**Java实现常用排序**

1.比较排序：

**package** com.select.sort;

**public** **class** SelectSort

{

**public** **static** **void** main(String[] args)

{

**int**[] data = **new** **int**[]{10, 9, 8, 7, 6, 5, 4, 3, 2, 1};

System.*out*.println("before the sort:");

*print*(data);

*sortBySelect*(data);

}

**public** **static** **void** sortBySelect(**int**[] data)

{

**int** minIndex = 0;

**for**(**int** i = 0; i < data.length - 1; i++)

{

minIndex = i;

**for**(**int** j = i+1; j < data.length; j++)

{

**if**(data[minIndex] > data[j])

{

minIndex = j;

}

}

**if**(minIndex != i)

{

*swap*(data, i, minIndex);

*print*(data);

}

}

}

**public** **static** **void** swap(**int**[] data, **int** i, **int** j)

{

data[i] = data[i] + data[j];

data[j] = data[i] - data[j];

data[i] = data[i] - data[j];

}

**public** **static** **void** print(**int**[] data)

{

**for**(**int** number : data)

{

System.*out*.print(number + " ");

}

System.*out*.println();

}

}

2.冒泡排序：

**package** bubble;

**public** **class** Bubble

{

**public** **static** **void** main(String[] args)

{

**int**[] data = **new** **int**[]{10, 9, 8, 7, 6, 5, 4, 3, 2, 1};

System.*out*.println("before sort");

*print*(data);

*BubbleSort*(data);

}

**public** **static** **void** print(**int**[] data)

{

**for**(**int** number : data)

{

System.*out*.print(number + " ");

}

System.*out*.println();

}

**public** **static** **void** swap(**int**[] data, **int** i, **int** j)

{

data[i] = data[i] + data[j];

data[j] = data[i] - data[j];

data[i] = data[i] - data[j];

}

**public** **static** **void** BubbleSort(**int**[] data)

{

**for**(**int** i = 0; i < data.length; i++)

{

**for**(**int** j =0; j < data.length - i -1; j++)

{

**if**(data[j] > data[j+1])

{

*swap*(data, j, j+1);

}

}

*print*(data);

}

}

}

3.快速排序

**public class FastSort**

**{**

**public static void main(String[] args)**

**{**

**int[] data = new int[]{10, 9, 8, 7, 12, 5, 4, 3, 2, 1};**

**System.out.println("before sort");**

**print(data);**

**quickSort(data, 0, data.length - 1);**

**}**

**public static void quickSort(int[] data, int begin, int end)**

**{**

**if(begin >= end)**

**{**

**return;**

**}**

**int pivot = data[begin];**

**int i = begin + 1;**

**int j = end;**

**while(true)**

**{**

**while(data[i] < pivot && i < end)**

**{**

**i++;**

**}**

**while(data[j] > pivot && j > begin)**

**{**

**j--;**

**}**

**if(i < j)**

**{**

**swap(data, i, j);**

**print(data);**

**}**

**else**

**{**

**print(data);**

**break;**

**}**

**}**

**swap(data, begin, j);**

**print(data);**

**quickSort(data, begin, j-1);**

**quickSort(data, j+1, end);**

**}**

**public static void print(int[] data)**

**{**

**for(int number : data)**

**{**

**System.out.print(number + " ");**

**}**

**System.out.println();**

**}**

**public static void swap(int[] data, int i, int j)**

**{**

**if(i == j)**

**{**

**return;**

**}**

**data[i] = data[i] + data[j];**

**data[j] = data[i] - data[j];**

**data[i] = data[i] - data[j];**

**}**

**}**

直接插入排序

直接插入排序的基本操作就是将待排序的数据元素按其关键字值的大小插入到前面的有序序列中。

直接插入的时间效率并不高，如果在最坏的情况下，所有元素的比较次数总和为（0+1+...+n-1）=O(n^2)。其他情况下也要考虑移动元素的次数，故时间复杂度为O(n^2)

直接插入空间效率很好，只需要1个缓存数据单元，也就是说空间复杂度为O(1). 直接插入排序是稳定的。

直接插入排序在数据已有一定顺序的情况下，效率较好。但如果数据无规则，则需要移动大量的数据，其效率就与冒泡排序法和选择排序法一样差了。

算法描述

对一个有n个元素的数据序列，排序需要进行n-1趟插入操作：

第1趟插入，将第2个元素插入前面的有序子序列--此时前面只有一个元素，当然是有序的。

第2趟插入，将第3个元素插入前面的有序子序列，前面2个元素是有序的。 第n-1趟插入，将第n个元素插入前面的有序子序列，前面n-1个元素是有序的。

代码实现

view plain

1. package sort; 2.

3. public class InsertSortTest { 4. public static int count = 0; 5.

6. public static void main(String[] args) { 7.

8. int[] data = new int[] { 5, 3, 6, 2, 1, 9, 4, 8, 7 }; 9. print(data); 10. insertSort(data); 11. print(data); 12.  13. } 14.

15. public static void insertSort(int[] data) { 16. for (int i = 1; i < data.length; i++) { 17. // 缓存i处的元素值 18. int tmp = data[i];

19. if (data[i] < data[i - 1]) { 20. int j = i - 1; 21. // 整体后移一格

22. while (j >= 0 && data[j] > tmp) { 23. data[j + 1] = data[j]; 24. j--; 25. }

26. // 最后将tmp插入合适的位置 27. data[j + 1] = tmp; 28. print(data); 29. } 30. } 31. } 32.

33. public static void print(int[] data) { 34. for (int i = 0; i < data.length; i++) { 35. System.out.print(data[i] + "\t"); 36. }

37. System.out.println(); 38. } 39.  40. }

运行结果：

view plain

1. 5 3 6 2 1 9 4 8 7 2. 3 5 6 2 1 9 4 8 7 3. 2 3 5 6 1 9 4 8 7 4. 1 2 3 5 6 9 4 8 7 5. 1 2 3 4 5 6 9 8 7 6. 1 2 3 4 5 6 8 9 7 7. 1 2 3 4 5 6 7 8 9 8. 1 2 3 4 5 6 7 8 9

排序算法有很多，所以在特定情景中使用哪一种算法很重要。为了选择合适的算法，可以按照建议的顺序考虑以下标准：   
（1）执行时间   
（2）存储空间   
（3）[编程](http://www.2cto.com/kf)工作   
   对于数据量较小的情形，（1）（2）差别不大，主要考虑（3）；而对于数据量大的，（1）为首要。   
    
主要排序法有：   
一、冒泡（Bubble）排序——相邻交换   
二、选择排序——每次最小/大排在相应的位置   
三、插入排序——将下一个插入已排好的序列中   
四、壳（Shell）排序——缩小增量   
五、归并排序   
六、快速排序   
七、堆排序   
八、拓扑排序   
九、锦标赛排序   
十、基数排序   
    
    
   
一、冒泡（Bubble）排序   
   
----------------------------------Code 从小到大排序n个数------------------------------------   
void BubbleSortArray()   
{   
      for(int i=1;i<n;i++)   
      {   
        for(int j=0;i<n-i;j++)   
         {   
              if(a[j]>a[j+1])//比较交换相邻元素   
               {   
                   int temp;   
                   temp=a[j]; a[j]=a[j+1]; a[j+1]=temp;   
               }   
         }   
      }   
}   
-------------------------------------------------Code------------------------------------------------   
效率 O（n²）,适用于排序小列表。   
    
    
二、选择排序   
----------------------------------Code 从小到大排序n个数--------------------------------   
void SelectSortArray()   
{   
    int min\_index;   
    for(int i=0;i<n-1;i++)   
    {   
         min\_index=i;   
         for(int j=i+1;j<n;j++)//每次扫描选择最小项   
            if(arr[j]<arr[min\_index])  min\_index=j;   
         if(min\_index!=i)//找到最小项交换，即将这一项移到列表中的正确位置   
         {   
             int temp;   
             temp=arr[i]; arr[i]=arr[min\_index]; arr[min\_index]=temp;   
}   
}   
}   
-------------------------------------------------Code-----------------------------------------   
效率O（n²），适用于排序小的列表。   
    
    
三、插入排序   
--------------------------------------------Code 从小到大排序n个数-------------------------------------   
void InsertSortArray()   
{   
for(int i=1;i<n;i++)//循环从第二个数组元素开始，因为arr[0]作为最初已排序部分   
{   
    int temp=arr[i];//temp标记为未排序第一个元素   
    int j=i-1;   
while (j>=0 && arr[j]>temp)/\*将temp与已排序元素从小到大比较，寻找temp应插入的位置\*/   
{   
    arr[j+1]=arr[j];   
    j--;   
}   
arr[j+1]=temp;   
}   
}   
------------------------------Code--------------------------------------------------------------   
最佳效率O（n）；最糟效率O（n²）与冒泡、选择相同，适用于排序小列表   
若列表基本有序，则插入排序比冒泡、选择更有效率。   
    
    
四、壳（Shell）排序——缩小增量排序   
-------------------------------------Code 从小到大排序n个数-------------------------------------   
void ShellSortArray()   
{   
  for(int incr=3;incr<0;incr--)//增量递减，以增量3，2，1为例   
{   
       for(int L=0;L<(n-1)/incr;L++)//重复分成的每个子列表   
{   
   for(int i=L+incr;i<n;i+=incr)//对每个子列表应用插入排序   
   {   
      int temp=arr[i];   
      int j=i-incr;   
      while(j>=0&&arr[j]>temp)   
      {   
          arr[j+incr]=arr[j];   
          j-=incr;   
}   
arr[j+incr]=temp;   
}   
}   
}   
}   
--------------------------------------Code-------------------------------------------   
适用于排序小列表。   
效率估计O（nlog2^n）~O（n^1.5），取决于增量值的最初大小。建议使用质数作为增量值，因为如果增量值是2的幂，则在下一个通道中会再次比较相同的元素。   
壳（Shell）排序改进了插入排序，减少了比较的次数。是不稳定的排序，因为排序过程中元素可能会前后跳跃。   
    
    
五、归并排序   
----------------------------------------------Code 从小到大排序---------------------------------------   
void MergeSort(int low,int high)   
{   
   if(low>=high)   return;//每个子列表中剩下一个元素时停止   
   else int mid=(low+high)/2;/\*将列表划分成相等的两个子列表,若有奇数个元素，则在左边子列表大于右侧子列表\*/   
   MergeSort(low,mid);//子列表进一步划分   
   MergeSort(mid+1,high);   
   int [] B=new int [high-low+1];//新建一个数组，用于存放归并的元素   
   for(int i=low,j=mid+1,k=low;i<=mid && j<=high;k++)/\*两个子列表进行排序归并，直到两个子列表中的一个结束\*/   
   {   
       if (arr[i]<=arr[j];)   
{   
    B[k]=arr[i];   
    I++;   
}   
else  
    { B[k]=arr[j]; j++; }   
}   
for(   ;j<=high;j++,k++)//如果第二个子列表中仍然有元素，则追加到新列表   
      B[k]=arr[j];   
   for(   ;i<=mid;i++,k++)//如果在第一个子列表中仍然有元素，则追加到新列表中   
      B[k]=arr[i];   
   for(int z=0;z<high-low+1;z++)//将排序的数组B的 所有元素复制到原始数组arr中   
      arr[z]=B[z];   
}   
-----------------------------------------------------Code---------------------------------------------------   
效率O（nlogn），归并的最佳、平均和最糟用例效率之间没有差异。   
适用于排序大列表，基于分治法。   
    
六、快速排序   
------------------------------------Code--------------------------------------------   
/\*快速排序的算法思想：选定一个枢纽元素，对待排序序列进行分割，分割之后的序列一个部分小于枢纽元素，一个部分大于枢纽元素，再对这两个分割好的子序列进行上述的过程。\*/                  void swap(int a,int b){int t;t =a ;a =b ;b =t ;}   
        int Partition(int [] arr,int low,int high)   
        {   
            int pivot=arr[low];//采用子序列的第一个元素作为枢纽元素   
            while (low < high)   
            {   
                //从后往前栽后半部分中寻找第一个小于枢纽元素的元素   
                while (low < high && arr[high] >= pivot)   
                {   
                    --high;   
                }   
                //将这个比枢纽元素小的元素交换到前半部分   
                swap(arr[low], arr[high]);   
                //从前往后在前半部分中寻找第一个大于枢纽元素的元素   
                while (low <high &&arr [low ]<=pivot )   
                {   
                    ++low ;   
                }   
                swap (arr [low ],arr [high ]);//将这个枢纽元素大的元素交换到后半部分   
            }   
            return low ;//返回枢纽元素所在的位置   
        }   
        void QuickSort(int [] a,int low,int high)   
        {   
            if (low <high )   
            {   
                int n=Partition (a ,low ,high );   
                QuickSort (a ,low ,n );   
                QuickSort (a ,n +1,high );   
            }   
        }   
----------------------------------------Code-------------------------------------   
平均效率O（nlogn），适用于排序大列表。   
此算法的总时间取决于枢纽值的位置；选择第一个元素作为枢纽，可能导致O（n²）的最糟用例效率。若数基本有序，效率反而最差。选项中间值作为枢纽，效率是O（nlogn）。   
基于分治法。   
    
    
   
七、堆排序   
最大堆：后者任一非终端节点的关键字均大于或等于它的左、右孩子的关键字，此时位于堆顶的节点的关键字是整个序列中最大的。   
思想：   
(1)令i=l,并令temp＝ kl ;   
(2)计算i的左孩子j=2i+1;   
(3)若j<＝n－1，则转(4),否则转(6);   
(4)比较kj和kj+1,若kj+1>kj,则令j＝j＋1，否则j不变；   
(5)比较temp和kj，若kj>temp，则令ki等于kj，并令i=j,j=2i+1,并转(3),否则转(6)   
(6)令ki等于temp，结束。   
-----------------------------------------Code---------------------------   
void HeapSort(SeqIAst R)   
   
    { //对R[1..n]进行堆排序，不妨用R[0]做暂存单元    int I;    BuildHeap(R)； //将R[1-n]建成初始堆for(i=n;i>1；i--) //对当前无序区R[1..i]进行堆排序，共做n-1趟。{      R[0]=R[1]; R[1]=R[i]; R[i]=R[0]; //将堆顶和堆中最后一个记录交换      Heapify(R,1,i-1);  //将R[1..i-1]重新调整为堆，仅有R[1]可能违反堆性质     }    } ---------------------------------------Code--------------------------------------   
   
    
堆排序的时间，主要由建立初始堆和反复重建堆这两部分的时间开销构成，它们均是通过调用Heapify实现的。   
   
      堆排序的最坏时间复杂度为O(nlgn)。堆排序的平均性能较接近于最坏性能。     由于建初始堆所需的比较次数较多，所以堆排序不适宜于记录数较少的文件。     堆排序是就地排序，辅助空间为O(1)，     它是不稳定的排序方法。   
   
    
堆排序与直接插入排序的区别:   
     直接选择排序中，为了从R[1..n]中选出关键字最小的记录，必须进行n-1次比较，然后在R[2..n]中选出关键字最小的记录，又需要做n-2次比较。事实上，后面的n-2次比较中，有许多比较可能在前面的n-1次比较中已经做过，但由于前一趟排序时未保留这些比较结果，所以后一趟排序时又重复执行了这些比较操作。   
     堆排序可通过树形结构保存部分比较结果，可减少比较次数。   
    
   
八、拓扑排序   
例 ：学生选修课排课先后顺序   
拓扑排序：把有向图中各顶点按照它们相互之间的优先关系排列成一个线性序列的过程。   
方法：   
在有向图中选一个没有前驱的顶点且输出   
从图中删除该顶点和所有以它为尾的弧   
重复上述两步，直至全部顶点均已输出（拓扑排序成功），或者当图中不存在无前驱的顶点（图中有回路）为止。   
---------------------------------------Code--------------------------------------   
void TopologicalSort()/\*输出拓扑排序函数。若G无回路，则输出G的顶点的一个拓扑序列并返回OK，否则返回ERROR\*/   
{   
      int indegree[M];   
      int i,k,j;   
      char n;   
      int count=0;   
      Stack thestack;   
      FindInDegree(G,indegree);//对各顶点求入度indegree[0....num]   
      InitStack(thestack);//初始化栈   
      for(i=0;i<G.num;i++)   
          Console.WriteLine("结点"+G.vertices[i].data+"的入度为"+indegree[i]);   
      for(i=0;i<G.num;i++)   
      {   
           if(indegree[i]==0)   
              Push(thestack.vertices[i]);   
      }   
      Console.Write("拓扑排序输出顺序为：");   
      while(thestack.Peek()!=null)   
      {   
               Pop(thestack.Peek());   
               j=locatevex(G,n);   
               if (j==-2)   
                  {   
                         Console.WriteLine("发生错误，程序结束。");   
                         exit();   
                  }   
                Console.Write(G.vertices[j].data);   
                count++;   
                for(p=G.vertices[j].firstarc;p!=NULL;p=p.nextarc)   
                {   
                     k=p.adjvex;   
                     if (!(--indegree[k]))   
                         Push(G.vertices[k]);   
                }   
      }   
      if (count<G.num)   
          Cosole.WriteLine("该图有环，出现错误，无法排序。");   
      else  
          Console.WriteLine("排序成功。");   
}   
----------------------------------------Code--------------------------------------   
算法的时间复杂度O（n+e）。   
    
    
   
九、锦标赛排序   
锦标赛排序的算法思想与体育比赛类似。   
    首先将n个数据元素两两分组，分别按关键字进行比较，得到n／2个比较的优胜者(关键字小者)，作为第一步比较的结果保留下来，   
    然后对这n／2个数据元素再两两分组，分别按关键字进行比较，…，如此重复，直到选出一个关键字最小的数据元素为止。   
   
   
    
--------------------------------Code in C---------------------------------------   
#include <stdio.h>   
#include <stdlib.h>   
#include <string.h>   
#include <math.h>   
#define SIZE 100000   
#define MAX 1000000   
struct node   
{   
 long num;//关键字   
 char str[10];   
 int lastwin;//最后胜的对手   
 int killer;//被击败的对手   
 long times;//比赛次数   
}data[SIZE];   
long CompareNum=0;   
long ExchangeNum=0;   
long Read(char name[])//读取文件a.txt中的数据，并存放在数组data[]中；最后返回数据的个数   
{   
 FILE \*fp;   
 long i=1;   
 fp=fopen(name,"rw");   
 fscanf(fp,"%d%s",&data[i].num,data[i].str);   
 while(!feof(fp))   
 {   
  i++;   
  fscanf(fp,"%d%s",&data[i].num,data[i].str);    
 }   
 return (i-1);   
}   
long Create(long num)//创建胜者树，返回冠军（最小数）在数组data[]中的下标   
{   
 int i,j1,j2,max,time=1;   
 long min;//记录当前冠军的下标   
 for(i=1;pow(2,i-1)<num;i++)   
  ;   
 max=pow(2,i-1);//求叶子结点数目   
 for(i=1;i<=max;i++)//初始化叶子结点   
 {   
  data[i].killer=0;   
  data[i].lastwin=0;   
  data[i].times=0;   
  if(i>num)   
   data[i].num=MAX;   
 }   
 for(i=1;i<=max;i+=2)//第一轮比赛   
 {   
  ++CompareNum;   
  if(data[i].num <= data[i+1].num)   
  {   
   data[i].lastwin = i+1;   
   data[i+1].killer=i;   
   ++data[i].times;   
   ++data[i+1].times;   
   min=i;   
  }   
  else  
  {   
   data[i+1].lastwin=i;   
   data[i].killer=i+1;   
   ++data[i].times;   
   ++data[i+1].times;   
   min=i+1;   
  }   
 }   
 j1=j2=0;//记录连续的两个未被淘汰的选手的下标   
 while(time <= (log(max)/log(2)))//进行淘汰赛   
 {   
  for(i=1;i<=max;i++)   
  {   
   if(data[i].times==time && data[i].killer==0)//找到一名选手   
   {   
    j2=i;//默认其为两选手中的后来的   
    if(j1==0)//如果第一位置是空的，则刚来的选手先来的   
     j1=j2;   
    else//否则刚来的选手是后来的，那么选手都已到场比赛开始   
    {   
     ++CompareNum;   
     if(data[j1].num <= data[j2].num)//先来的选手获胜   
     {   
      data[j1].lastwin = j2;//最后赢的是j2   
      data[j2].killer=j1;//j2是被j1淘汰的   
      ++data[j1].times;   
      ++data[j2].times;//两选手场次均加1    
      min=j1;//最小数下标为j1   
      j1=j2=0;//将j1，j2置0   
     }   
     else//同理   
     {   
      data[j2].lastwin=j1;   
      data[j1].killer=j2;   
      ++data[j1].times;   
      ++data[j2].times;        
      min=j2;   
      j1=j2=0;   
     }   
    }   
   }   
     
  }   
  time++;//轮数加1   
 }   
 return min;//返回冠军的下标   
}   
void TournamentSort(long num)//锦标赛排序   
{   
 long tag=Create(num);//返回最小数下标   
 FILE \*fp1;   
 fp1=fopen("sort.txt","w+");//为写入创建并打开文件sort.txt   
 while(data[tag].num != MAX)//当最小值不是无穷大时   
 {   
  printf("%d %s\n",data[tag].num,data[tag].str);//输出数据   
  fprintf(fp1,"%d %s\n",data[tag].num,data[tag].str);//写入数据   
  data[tag].num=MAX;//将当前冠军用无穷大替换   
  tag=Create(num);//返回下一个冠军的下标    
 }   
}   
int main()   
{   
 int num;   
 char name[10];   
 printf("Input name of the file:");   
 gets(name);   
 num=Read(name);//读文件   
 TournamentSort(num);//锦标赛排序   
 printf("CompareNum=%d\nExchangeNum=%d\n",CompareNum,ExchangeNum);   
 return 0;   
}   
------------------------------------------Code-------------------------------------   
    
    
十、基数排序   
基数排序又被称为桶排序。与前面介绍的几种排序方法相比较，基数排序和它们有明显的不同。   
    前面所介绍的排序方法都是建立在对数据元素关键字进行比较的基础上，所以可以称为基于比较的排序；   
    而基数排序首先将待排序数据元素依次“分配”到不同的桶里，然后再把各桶中的数据元素“收集”到一起。   
通过使用对多关键字进行排序的这种“分配”和“收集”的方法，基数排序实现了对多关键字进行排序。   
———————————————————————————————————————   
例：   
    每张扑克牌有两个“关键字”：花色和面值。其大小顺序为：   
    花色：§＜¨＜©＜ª   
    面值：2＜3＜……＜K＜A   
    扑克牌的大小先根据花色比较，花色大的牌比花色小的牌大；花色一样的牌再根据面值比较大小。所以，将扑克牌按从小到大的次序排列，可得到以下序列：   
 §2，…，§A，¨2，…，¨A，©2，…，©A，ª2，…，ªA   
    这种排序相当于有两个关键字的排序，一般有两种方法实现。   
    其一：可以先按花色分成四堆（每一堆牌具有相同的花色），然后在每一堆牌里再按面值从小到大的次序排序，最后把已排好序的四堆牌按花色从小到大次序叠放在一起就得到排序的结果。   
其二：可以先按面值排序分成十三堆（每一堆牌具有相同的面值），然后将这十三堆牌按面值从小到大的顺序叠放在一起，再把整副牌按顺序根据花色再分成四堆（每一堆牌已按面值从小到大的顺序有序），最后将这四堆牌按花色从小到大合在一起就得到排序的结果。   
———————————————————————————————————————   
实现方法：   
　　最高位优先(Most Significant Digit first)法，简称MSD法：先按k1排序分组，同一组中记录，关键码k1相等，再对各组按k2排序分成子组，之后，对后面的关键码继续这样的排序分组，直到按最次位关键码kd对各子组排序后。再将各组连接起来，便得到一个有序序列。   
　　最低位优先(Least Significant Digit first)法，简称LSD法：先从kd开始排序，再对kd-1进行排序，依次重复，直到对k1排序后便得到一个有序序列。   
---------------------------------Code in C#------------------------------------------   
　　using System;   
　　using System.Collections.Generic;   
　　using System.Linq;   
　　using System.Text;   
　　namespace LearnSort   
　　{   
　　class Program   
　　{   
　　static void Main(string[] args)   
　　{   
　　int[] arr = CreateRandomArray(10);//产生随机数组   
　　Print(arr);//输出数组   
　　RadixSort(ref arr);//排序   
　　Print(arr);//输出排序后的结果   
　　Console.ReadKey();   
　　}   
　　public static void RadixSort(ref int[] arr)   
　　{   
　　int iMaxLength = GetMaxLength(arr);   
　　RadixSort(ref arr, iMaxLength);   
　　}   
　　private static void RadixSort(ref int[] arr, int iMaxLength)   
　　{   
　　List<int> list = new List<int>();//存放每次排序后的元素   
　　List<int>[] listArr = new List<int>[10];//十个桶   
　　char currnetChar;//存放当前的字符比如说某个元素123 中的2   
　　string currentItem;//存放当前的元素比如说某个元素123   
　　for (int i = 0; i < listArr.Length; i++)//给十个桶分配内存初始化。   
　　listArr[i] = new List<int>();   
　　for (int i = 0; i < iMaxLength; i++)//一共执行iMaxLength次，iMaxLength是元素的最大位数。   
　　{   
　　foreach (int number in arr)//分桶   
　　{   
　　currentItem = number.ToString();//将当前元素转化成字符串   
　　try { currnetChar = currentItem[currentItem.Length-i-1]; }//从个位向高位开始分桶   
　　catch { listArr[0].Add(number); continue; }//如果发生异常，则将该数压入listArr[0]。比如说5 是没有十位数的，执行上面的操作肯定会发生越界异常的，这正是期望的行为，我们认为5的十位数是0，所以将它压入listArr[0]的桶里。   
　　switch (currnetChar)//通过currnetChar的值，确定它压人哪个桶中。   
　　{   
　　case '0': listArr[0].Add(number); break;   
　　case '1': listArr[1].Add(number); break;   
　　case '2': listArr[2].Add(number); break;   
　　case '3': listArr[3].Add(number); break;   
　　case '4': listArr[4].Add(number); break;   
　　case '5': listArr[5].Add(number); break;   
　　case '6': listArr[6].Add(number); break;   
　　case '7': listArr[7].Add(number); break;   
　　case '8': listArr[8].Add(number); break;   
　　case '9': listArr[9].Add(number); break;   
　　default: throw new Exception("unknow error");   
　　}   
　　}   
　　for (int j = 0; j < listArr.Length; j++)//将十个桶里的数据重新排列，压入list   
　　foreach (int number in listArr[j].ToArray<int>())   
　　{   
　　list.Add(number);   
　　listArr[j].Clear();//清空每个桶   
　　}   
　　arr = list.ToArray<int>();//arr指向重新排列的元素   
　　//Console.Write("{0} times:",i);   
　　Print(arr);//输出一次排列的结果   
　　list.Clear();//清空list   
　　}   
　　}   
　　//得到最大元素的位数   
　　private static int GetMaxLength(int[] arr)   
　　{   
　　int iMaxNumber = Int32.MinValue;   
　　foreach (int i in arr)//遍历得到最大值   
　　{   
　　if (i > iMaxNumber)   
　　iMaxNumber = i;   
　　}   
　　return iMaxNumber.ToString().Length;//这样获得最大元素的位数是不是有点投机取巧了...   
　　}   
　　//输出数组元素   
　　public static void Print(int[] arr)   
　　{   
　　foreach (int i in arr)   
　　System.Console.Write(i.ToString()+'\t');   
　　System.Console.WriteLine();   
　　}   
　　//产生随机数组。随机数的范围是0到1000。参数iLength指产生多少个随机数   
　　public static int[] CreateRandomArray(int iLength)   
　　{   
　　int[] arr = new int[iLength];   
　　Random random = new Random();   
　　for (int i = 0; i < iLength; i++)   
　　arr[i] = random.Next(0,1001);   
　　return arr;   
　　}   
　　}   
　　}   
---------------------------------Code ---------------------------------------------