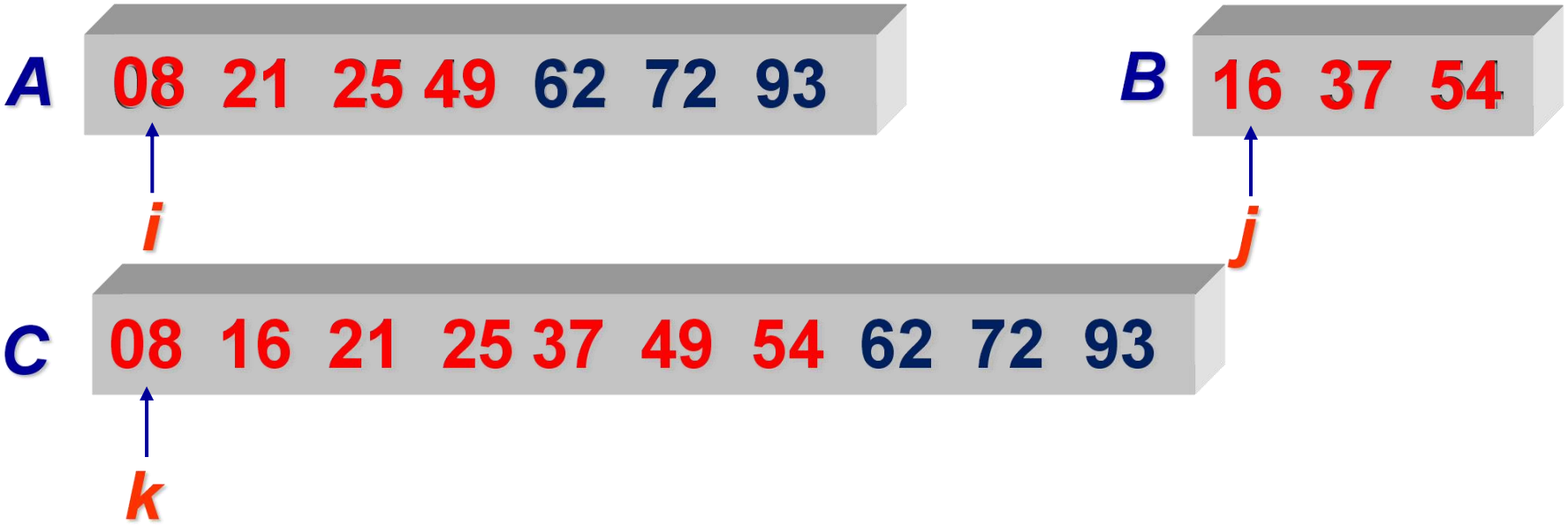
```
main()
{
   int a[10],key,i,low=0,high=9,mid;
   printf("请按照从小到大的顺序输入10个数：\n");
   for(i=0;i<10;i++)
        scanf("%d",&a[i]);
   printf("请输入要查找的数:");
   scanf("%d",&key);
```

3.2 二分法

```
    while(low<=high)
    {    mid=(low+high)/2;
         if(a[mid]>key)
                high=mid-1;
         else
                if(a[mid]<key)
                       low=mid+1;
                else
                {      printf("你查找的是其中的第%d个数\n",mid+1);
                       return;
                }
    }
    printf("没有你要查找的数！\n");
}
```

3.2 二分法

- 归并：
  - 设两个有序表A和B，变量i和j分别是表A和表B的当前检测位置。设表C是归并后的新有序表，变量k是它的当前存放位置。
  - 当i和j都在两个表的表长内变化时，根据A[i]与B[j]的关键字的大小，依次把关键字小的对象排放到新表C[k]中；
  - 当i与j中有一个已经超出表长时，将另一个表中的剩余部分照抄到新表C[k]中。



3.2 二分法

- 例：归并排序
- 通常用递归实现，先把待排序区间[low,high]以中点mid二分，接着把左边子区间[low,mid]进行递归归并排序，再把右边子区间[mid+1,high]进行递归归并排序，最后把左区间和右区间用一次归并操作合并成有序的区间。

3.2 二分法

- 算法实现：

```
void Merge(int a[],int c[], int low, int mid, int high)
{    // 归并算法
    int i = low, j=mid+1, k = low;
    while(i<=mid && j<=high)  //分别取前半部分和后半部分归并到c数组
    {
        if(a[i] > a[j])  //从小到大归并排序
            c[k++] = a[j++];
        else
            c[k++] = a[i++];
    }
```

3.2 二分法

```
    while(i<= mid) //后半部分已全部归并完毕
        c[k++] = a[i++]; //前半部分剩余的直接并入c数组
    while(j<= high) //前半部分已全部归并完毕
        c[k++] = a[j++]; //后半部分剩余的直接并入c数组
    for(i=low; i<=high; i++)
        a[i] = c[i];
}
```

3.2 二分法

```
void MergeSort(int a[], int c[], int low, int high)
{    //归并排序算法
    int mid;
    if(low < high)
    {
        mid = (low + high) / 2;
        MergeSort(a, c, low, mid);  //前半部分归并排序
        MergeSort(a, c, mid+1, high);  //后半部分归并排序
        Merge(a, c, low, mid, high);  //合并前后两部分
    }
}
```

3.2 二分法

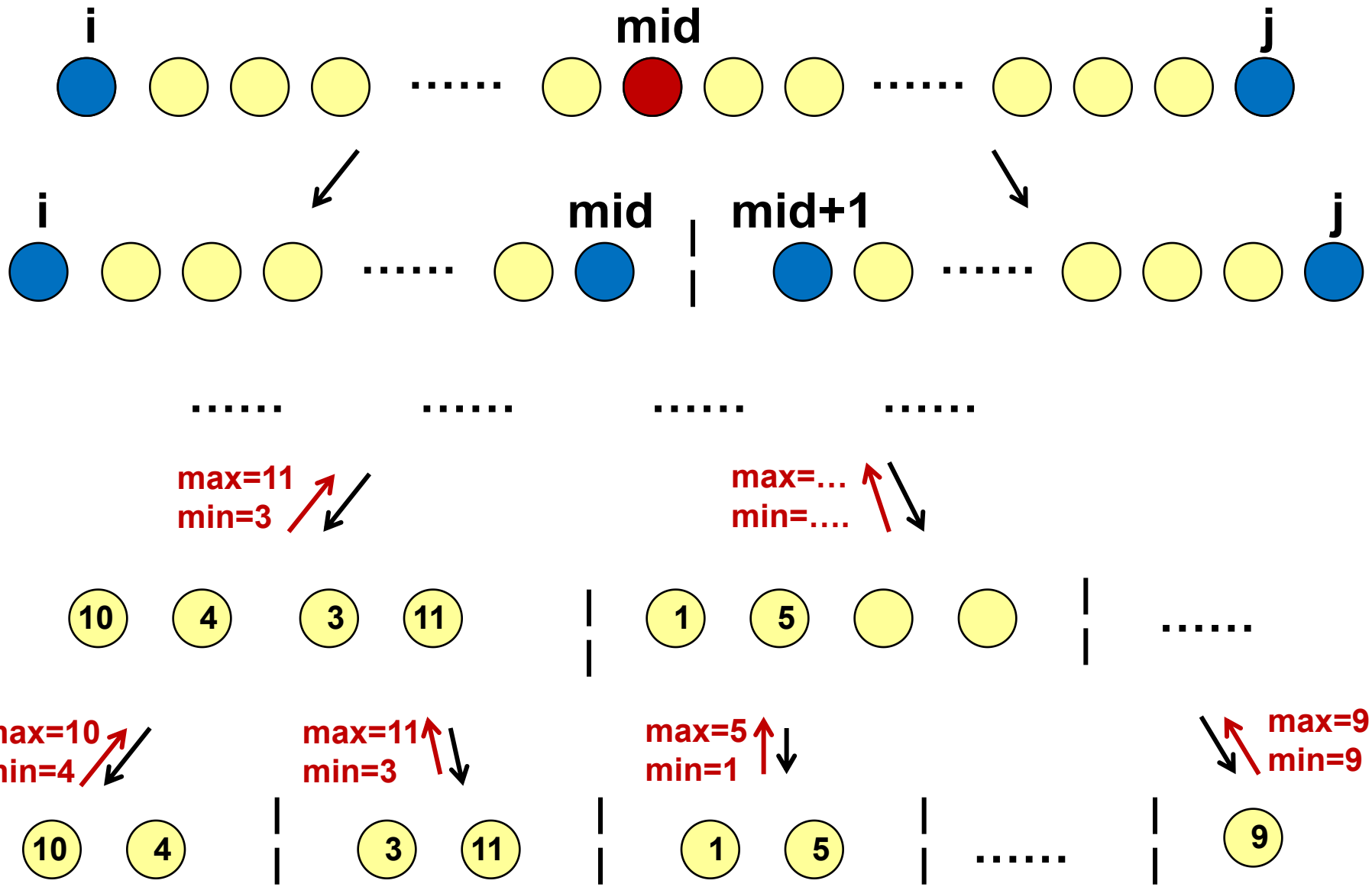
```
int main()
{
    int a[8] = {50, 10, 20, 30, 70, 40, 80, 60};
    int i, c[8];
    MergeSort(a, c, 0, 7);
    for(i=0; i<8; i++)
        printf("%d ", a[i]);
    printf("\n");
    return 0;
}
```

3.2 二分法

- 例：金块问题：老板有一袋金块(共n块)，最优秀的雇员得到其中最重的一块，最差的雇员得到其中最轻的一块。假设有一台比较重量的仪器，希望用最少的比较次数找出最重的金块。
- 算法设计1：比较简单的方法是逐个的进行比较查找。先拿两块比较重量，留下重的一个与下一块比较，直到全部比较完，找到最重的金子。算法类似于一趟选择排序。
- 算法分析1：算法中需要n-1次比较得到max。最好的情况是金块由小到大取出，不需要进行与min的比较，共进行n-1次比较。最坏的情况是金块是由大到小取出的，需要再经过n-1次比较得到min,共进行2*(n-1)次比较。平均情况下，比较数是3*(n-1)/2。

3.2 二分法

- 算法设计2：问题可以简化为在含n个元素的集合中寻找极大元和极小元。用二分法可以用较少比较次数解决上述问题：
  - （1）类似折半查找，先求出左半部分的最大值和最小值，再求解出右半部分的最大值和最小值然后合并求解。
  - （2）当划分以后只有一个元素或者两个元素的情况下，就能直接解出最大值和最小值，这时候就是递归的出口。


i
mid
j
i
mid
mid+1
j
max=11
min=3
max=…
min=….
10
4
3
11
1
5
max=10
min=4
max=11
min=3
max=5
min=1
max=9
min=9
10
4
3
11
1
5
9

3.2 二分法

- 算法实现：

```
int a[9]={10,4,3,11,1,5,12,8,9};
void maxmin(int i,int j,int *max,int *min)
{
   int mid;
   int lmax,lmin,rmax,rmin;
   if(i==j||i==j-1)  //每组有一个或两个元素时
   {
        if(a[i]<=a[j])
        {
              *max=a[j];
              *min=a[i];
        }
```

3.2 二分法

```
        else
        {
                *max=a[i];
                *min=a[j];
        }
    }
    else  //每组有多个元素时
    {
        mid=(i+j)/2;  //取每组中间位置
        maxmin(i,mid,&lmax,&lmin);    //左半部分递归求最大最小值
        maxmin(mid+1,j,&rmax,&rmin); //右半部分递归求最大最小值
        *max=lmax>rmax?lmax:rmax;
        *min=lmin>rmin?rmin:lmin;
    }
}
```

3.2 二分法

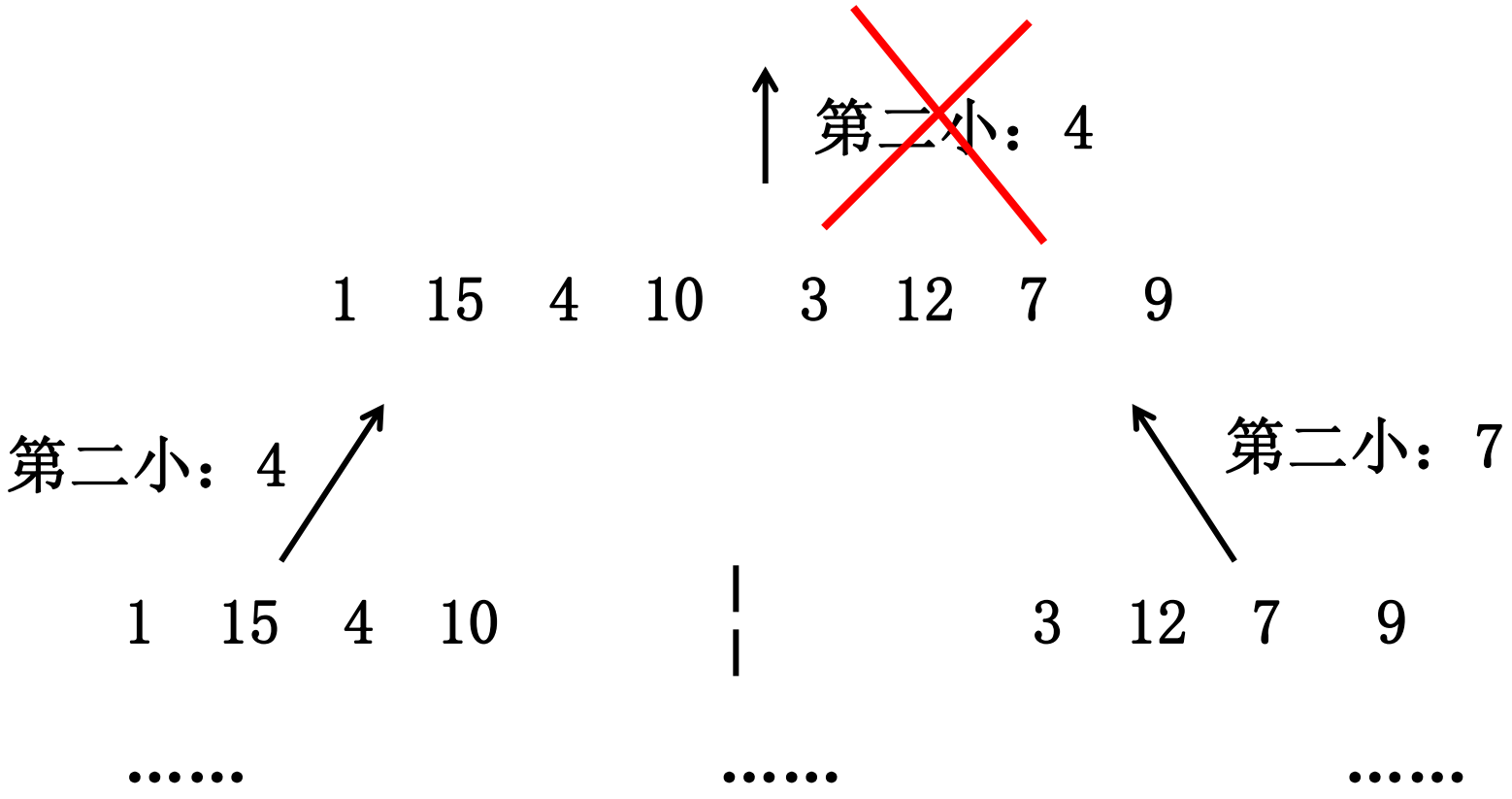
```
int main()
{
   int x=a[0],y=a[0];
   maxmin(0,8,&x,&y);
   printf("max=%d,min=%d\n",x,y);
   return 0;
}
```

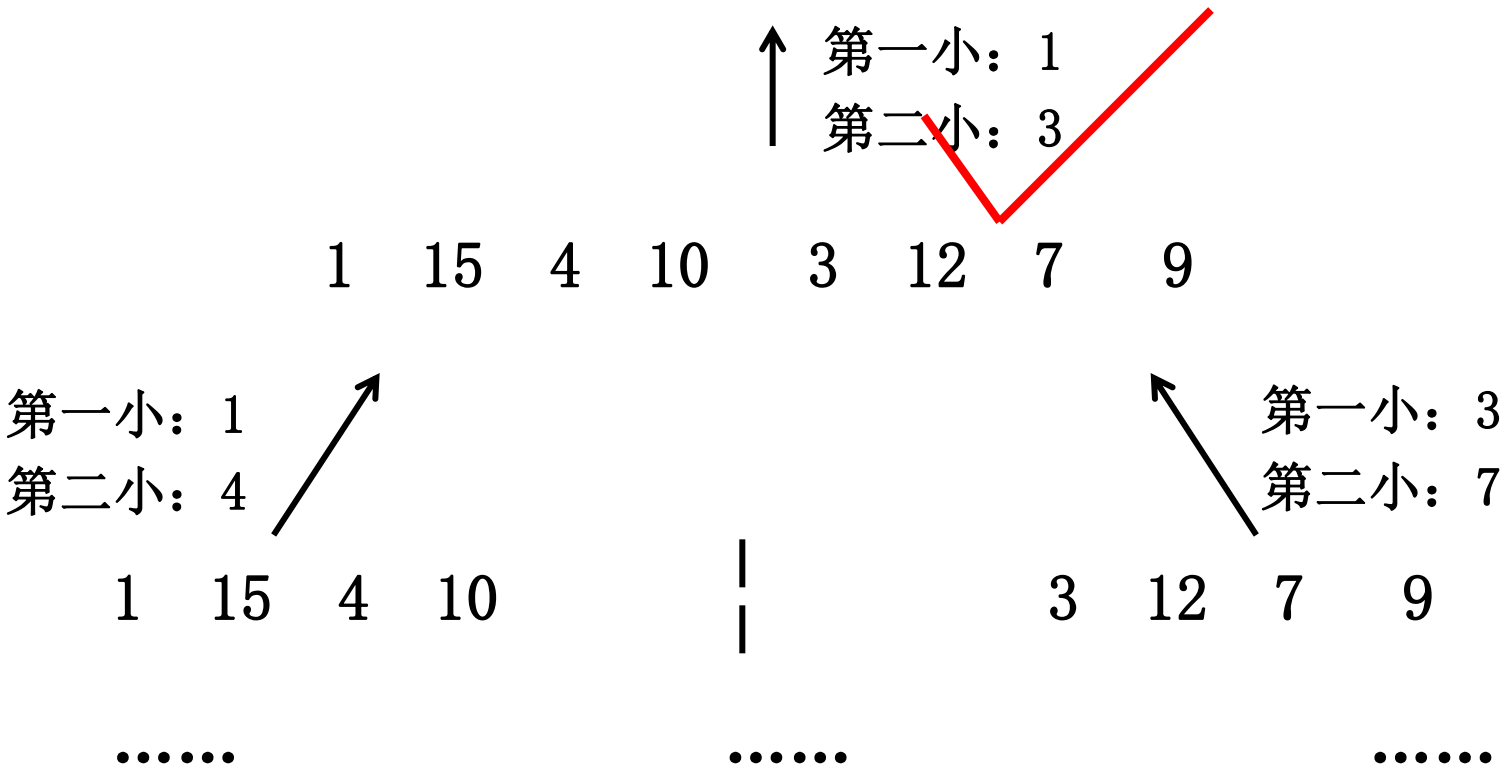
3.3 其它分治方法

- 以上的例子都是用二分策略把问题分解为与原问题相似“相等”的子问题。
- 下面讲解用“非等分二分法”解决问题的例子。

3.3 其它分治方法

- 例：选择问题：求一组数中第二小的数。
- 选择问题是“从一组数中选择第k小的数据”，这种问题不能用简单的二分法分解成完全独立、相似的两个子问题。因为在选出分解后第一组的第k小的数据和第二组的第k小的数据，不能保证这两个数据之一是原问题的解。


第二小：4
1 15 4 10 3 12 7 9
第二小：4
第二小：7
1 15 4 10
3 12 7 9
……
……
……


第一小：1
第二小：3
1 15 4 10 3 12 7 9
第一小：1
第二小：4
第一小：3
第二小：7
1 15 4 10
3 12 7 9
……
……
……

3.3 其它分治方法

- 算法实现：

```
int a[10]={10,4,3,11,1,5,12,8,9,2};
second(int i,int j,int *min2,int *min1)
{
   int lmin2,lmin1,rmin2,rmin1;
   int mid;
   if(i==j)     //每组只有一个元素时
        *min2=*min1=a[i];
```

3.3 其它分治方法

```
else
     if(i==j-1)       //每组有两个元素时
              if(a[i]<a[j])
              {
                       *min2=a[j];
                       *min1=a[i];
              }
              else
              {
                       *min2=a[i];
                       *min1=a[j];
              }
```

3.3 其它分治方法

```
else        //每组有多个元素时
{
        mid=(i+j)/2;     //取每组中间位置
        second(i,mid,&lmin2,&lmin1); //左半部分递归求第一和第二小值
      second(mid+1,j,&rmin2,&rmin1); //右半部分递归求第一和第二小值
        if (lmin1<rmin1)
                if(lmin2<rmin1)    //lmin1<lmin2<rmin1
                {
                        *min1=lmin1;
                        *min2=lmin2;
                }
                else     //lmin1<rmin1<lmin2
                {
                        *min1=lmin1;
                        *min2=rmin1;
                }
```

3.3 其它分治方法

```
                else
                        if (rmin2<lmin1)  //rmin1<rmin2<lmin1
                        {
                                *min1=rmin1;
                                *min2=rmin2;
                        }
                        else     //rmin1<lmin1<rmin2
                        {
                                *min1=rmin1;
                                *min2=lmin1;
                        }
        }
}
main()
{  int x=a[0],y=a[0];
   second(0,9,&x,&y);
   printf("第一小值为%d,第二小值为%d\n",y,x);
}
```