算法-分治

信智学院　陈琰宏

教学内容

01 分治原理

02 分析经典问题分析

03 扩展

1.1 分治法概述

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。

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。

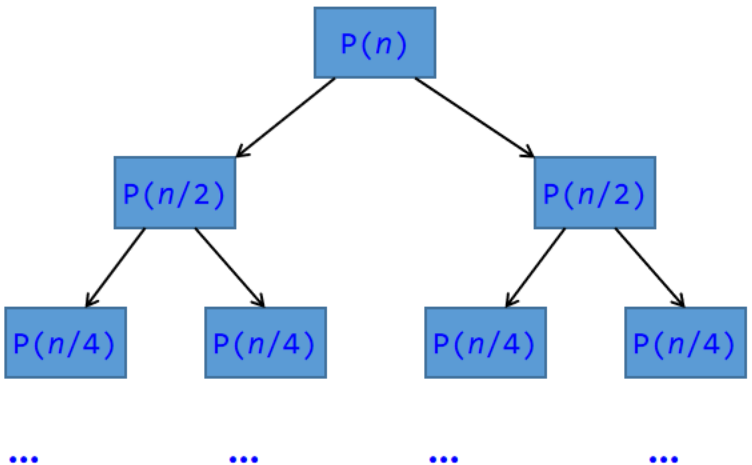
③ 合并：将各个子问题的解合并为原问题的解。

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 二分（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
}
```

【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：

$$C(n)=1 \qquad \text{当} n=1$$

$$C(n)\le 1+C(\lfloor n/2 \rfloor) \qquad \text{当} n\ge 2$$

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

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 [3905] 二分搜索

给定n个元素，使用二分法从中查找特定元素x。

输入：包含T组数据。先给定一个T。每组数据第一行是n，第二行是n个数。第三行为要查找的数。 xT<2000,n<100000

输出：先输出排序后的数组。如果找到x，则输出x的位置；
如果没找到，输出 “-1”。

样例输入
1
11
60 17 39 15 8 34 30 45 5 52 25
17
样例输出
5 8 15 17 25 30 34 39 45 52 60
4

```cpp
#include<iostream>
#include<algorithm>
using namespace std;
const int M=200000;
int x,n,i;
int s[M];

int BinarySearch(int n,int s[],int x)
{
    int low=0,high=n-1;
    //low指向有序数组的第一个元素，high指向有序数组的最后一个元素
    while(low<=high)
    {
        int middle=(low+high)/2;  //middle为查找范围的中间值
        if(x==s[middle])  //x等于查找范围的中间值，算法结束
            return middle;
        else if(x>s[middle])//x大于查找范围的中间元素，则从左半部分查找
            low=middle+1;
        else   //x小于查找范围的中间元素，则从右半部分查找
            high=middle-1;
    }
    return -1;
}
```

```cpp
int main()
{
    int T;
    cin>>T;
    while(T--)
    {
        cin>>n;
        for(int i=0;i<n;i++)cin>>s[i];
        sort(s,s+n);
        cout<<s[0];
        for(int i=1;i<n;i++)cout<<" "<<s[i];
        cout<<endl;
        cin>>x;
        int k=BinarySearch(n,s,x);
        if(k==-1)cout<<k<<endl;
        else cout<<k+1<<endl;
    }
    return 0;
}
```

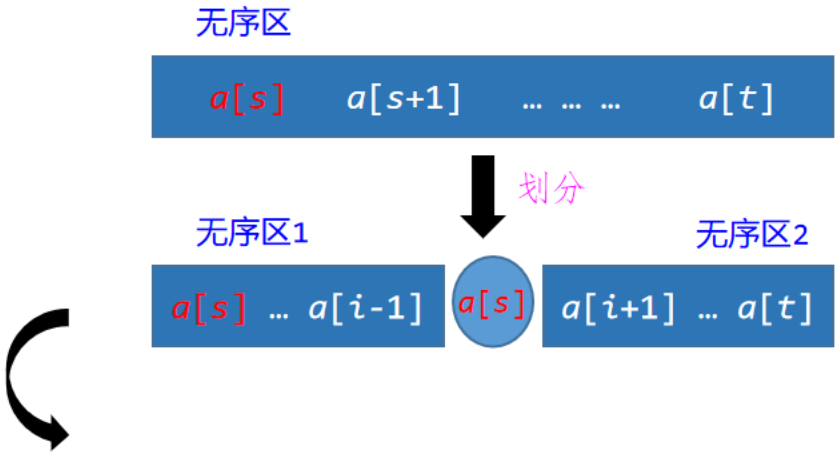
2 求解排序问题

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。

2.1 快速排序




```
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);
```

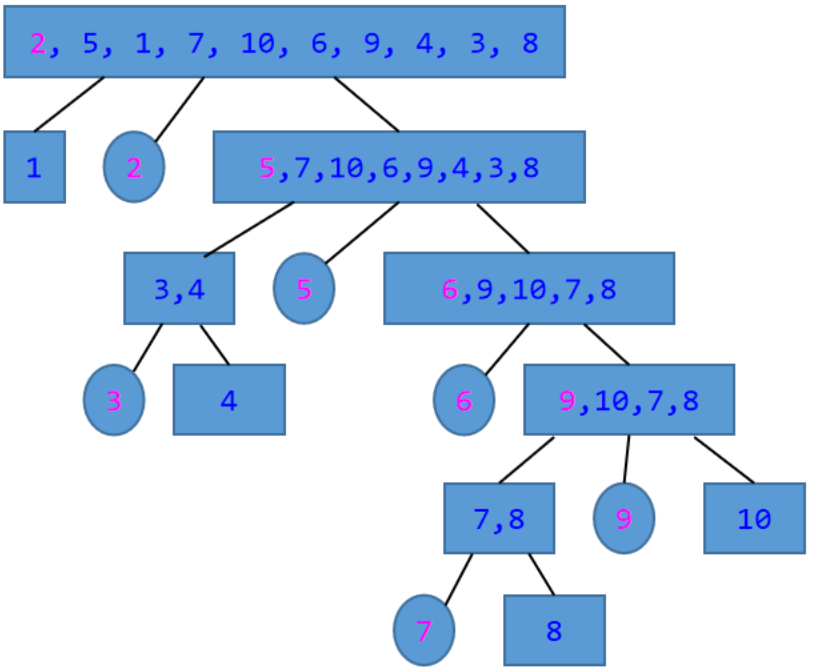
2.1 快速排序分治策略

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。

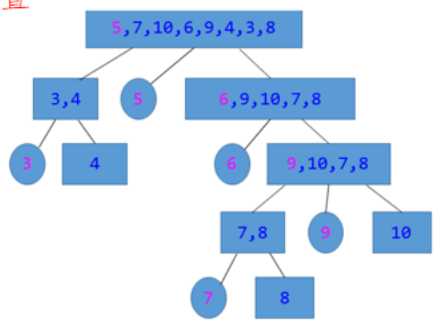
③ 合并：由于整个序列存放在数组中a中，排序过程是就地进行的，合并步骤不需要执行任何操作。

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。



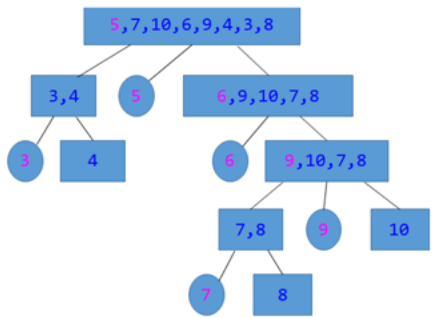
快速排序算法：

```
int Partition(int a[], int s, int t) //划分算法
{   int i=s, j=t;//s左边界 t右边界
    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
   }
}
```



快速排序算法性能分析

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$O(n\log_2 n)$。

1 [2496] 排序（1 <= n <= 100000）

题目描述
输入n组测试数据，从小到大排序。
输入
输出
样例输入
2
3
3 6 5
4
8 5 9 7
样例输出
3 5 6
5 7 8 9
提示
（1 <= n <= 100000）

写法1

```
#include<iostream>
using namespace std;
int a[100005];
int Partition(int s,int t)  //划分算法
{   int i=s,j=t;//s左边界 t右边界
    int tmp=a[s];   //用序列的第1个记录
    while (i!=j)    //从序列两端交替向
    {   while (j>i && a[j]>=tmp)
            j--;        //从右向左扫描,
        a[i]=a[j];  //将a[j]前移到a[i]
        while (i<j && a[i]<=tmp)
            i++;    //从左向右扫描,找第
        a[j]=a[i];  //将a[i]后移到a[j]
    }
    a[i]=tmp;
    return i;
}
```

```
void QuickSort(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(s,t);
        QuickSort(s,i-1);     //对左子序列递归排序
        QuickSort(i+1,t);     //对右子序列递归排序
    }
}
int main(){
    int n,m;
    cin>>n;
    for(int i=1;i<=n;i++){
        cin>>m;
        for(int j=1;j<=m;j++)
            cin>>a[j];
        QuickSort(1,m);
        for(int j=1;j<=m;j++)
            cout<<a[j]<<" ";
        cout<<endl;
    }
    return 0;
}
```

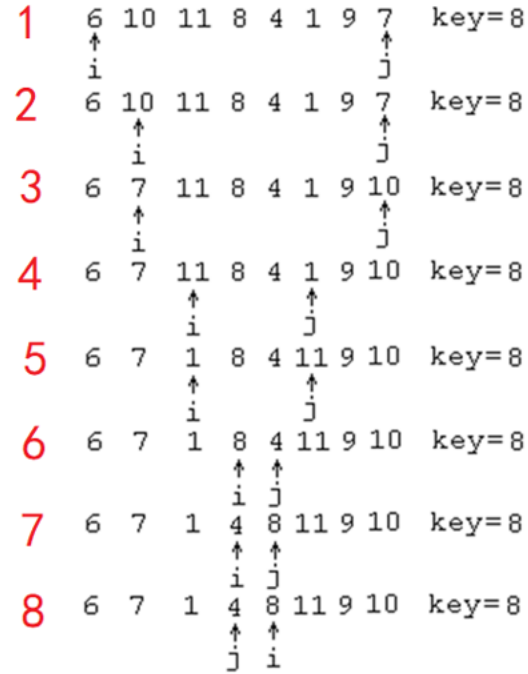
写法2

1. 设置两个变量i、j，排序开始的时候：i=0，j=N-1；
2. 以数组中任意元素作为关键数据(一般以数组中间元素作为关键数据)，赋值给key，可以是key=A[(i+j)/2]；
3. 从j开始向前搜索，即由后开始向前搜索(j--)，找到第一个小于等于key的值A[j]；
4. 从i开始向后搜索，即由前开始向后搜索(i++)，找到第一个大于等于key的A[i]；
5. 交换A[i]和A[j]的值，同时i++，j--；
6. 重复第3、4、5步，直到i>j；

例如有8个元素需要排序：

6 10 11 8 4 1 9 7

一趟快速排序后：



此时i>j,并且i左边的数字都小于等于key,j右边的数字都大于等于key，进而接下来可以分别对左边段[0,j]和右边段[i,N-1]利用同样的方法排序。

```
void qsort(int le,int ri)
{
     int i=le, j=ri, mid=a[(le+ri)/2];
     while(i<=j)                              //注意这里要有等号
     {
          while(a[i]<mid) i++;                //在左边找大于等于mid的数
          while(a[j]>mid) j--;                //在右边找小于等于mid的数
          if(i<=j)
          {
               swap(a[i],a[j]);               //交换
               i++, j--;                      //继续找
           }
      }
      if(le<j) qsort(le,j);                  //分别递归继续排序
      if(i<ri) qsort(i,ri);
}
```

```cpp
#include<iostream>
using namespace std;
int a[100005];
void qsort(int le, int ri){//left right
    //le左边界 ri右边界
    int i=le,j=ri,mid=a[(le+ri)/2];
    while(i<=j){//从序列两端交替向中间扫描,直至i=j为止
        //向右扫描,找第1个关键字大于等于mid的a[i]
        while(a[i]<mid)i++;
        //向右扫描,找第1个关键字小于等于mid的a[j]
        while(a[j]>mid)j--;
        if(i<=j){
            swap(a[i],a[j]);
            i++,j--;
        }
    }
    if(le<j)qsort(le,j);
    if(i<ri)qsort(i,ri);
}
```

```cpp
int main(){
    int n,m;
    cin>>n;
    for(int i=1;i<=n;i++){
        cin>>m;
        for(int j=1;j<=m;j++)
            cin>>a[j];
        qsort(1,m);
        for(int j=1;j<=m;j++)
            cout<<a[j]<<" ";
        cout<<endl;
    }
    return 0;
}
```

思考：

第一个代码会超时，第二个代码能通过。快排如果知道了选取基准数的规则，在特定数据下，会退化到n^2

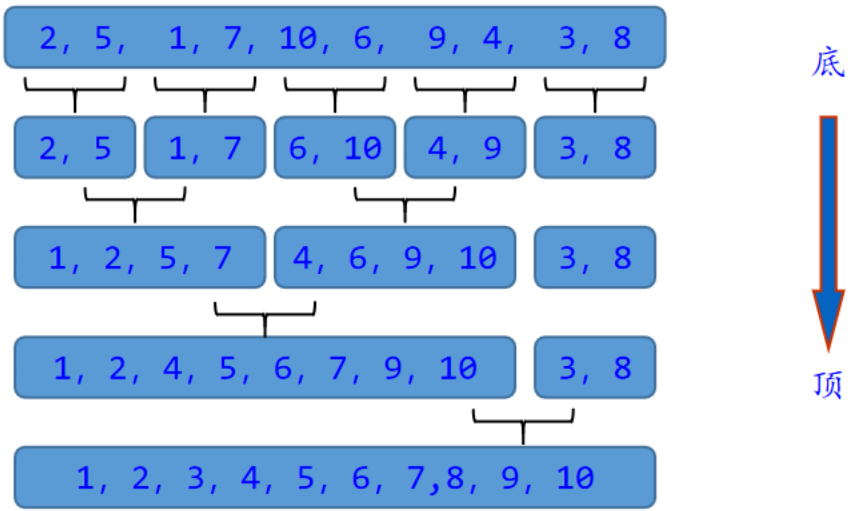
2.2 归并排序

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（$k\geqslant 2$）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。

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。

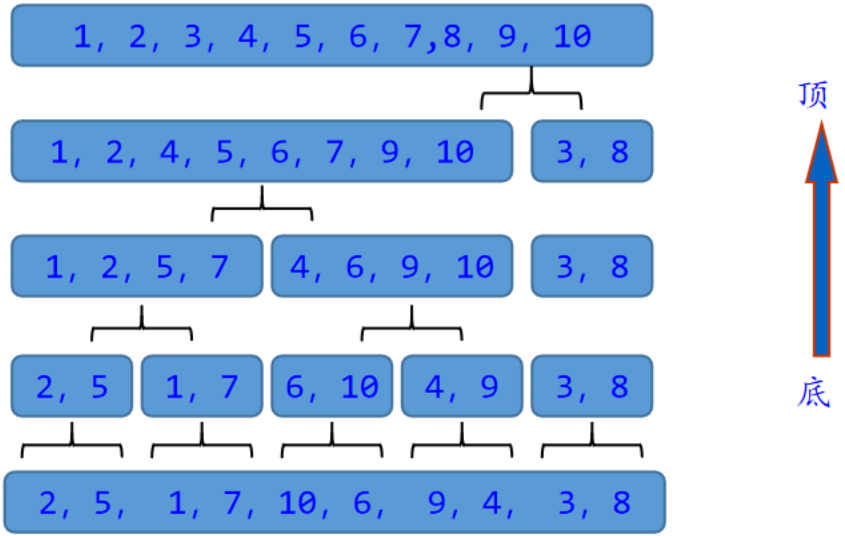
2.2.1 自底向上的二路归并排序算法

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。



2.2.1 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：

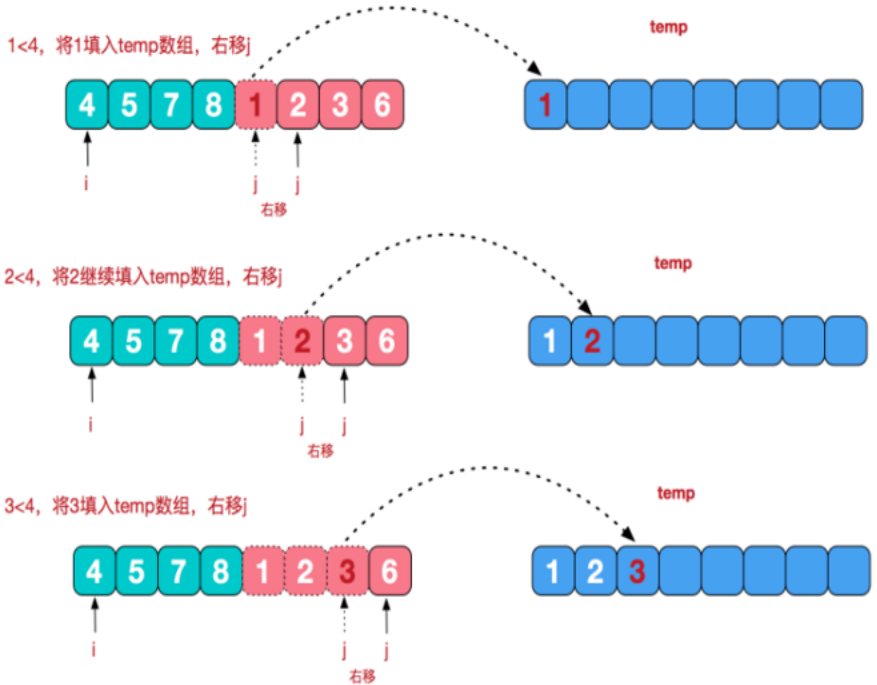
循环$\lceil \log_2 n \rceil$次，length依次取1、2、…、$\log_2 n$。每次执行以下步骤：

① 分解：将原序列分解成length长度的若干子序列。

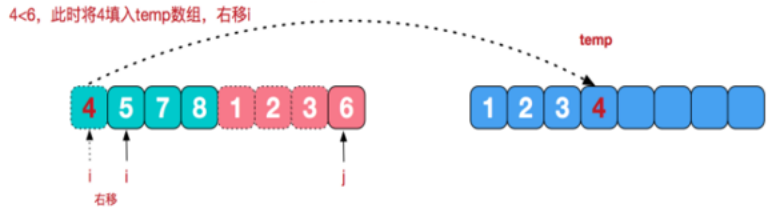
② 求解子问题：将相邻的两个子序列调用Merge算法合并成一个有序子序列。

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。

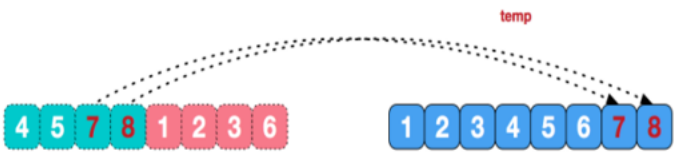
合并如下：


1<4，将1填入temp数组，右移j
temp
4 5 7 8 1 2 3 6
1
i
j
j
右移
2<4，将2继续填入temp数组，右移j
temp
4 5 7 8 1 2 3 6
1 2
i
j
j
右移
3<4，将3填入temp数组，右移j
temp
4 5 7 8 1 2 3 6
1 2 3
i
j
j
右移

合并如下：



继续重复这种比较+填入的步骤，直到右子序列已经填完，这时将左边剩余的7和8依次填入



最后，将temp中的内容全部拷到原数组中去，排序完成

```
1  void Merge(int a[],int low,int mid,int high)
2  //a[low..mid]和a[mid+1..high]→ a[low..high]
3  {
4     int tmpa[MAXN];
5     int i=low,j=mid+1,k=0;
6     while (i<=mid && j<=high){
7          if (a[i]<=a[j])      //将第1子表中的元素放入tmpa中
8          {  tmpa[k]=a[i];  i++; k++; }
9          else               //将第2子表中的元素放入tmpa中
10         {  tmpa[k]=a[j];  j++; k++; }
11    }
12
13    while (i<=mid)         //将第1子表余下部分复制到tmpa
14    {  tmpa[k]=a[i]; i++; k++; }
15
16    while (j<=high)        //将第2子表余下部分复制到tmpa
17    {  tmpa[k]=a[j]; j++; k++; }
18
19    for (k=0, i=low;i<=high;k++,i++)  //将tmpa复制回a中
20       a[i]=tmpa[k];
21 }
```

```
1  void MergePass(int a[],int length,int n)
2  //一趟二路归并排序
3  {  int i;
4     //归并length长的两相邻子表
5     for (i=0;i+2*length-1<n;i=i+2*length)
6          Merge(a,i,i+length-1,i+2*length-1);
7     if (i+length-1<n)//余下两个子表,后者长度小于length
8          Merge(a,i,i+length-1,n-1);    //归并这两个子表
9  }
```

```
1  void MergeSort(int a[],int n)//二路归并算法
2  {  int length;
3     for (length=1;length<n;length=2*length)
4          MergePass(a,length,n);
5  }
```

【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$O(n\log_2 n)$。

1 [2496] 排序（1 <= n <= 100000）

```cpp
#include<bits/stdc++.h>
using namespace std;
int a[100005];
void Merge(int a[],int low,int mid,int high)
//a[low..mid]和a[mid+1..high]→ a[low..high]
{  int *tmpa;
void MergePass(int a[],int length,int n)
//一趟二路归并排序
{  int i;

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);
}

int main(){
    int n,m;
    cin>>n;
    for(int i=1;i<=n;i++){
        cin>>m;
        for(int j=0;j<m;j++)
            cin>>a[j];
        MergeSort(a,m);
        for(int j=0;j<m;j++)
            cout<<a[j]<<" ";
        cout<<endl;
    }
    return 0;
}
```

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

```cpp
void Merge(int a[],int low,int mid,int high)
//a[low..mid]和a[mid+1..high]→ a[low..high]
{
   int tmpa[MAXN];
   int i=low,j=mid+1,k=0;
   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];
}
```

2 [3431] 归并排序

归并排序是基于归并操作完成的，而一次归并操作是通过两个或两个以上的有序表合并成一个新的有序表完成的。常见的归并排序是2-路归并排序，其核心操作是将一维数组中前后相邻的两个有序序列归并成一个有序序列。

输入
输入的第一行包含1个正整数n，表示共有n个整数需要参与排序。其中n不超过100000。
第二行包含n个用空格隔开的正整数，表示n个需要排序的整数。
输出
只有1行，包含n个整数，表示从小到大排序完毕的所有整数。
请在每个整数后输出一个空格，并请注意行尾输出换行。

2 [3431] 归并排序

```
3   int a[100005];
4   void Merge(int a[],int low,int mid,int high)
5   //a[low..mid]和a[mid+1..high]→ a[low..high]
6   {  int *tmpa;
7      int i=low,j=mid+1,k=0;
8      tmpa=(int *)malloc((high-low+1)*sizeof(int));
9      while (i<=mid && j<=high)
10       if (a[i]<=a[j])//将第1子表中的元素放入tmpa中
11     {  tmpa[k]=a[i]; i++; k++; }
12     else//将第2子表中的元素放入tmpa中
13     {  tmpa[k]=a[j]; j++; k++; }
14     while (i<=mid)        //将第1子表余下部分复制到tmpa
15     {  tmpa[k]=a[i]; i++; k++; }
16     while (j<=high)       //将第2子表余下部分复制到tmpa
17     {  tmpa[k]=a[j]; j++; k++; }
18     for (k=0,i=low;i<=high;k++,i++)  //将tmpa复制回a中
19      a[i]=tmpa[k];
20     free(tmpa);           //释放tmpa所占内存空间
21  }
22  void MergePass(int a[],int length,int n)
23  //一趟二路归并排序
24  {  int i;
25     for (i=0;i+2*length-1<n;i=i+2*length)    //归并length长的两相邻子表
26      Merge(a,i,i+length-1,i+2*length-1);
27     if (i+length-1<n)                        //余下两个子表,后者长度小于length
28      Merge(a,i,i+length-1,n-1);    //归并这两个子表
29  }
```

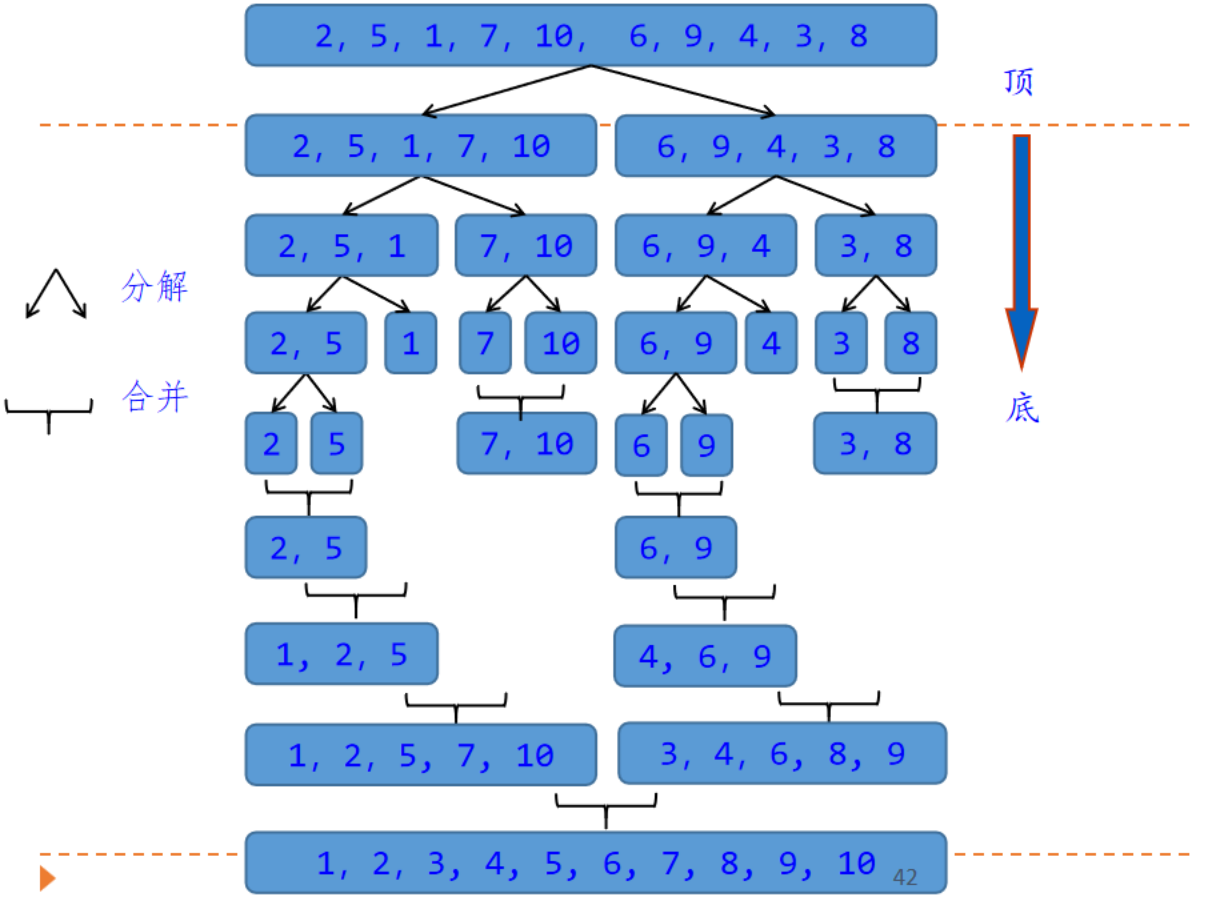
2 [3431] 归并排序

```
31  void MergeSort(int a[],int n)//二路归并算法
32  {  int length;
33     for (length=1;length<n;length=2*length)
34      MergePass(a,length,n);
35  }
36
37  int main(){
38      int n,m;
39      cin>>n;
40      for(int i=1;i<=n;i++){
41          cin>>m;
42          for(int j=0;j<m;j++)
43              cin>>a[j];
44          MergeSort(a,m);
45          for(int j=0;j<m;j++)
46              cout<<a[j]<<" ";
47          cout<<endl;
48      }
49      return 0;
50  }
```

```
9
4 3 9 5 7 6 8 1 2
第1次合并:3 4 9 5 7 6 8 1 2
第2次合并:3 4 5 9 7 6 8 1 2
第3次合并:3 4 5 9 6 7 8 1 2
第4次合并:3 4 5 9 6 7 1 8 2
第5次合并:3 4 5 9 6 7 1 8 2
第6次合并:3 4 5 9 6 7 1 8 2
第7次合并:3 4 5 9 1 6 7 8 2
第8次合并:1 3 4 5 6 7 8 9 2
第9次合并:1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9
```

2.2.1 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：

```
31  void MergeSort(int a[],int low,int high)
32  //二路归并算法
33  {   int mid;
34      if (low<high)               //子序列有两个或以上元素
35      {   mid=(low+high)/2;       //取中间位置
36          MergeSort(a,low,mid);   //对a[low..mid]子序列排序
37          MergeSort(a,mid+1,high);    //对a[mid+1..high]子序列排序
38          Merge(a,low,mid,high);  //将两子序列合并,见前面的算法
39      }
40  }
```

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：

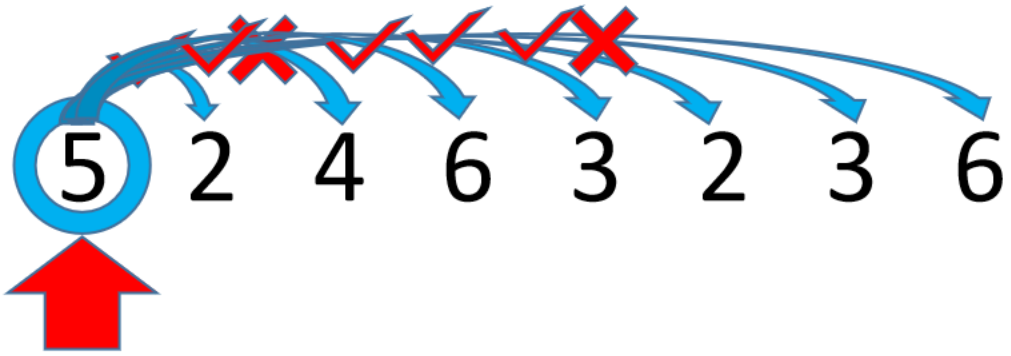
$$
\begin{cases}
T(n)=1 & \text{当 } n=1 \\
T(n)=2T(n/2)+O(n) & \text{当 } n>1
\end{cases}
$$

容易推出，$T(n)=O(n\log_2 n)$。

7005: 求逆序对个数

有一实数序列A[1]、A[2]、A[3]、……A[n-1]、A[n] (n<10000)，若i<j，并且A[i]>A[j]，则称A[i]与A[j]构成了一个逆序对，求数列A中逆序对的个数。
例如，5 2 4 6 2 3 2 6，可以组成的逆序对有
(5 2)，(5 4)，(5 2)，(5 3)，(5 2)，
(4 2)，(4 3)，(4 2)，
(6 2)，(6 3)，(6 2)，
(3 2)
共：12个
输入
共两行，第一行有一个正整数n，第二行有n个整数。
输出
只有一行为逆序对个数。
样例输入
8
5 2 4 6 2 3 2 6
样例输出
12

什么是逆序对



Sum: [redacted]++

再看一遍题目

有一实数序列A[1]、A[2]、A[3]、……A[n-1]、A[n]（n<10000），若i<j，并且A[i]>A[j]，则称A[i]与A[j]构成了一个逆序对，求数列A中逆序对的个数。

例如，5 2 4 6 2 3 2 6，可以组成的逆序对有

（5 2），（5 4），（5 2），（5 3），（5 2），
（4 2），（4 3），（4 2），
（6 2），（6 3），（6 2），
（3 2）

共：12个

TLE!

输入

共两行，第一行有一个正整数n，第二行有n个整数。

输出

只有一行为逆序对个数。

样例输入

8
5 2 4 6 2 3 2 6

样例输出

12

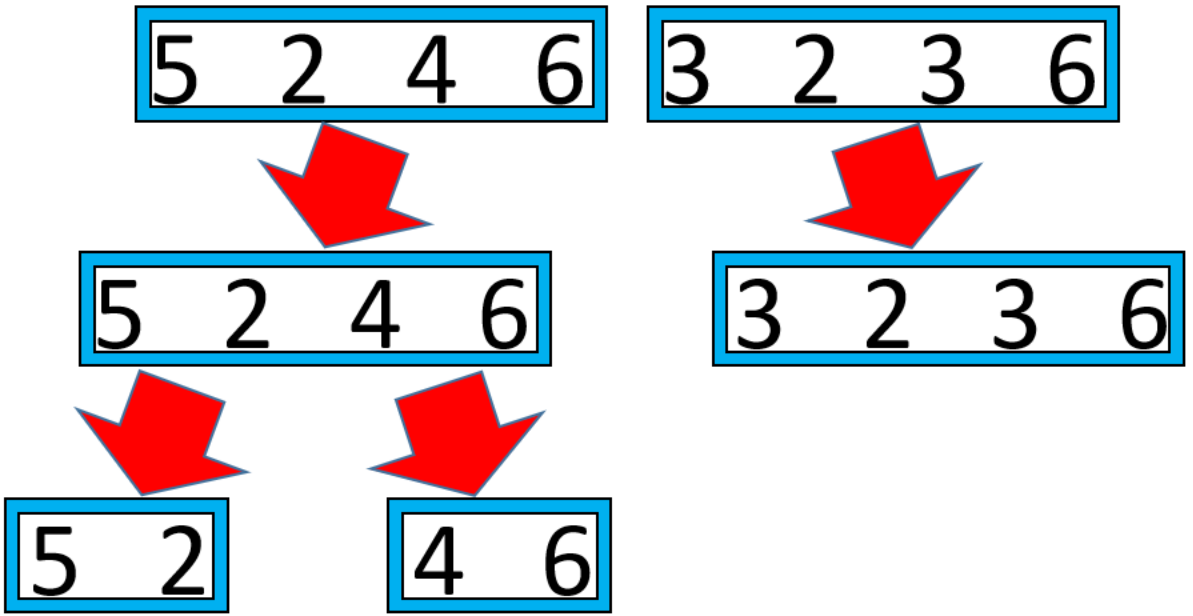
动态规划？

Dp[2]=1

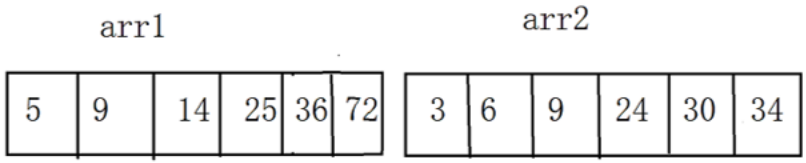
Dp[3]=2

Dp[4]=?

分而治之!

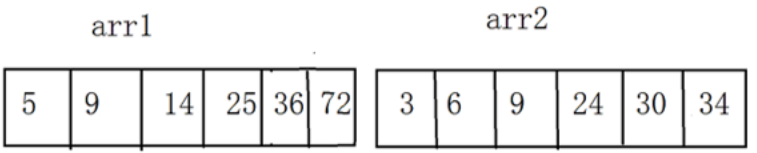


通过归并排序中的合并两个数组的操作，我们能够快速找出数组中所有的逆序对。具体做法如下：



假设前面的数组为arr1，后面的数组为arr2，按照归并排序的思想，这两个数组已经是有序的了。

同时遍历两个数组，对于遍历到的两个数，比较大小。如果arr1数组中的元素比arr2数组中的元素小，就把arr1中的当前元素放入到我们新生成的数组中去，arr1数组的指针往后移一下，反之亦然。



仔细想一下，我们在进行arr1和arr2数组比较的时候，肯定有一种情况是：arr2中的当前元素比arr1中的当前元素小，我们只需要在这种情况下做一点文章，就可以轻松找出所有的逆序对了。

每当我们遇到上述情况时，就计算出arr1数组中当前位置到最后位置的元素的个数，类加上，得到的累加和就是最后逆序对的个数。

上图，当arr1数组中的第一个位置5和arr2数组中的第一个位置3进行比较时，正好符合我们所说的那种情况，所以我们可以确定出至少有6个逆序对，分别是(5, 3) (9, 3) (14, 3) (25, 3) (36, 3) (72, 3)。

所以我们只需要在归并排序的代码上加上一句话，就可以实现求逆序对的算法了。

主函数代码

```
37  int main()
38  {
39      //input
40      int n;
41      cin>>n;
42      int A[n+1];
43      for(int i=0;i<n;i++)cin>>A[i];
44
45      //output
46      cout<<Merge(A,0,n)<<endl;
47
48      return 0;
49  }
50
```

合并函数代码(1)

```
 9  long long Merge(int A[],int lt,int rt)
10  {
11      //边界判断
12      if(lt>=rt-1)return 0;
13
14      //直接合并
15      int mid=(lt+rt)/2;
16      long long sum=Merge(A,lt,mid)+Merge(A,mid,rt);
```

合并函数代码(2)

```
18      //归并排序
19      int tmp[rt-lt+1];
20      int i=lt,j=mid,k=0;
21      while(i<mid&&j<rt)
22      {
23          if(A[i]<=A[j])
24          {
25              tmp[k++]=A[i++];
26              sum+=j-mid;
27          }
28          else
29              tmp[k++]=A[j++];
30      }
31      while(i<mid)
32      {
33          tmp[k++]=A[i++];
34          sum+=j-mid;
35      }
36      while(j<rt)
37          tmp[k++]=A[j++];
38      for(int l=0;l<k;l++)A[lt+l]=tmp[l];
39      //cout<<"merge["<<lt<<"]["<<rt<<"]:"<<sum<<endl;
40
41      return sum;
42  }
```

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。

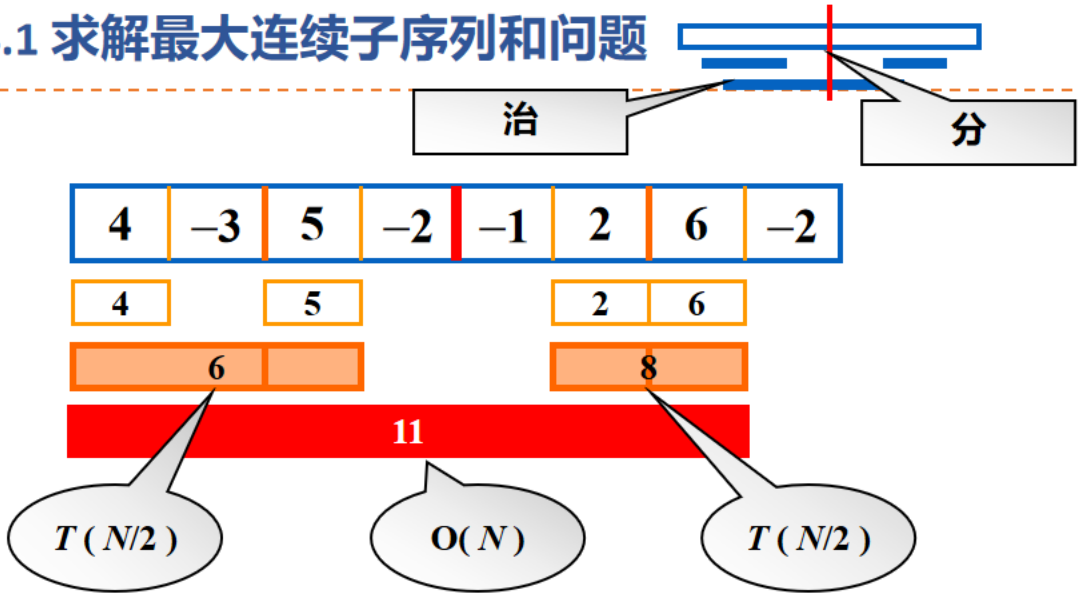
例如：

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。

3.4.1 求解最大连续子序列和问题



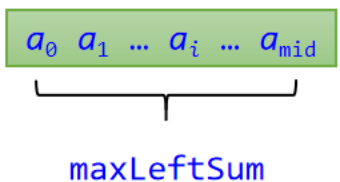
$$T(N) = 2\,T(N/2) + c\,N, \qquad T(1) = O(1)$$
$$= 2\,[2\,T(N/2^2) + c\,N/2] + c\,N$$
$$= 2^k\,O(1) + c\,k\,N \qquad \text{此处 } N=$$
$$= O(N \log N)$$

结论对$N \neq 2^k$同样正确

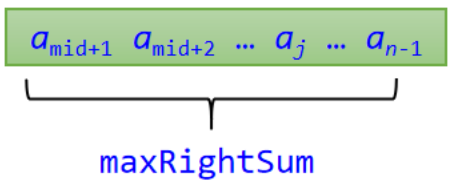
【问题求解】对于含有n个整数的序列$a[0..n-1]$，若$n=1$，表示该序列仅含一个元素，如果该元素大于0，则返回该元素；否则返回0。

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。

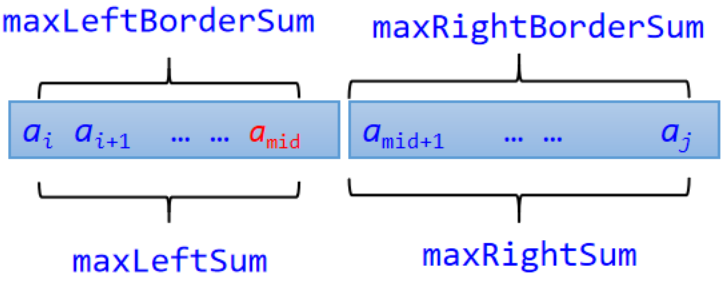
（1）该子序列完全落在左半部即$a[0..mid]$中。采用递归求出其最大连续子序列和maxLeftSum。



（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)

```
1  long maxSubSum(int a[],int left,int right)
2  //求a[left..high]序列中最大连续子序列和
3  {  int i,j;
4     long maxLeftSum,maxRightSum;
5     long maxLeftBorderSum,leftBorderSum;
6     long maxRightBorderSum,rightBorderSum;
7     if (left==right)      //子序列只有一个元素时
8     {     if (a[left]>0)  //该元素大于0时返回它
9               return a[left];
10          else            //该元素小于或等于0时返回0
11                return 0;
12    }
13     int mid=(left+right)/2;          //求中间位置
14     maxLeftSum=maxSubSum(a,left,mid);   //求左边
15     maxRightSum=maxSubSum(a,mid+1,right);   //求右边
```

```
16      maxLeftBorderSum=0,leftBorderSum=0;
17      for (i=mid;i>=left;i--)                  //求出以左边加上a[mid]元素
18      {  leftBorderSum+=a[i];                  //构成的序列的最大和
19         if (leftBorderSum>maxLeftBorderSum)
20              maxLeftBorderSum=leftBorderSum;
21      }
22      maxRightBorderSum=0,rightBorderSum=0;
23      for (j=mid+1;j<=right;j++)          //求出a[mid]右边元素
24      {  rightBorderSum+=a[j];            //构成的序列的最大和
25         if (rightBorderSum>maxRightBorderSum)
26              maxRightBorderSum=rightBorderSum;
27      }
28      return max3(maxLeftSum,maxRightSum,
29             maxLeftBorderSum+maxRightBorderSum);
30  }
```

【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，$T(n)=O(n\log_2 n)$。

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。

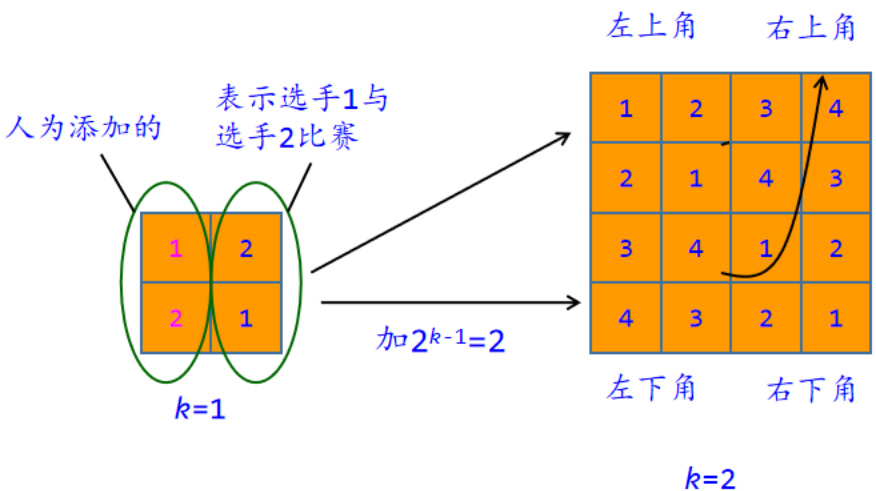
（2）每个选手一天只能赛一次。

（3）循环赛在$n-1$天之内结束。

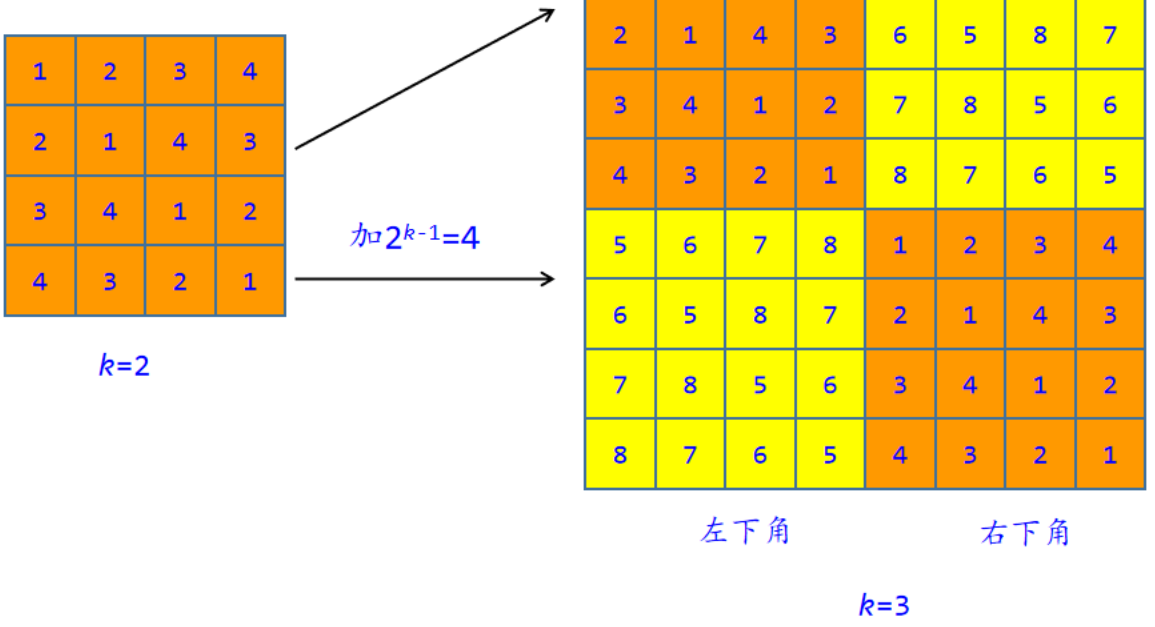
【问题求解】按问题要求可将比赛日程表设计成一个n行$n-1$列的二维表，其中第i行、第j列表示和第i个选手在第j天比赛的选手。

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
k=1
加2k-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=2

由$k=2$创建$k=3$的过程


左上角
右上角
1 2 3 4
2 1 4 3
3 4 1 2
4 3 2 1
加$2^{k-1}=4$
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
$k=2$
$k=3$

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。

```
#define MAX 101
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];
    }
}
```

[2792] 一元三次方程求解

有形如：$ax^3+bx^2+cx+d=0$这样的一个一元三次方程。给出该方程中各项的系数（a, b, c, d均为实数），并约定该方程存在三个不同实根（根的范围在-100至100之间），且根与根之差的绝对值≥1。

要求由小到大依次在同一行输出这三个实根（根与根之间留有空格），并精确到小数点后2位。

提示：记方程f（x）=0，若存在2个数x1和x2，且x1<x2，f（x1）*f（x2）＜0，则在（x1，x2）之间一定有一个根。

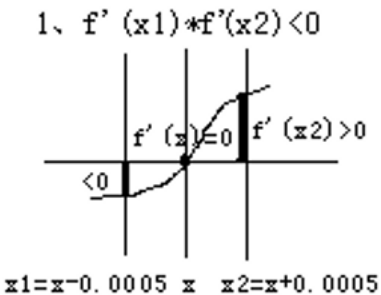
输入：a，b，c，d

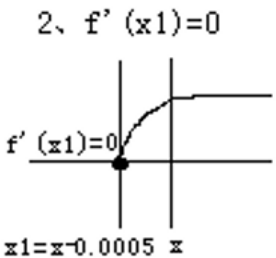
输出：三个实根（根与根之间留有空格）

【输入输出样例】

输入：1 -5 -4 20

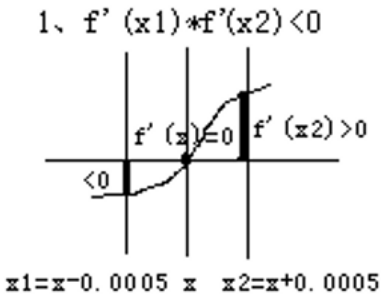
输出：-2.00 2.00 5.00

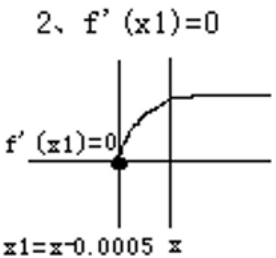




算法分析

这是一道有趣的解方程题。为了便于求解，设方程f(x)=ax3+bx2+cx+d＝0，设根的值域（-100至100之间）中有x， 其左右两边相距0.0005的地方有x1和x2两个数，即x1=x-0.0005，x2=x+0.0005。x1和x2间的距离（0.001）满足精度要求（精确到小数点后2位）。若出现如图1所示的两种情况之一，则确定x为f(x)=0的根。





有两种方法计算f(x)=0的根x：

1．枚举法

根据根的值域和根与根之间的间距要求(≥1)，我们不妨将根的值域扩大100倍(-10000≤x≤10000)，依次枚举该区间的每一个整数值x，并在题目要求的精度内设定区间：x1=　，x2=　。若区间端点的函数值f(x1)和f(x2)异号或者在区间端点x1的函数值f(x1)=0，则确定为f(x)=0的一个根。由此得出算法：

```
输入方程中各项的系数a，b，c，d ；
for (x=-10000;x<=10000;x++)
//枚举当前根*100的可能范围
 {
        x1=(x-0.05)/100;
        x2=(x+0.05)/100;
//在题目要求的精度内设定区间
if (f(x1)*f(x2)<0||f(x1)==0)
        printf(“%.2f”,x/100);
//若在区间两端的函数值异号
或在x1处的函数值为0，
则确定x/100为根

}
```

2. 分治法

枚举根的值域中的每一个整数x(-100≤x≤100)。由于根与根之差的绝对值≥1，因此设定搜索区间[x1，x2]，其中x1=x，x2=x+1。若

(1)f(x1)=0，则确定x1为f(x)的根；

(2)f(x1)*f(x2)>0，则确定根x不在区间[x1，x2]内，设定[x2，x2+1]为下一个搜索区间

(3)f(x1)*f(x2)<0，则确定根x在区间[x1，x2]内。

如果确定根x在区间[x1，x2]内的话（f(x1)*f(x2)<0），如何在该区间找到根的确切位置。采用二分法，将区间[x1，x2]分成左右两个子区间：左子区间[x1，x]和右子区间[x，x2]（其中x=(x1+x2)/2）：

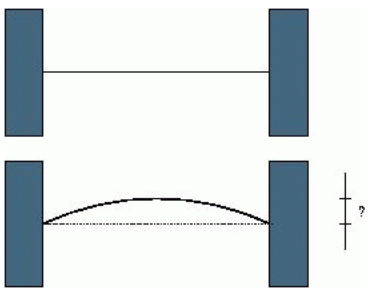
如果f(x1)*f(x)≤0，则确定根在左区间[x1，x]内，将x设为该区间的右指针（x2=x），继续对左区间进行对分；如果f(x1)*f(x)>0，则确定根在右区间[x，x2]内，将x设为该区间的左指针（x1=x），继续对右区间进行对分；

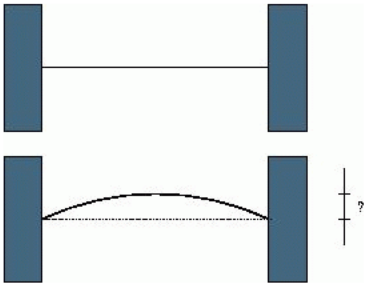
上述对分过程一直进行到区间的间距满足精度要求为止（x2-x1<0.001）。此时确定x1为f(x)的根。

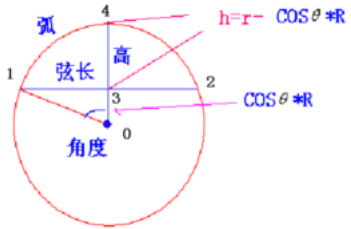
```cpp
#include<bits/stdc++.h>
using namespace std;
double a,b,c,d;
double f(double x){
    return a*x*x*x+b*x*x+c*x+d;
}
int main(){
    while(cin>>a>>b>>c>>d){
        for(double i=-100;i<=100;i++){////枚举每一个可能的根
            double x=i,y=i+1;////确定根的可能区间
            if(f(x)==0) //若x1为根，则输出
                printf("%.2lf ",x);
            else if(f(x)*f(y)<0){ //若根在区间[x，y]中
                while(y-x>=0.001){
                //若区间[x1，x2]不满足精度要求，则循环
                    double mid=(x+y)/2; //计算区间[x1，x2]的中间位置
                    if(f(x)*f(mid)<=0)
                        y=mid;//若根在左区间，则调整右指针
                    else x=mid;//若根在右区间，则调整左指针
                }
                printf("%.2lf ",x);//区间[x1，x2]满足精度要求，确定x为根
            }
        }
        printf("\n");
    }
    return 0;
}
```

[3226] 热膨胀

棒一根，加热膨胀成圆弧，求膨胀后的高度。
已知：棒的长度L，膨胀系数C，加热温度n.
L'=(1+n*C)*L







分析：

h=r-r*cos θ

*关键求 $*\theta$

l/2=r* θ (1式)

l/2=r*sin θ (2式)

1式/2式，得到 θ 的方程

l/l = θ / sin θ

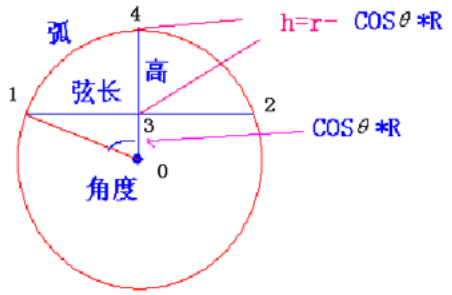
能解吗？怎么解？

解方程：插值法 二分法

$$ll/l = \theta / \sin\theta = t$$

$$\sin\theta = \theta * t$$

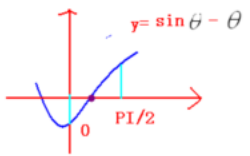
$$y = \sin\theta - t * \theta$$



对1/4圆弧来说，确定 θ的最大值、最小值

最大值：PI/2

最小值：0



```
#include <stdio.h>
#include <math.h>
int main()
{
 double l,c,n,ll,t,min,max,mid,y,h;
 while(scanf("%lf %lf %lf",&l,&n,&c)!=-1)
 {//l是弦长 ll是弧长 min,max,mid表示角度 h是高度
  if(l==-1&&n==-1&&c==-1)
   break;
  if(n*c==0)
   printf("%.3lf\n",0.0);
  else
  {
 }
 return 0;
}
```

```
12    else
13    {
14      ll=(1+c*n)*l;
15      t=l/ll;
16      min=0;
17      max=acos(-1)/2;//当是半圆时PI*r=ll; l=2r;l和ll的最大比值为PI/2
18
19      for(int i=0;i<100;i++)
20      {
21         mid=(min+max)/2;
22         y=sin(mid)-t*mid;//求角度，当角度很小的时候
23         if(y>0)
24          min=mid;
25         else
26          max=mid;
27      }
28      h=(1-cos(mid))*ll/2/mid;
29      /*  半径r; 圆弧长ll;  ll=2*角度mid*r;
30      h=r-rcos角度=(1-cos角度)× r;  r=ll/2/mid  */
31      printf("%.3lf\n",h);
32    }
33  }
```

2875 循环比赛日程表（match）

设有N个选手进行循环比赛，其中$N=2^M$，要求每名选手要与其他N-1名选手都赛一次，每名选手每天比赛一次，循环赛共进行N-1天，要求每天没有选手轮空。

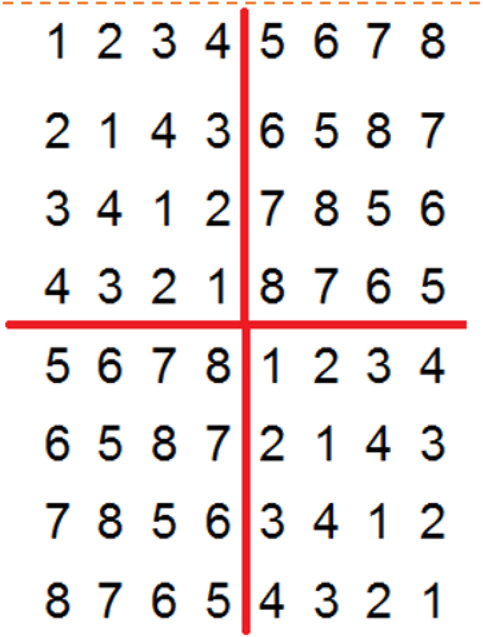
输入：M

输出：表格形式的比赛安排表

样例输入 3

样例输出

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



问题分析

以M=3(即N=2^3=8)为例，可以根据问题要求，制定出如下图所示的一种方案

	第一天	第二天	第三天	第四天	第五天	第六天	第七天
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

A　B　C　D

以表格的中心为拆分点，将表格分成A、B、C、D四个部分，就很容易看出有A=D，B=C，并且，这一规律同样适用于各个更小的部分。

问题分析

这是一个具有对称性的方阵

可以把方阵一分为四来看，那么左上角的4*4的方阵就是前四位选手的循环比赛表，而右上角的4*4的方阵就是后四位选手的循环比赛表，它们在本质上是一样的，都是4个选手的循环比赛表，所不同的只是选手编号不同而已，将左上角中方阵的所有元素加上4就能得到右上角的方阵.

下方的两个方阵表示前四位选手和后四位选手进行交叉循环比赛的情况，同样具有对称性，将右上角方阵复制到左下角即得到1，2，3，4四位选手和5，6，7，8四位选手的循环比赛表，根据对称性，右下角的方阵应与左上角的方阵相同. 这样，八名选手的循环比赛表可以由四名选手的循环比赛表根据对称性生成出来. 同样地，四名选手的循环比赛表可以由二名选手的循环比赛表根据对称性生成出来，而两名选手的循环比赛表可以说是已知的，。

程序中用数组matchlist记录n名选手的循环比赛表，整个循环比赛表从最初的1*1的方阵按上述规则生成出2*2 的方阵，再生成出4*4 的方阵，……，直到生成出整个循环比赛表为止. 变量half表示当前方阵的大小，也是要生成的下一个方阵的大小的一半 。

```
#include<cstdio>
const int MAXN=33,MAXM=5;
int matchlist[MAXN][MAXN];
int m;
int main()
{
        printf("Input m:");
        scanf("%d",&m);
        int n=1<<m,k=1,half=1;                    // 1<<m 相当于 2^m
        matchlist[0][0]=1;
```

```
while (k<=m)
{
        for (int i=0;i<half;i++)                //构造右上方方阵
                for (int j=0;j<half;j++)
                        matchlist[i][j+half]=matchlist[i][j]+half;
        for (int i=0;i<half;i++)                //对称交换构造下半部分方阵
                for (int j=0;j<half;j++)
                {
                        matchlist[i+half][j]=matchlist[i][j+half];
                          //左下方方阵等于右上方方阵
                        matchlist[i+half][j+half]=matchlist[i][j];
                          //右下方方阵等于左上方方阵
                }
```

```
            half*=2;
            k++;
        }
        for (int i=0;i<n;i++)
        {
            for (int j=0;j<n;j++)
                printf("%4d",matchlist[i][j]);
            putchar('\n');
        }
        return 0;
}
```

```cpp
const int MAXN=1000,MAXM=1000;
int matchlist[MAXN][MAXN];
int m;
int main()
{
    scanf("%d",&m);
    int n=1<<m,k=1,half=1;  // 1<<m 相当于 2^m
    matchlist[0][0]=1;
    while (k<=m)
    {
        for (int i=0;i<half;i++) //构造右上方方阵
            for (int j=0;j<half;j++)
                matchlist[i][j+half]=matchlist[i][j]+half;
        for (int i=0;i<half;i++)  //对称交换构造下半部分方阵
            for (int j=0;j<half;j++)
            {
                matchlist[i+half][j]=matchlist[i][j+half];
                //左下方方阵等于右上方方阵
                matchlist[i+half][j+half]=matchlist[i][j];
                //右下方方阵等于左上方方阵
            }
        half*=2;
        k++;
    }
    for (int i=0;i<n;i++)
        for (int j=0;j<n;j++)
            printf("%d ",matchlist[i][j]);
        putchar('\n');
    return 0;
}
```

取余运算（mod）

【问题描述】

输入b，p，k的值，求b^p mod k的值。其中b，p，k*k为长整形数。

【输入样例】mod.in

2 10 9

【输出样例】mod.out

2^10 mod 9=7

【算法分析】

本题主要的难点在于数据规模很大（b，p都是长整型数），对于b^p显然不能死算，那样的话时间复杂度和编程复杂度都很大。

下面先介绍一个原理：A*B%K = (A%K)*(B% K)%K。显然有了这个原理，就可以把较大的幂分解成较小的，因而免去高精度计算等复杂过程。那么怎样分解最有效呢？

- 显然对于任何一个自然数P，有P=2 * P/2 + P%2，如19=2 * 19/2 + 19%2=2*9+1，利用上述原理就可以把B的19次方除以K的余数转换为求B的9次方除以K的余数，即$B^{19}=B^{2*9+1}=B*B^{9}*B^{9}$，再进一步分解下去就不难求得整个问题的解。

【参考程序】

```
#include<iostream>
#include<cstdio>
using namespace std;
int b,p,k,a;
int f(int p)                          //利用分治求b^p % k
{
        if (p==0) return 1;                 // b^0 %k=1
        int tmp=f(p/2)%k;
        tmp=(tmp*tmp) % k;                  // b^p %k=(b^(p/2))^2 % k
        if (p%2==1) tmp=(tmp * b) %k;  //如果p为奇数，则 b^p %
    return tmp;                         //k=((b^(p/2))^2)* b%k

}
int main()
{
   cin>>b>>p>>k;                //读入3个数
   int tmpb=b;                  //将b的值备份
   b%=k;                        //防止b太大
        printf("%d^%d mod %d=%d\n",tmpb,p,k,f(p)); //输出
        return 0;
}
```

【例8】、黑白棋子的移动（chessman）

【问题描述】

有2n个棋子（n≥4）排成一行，开始位置为白子全部在左边，黑子全部在右边，如下图为n=5的情形：

○○○○○●●●●●

移动棋子的规则是：每次必须同时移动相邻的两个棋子，颜色不限，可以左移也可以右移到空位上去，但不能调换两个棋子的左右位置。每次移动必须跳过若干个棋子（不能平移），要求最后能移成黑白相间的一行棋子。如n=5时，成为：

○●○●○●○●○●

任务：编程打印出移动过程。

【输入样例】

7

【输出样例】

step 0:ooooooo*******--

step 1:oooooo--******o*

step 2:oooooo******--o*

step 3:ooooo--*****o*o*

step 4:ooooo*****--o*o*

step 5:oooo--****o*o*o*

step 6:oooo****--o*o*o*

step 7:ooo--***o*o*o*o*

step 8:ooo*o**--*o*o*o*

step 9:o--*o**oo*o*o*o*

step10:o*o*o*--o*o*o*o*

step11:--o*o*o*o*o*o*o*

【算法分析】

我们先从n=4开始试试看，初始时：

○○○○●●●●

第1步：○○○——●●●○● {—表示空位}

第2步：○○○●○●●——●

第3步：○——●○●●○○●

第4步：○●○●○●——○●

第5步：——○●○●○●○●

如果n=5呢？我们继续尝试，希望看出一些规律，初始时：

○○○○○●●●●●

第1步：○○○○——●●●●○●

第2步：○○○○●●●●——○●

这样，n=5的问题又分解成了n=4的情况，下面只要再做一下n=4的5个步骤就行了。同理，n=6的情况又可以分解成n=5的情况，……，所以，对于一个规模为n的问题，我们很容易地就把他分治成了规模为n-1的相同类型子问题。

数据结构如下：数组c[1..max]用来作为棋子移动的场所，初始时，c[1]~c[n]存放白子（用字符o表示），c[n+1]~c[2n]存放黑子（用字符*表示），c[2n+1]，c[2n+2]为空位置（用字符—表示）。最后结果在c[3]~c[2n+2]中。

【参考程序】

```
#include<iostream>
using namespace std;
int n,st,sp;
char c[101];
void print()                    //打印
{
        int i;
        cout<<"step "<<st<<':';
        for (i=1;i<=2*n+2;i++) cout<<c[i];
        cout<<endl;
        st++;
}
void init(int n)                //初始化
{
        int i;
        st=0;
        sp=2*n+1;
        for (i=1;i<=n;i++) c[i]='o';
        for (i=n+1;i<=2*n;i++) c[i]='*';
        c[2*n+1]='-';c[2*n+2]='-';
        print();
}
```

```
void move(int k)                    //移动一步
{
        int j;
        for (j=0;j<=1;j++)
        {
          c[sp+j]=c[k+j];
          c[k+j]='-';
        }
        sp=k;
        print();
}
void mv(int n)                      //主要过程
{
        int i,k;
        if (n==4)                       //n等于4的情况要特殊处理
        {
                move(4); move(8); move(2); move(7); move(1);
                }
        else
        {
                move(n); move(2*n-1); mv(n-1);
                }
}
int main()
{
        cin>>n;
        init(n);
        mv(n);
}
```

请大家发挥想象！

今天的课程结束啦……

下课了…
同学们再见！