c排序算法大全

c排序算法大全

排序算法是一种基本并且常用的算法。由于实际工作中处理的数量巨大，所以排序算法 对算法本身的速度要求很高。 而一般我们所谓的算法的性能主要是指算法的复杂度，一般用O方法来表示。在后面我将 给出详细的说明。

按照算法的复杂度，从简单到难来分析算法。第一部分是简单排序算法，后面你将看到他们的共同点是算法复杂度为O(N*N)。第二部分是高级排序算法，复杂度为O(Log2(N))。这里我们只介绍一种算法。另外还有几种 算法因为涉及树与堆的概念，所以这里不于讨论。第三部分类似动脑筋。这里的两种算法并不是最好的（甚至有最慢的），但是算法本身比较奇特，值得参考（编程的角度）。同时也可以让我们从另外的角度来认识这个问题。现在，让我们开始吧：

一、简单排序算法

由于程序比较简单，所以没有加什么注释。所有的程序都给出了完整的运行代码，并在我的VC环境

下运行通过。

在代码的后面给出了运行过程示意，希望对理解有帮助。

1.冒泡法：

这是最原始，也是众所周知的最慢的算法了。

```
#include <iostream.h>
void BubbleSort(int* pData,int Count)
{
int iTemp;
for(int i=1;i<Count;i++)
      {
        for(int j=Count-1;j>=i;j--)
            {
              if(pData[j]<pData[j-1])
                    {
                      iTemp = pData[j-1];
                      pData[j-1] = pData[j];
                      pData[j] = iTemp;
                    }
            }
      }
}

void main()
{
int data[] = {10,9,8,7,6,5,4};
BubbleSort(data,7);
for (int i=0;i<7;i++)
          cout<<data[i]<<" ";
cout<<"\n";
}
```

倒序(最糟情况)

第一轮：10,9,8,7->10,9,7,8->10,7,9,8->7,10,9,8(交换3次)

第二轮：7,10,9,8->7,10,8,9->7,8,10,9(交换2次)

第一轮：7,8,10,9->7,8,9,10(交换1次)

循环次数：6次

交换次数：6次

其他：

第一轮：8,10,7,9->8,10,7,9->8,7,10,9->7,8,10,9(交换2次)

第二轮：7,8,10,9->7,8,10,9->7,8,10,9(交换0次)

第一轮：7,8,10,9->7,8,9,10(交换1次)

循环次数：6次

交换次数：3次

上面我们给出了程序段，现在我们分析它：这里，影响我们算法性能的主要部分是循环和交换，显然，次数越多，性能就越差。从上面的

程序我们可以看出循环的次数是固定的，为1+2+...+n-1。 写成公式就是1/2*(n-1)*n。现在注意，我们给出O方法的定义：
若存在一常量K和起点n0，使当n>=n0时，有f(n)<=K*g(n),则f(n) = O(g(n))。（呵呵，不要说没 学好数学呀，对于编程数学是非常重要的！！！）

现在我们来看1/2*(n-1)*n，当K=1/2，n0=1，g(n)=n*n时，1/2*(n-1)*n<=1/2*n*n=K*g(n)。所以f(n) =O(g(n))=O(n*n)。所以我们程序循环的复杂度为O(n*n)。再看交换。从程序后面所跟的表可以看到，两种情况的循环相同，交换不同。其实交换本身同数据源的有序程度有极大的关系，当数据处于倒序的情况时，交换次数同循环一样（每次循环判断都会交换），复杂度为O(n*n)。当数据为正序，将不会有交换。复杂度为O(0)。乱序时处于中间状态。正是由于这样的原因，我们通常都是通过循环次数来对比算法。

2.交换法：

交换法的程序最清晰简单，每次用当前的元素一一的同其后的元素比较并交换。

```
#include <iostream.h>
void ExchangeSort(int* pData,int Count)
{
int iTemp;
for(int i=0;i<Count-1;i++)
        {
          for(int j=i+1;j<Count;j++)
                {
                  if(pData[j]<pData[i])
                        {
                          iTemp = pData[i];
                          pData[i] = pData[j];
                          pData[j] = iTemp;
                        }
                }
        }
}

void main()
{
int data[] = {10,9,8,7,6,5,4};
ExchangeSort(data,7);
for (int i=0;i<7;i++)
            cout<<data[i]<<" ";
cout<<"\n";
}
```

倒序(最糟情况)

第一轮：10,9,8,7->9,10,8,7->8,10,9,7->7,10,9,8(交换3次)

第二轮：7,10,9,8->7,9,10,8->7,8,10,9(交换2次)

第一轮：7,8,10,9->7,8,9,10(交换1次)

循环次数：6次

交换次数：6次

其他：

第一轮：8,10,7,9->8,10,7,9->7,10,8,9->7,10,8,9(交换1次)

第二轮：7,10,8,9->7,8,10,9->7,8,10,9(交换1次)

第一轮：7,8,10,9->7,8,9,10(交换1次)

循环次数：6次

交换次数：3次

从运行的表格来看，交换几乎和冒泡一样糟。事实确实如此。循环次数和冒泡一样也是1/2*(n-1)*n，所以算法的复杂度仍然是O(n*n)。由于我们无法给出所有的情况，所以只能直接告诉大家他们在交换上面也是一样的糟糕（在某些情况下稍好，在某些情况下稍差）。

3.选择法：

现在我们终于可以看到一点希望：选择法，这种方法提高了一点性能（某些情况下） 这种方法类似我们人为的排序习惯：从数据中选择最小的同第一个值交换，在从省下的部分中 选择最小的与第二个交换，这样往复下去。

```
#include <iostream.h>
void SelectSort(int* pData,int Count)
{
int iTemp;
int iPos;
for(int i=0;i<Count-1;i++)
        {
         iTemp = pData[i];
         iPos = i;
         for(int j=i+1;j<Count;j++)
                  {
                   if(pData[j]<iTemp)
                           {
                            iTemp = pData[j];
                            iPos = j;
                           }
                  }
         pData[iPos] = pData[i];
         pData[i] = iTemp;
        }
}

void main()
{
int data[] = {10,9,8,7,6,5,4};
SelectSort(data,7);
for (int i=0;i<7;i++)
        cout<<data[i]<<" ";
cout<<"\n";
}
```

倒序(最糟情况)
第一轮：10,9,8,7->(iTemp=9)10,9,8,7->(iTemp=8)10,9,8,7->(iTemp=7)7,9,8,10(交换1次)
第二轮：7,9,8,10->7,9,8,10(iTemp=8)->(iTemp=8)7,8,9,10(交换1次)
第一轮：7,8,9,10->(iTemp=9)7,8,9,10(交换0次)
循环次数：6次
交换次数：2次

其他：
第一轮：8,10,7,9->(iTemp=8)8,10,7,9->(iTemp=7)8,10,7,9->(iTemp=7)7,10,8,9(交换1次)
第二轮：7,10,8,9->(iTemp=8)7,10,8,9->(iTemp=8)7,8,10,9(交换1次)
第一轮：7,8,10,9->(iTemp=9)7,8,9,10(交换1次)
循环次数：6次
交换次数：3次
遗憾的是算法需要的循环次数依然是1/2*(n-1)*n。所以算法复杂度为O(n*n)。我们来看他的交换。由于每次外层循环只产生一次交换（只有一个最小值）。所以f(n)<=n 所以我们有f(n)=O(n)。所以，在数据较乱的时候，可以减少一定的交换次数。

4.插入法：
插入法较为复杂，它的基本工作原理是抽出牌，在前面的牌中寻找相应的位置插入，然后继续下一张

```
#include <iostream.h>
void InsertSort(int* pData,int Count)
{
int iTemp;
int iPos;
```

```
for(int i=1;i<Count;i++)
            {
             iTemp = pData[i];
             iPos = i-1;
             while((iPos>=0) && (iTemp<pData[iPos]))
                        {
                          pData[iPos+1] = pData[iPos];
                          iPos--;
                        }
             pData[iPos+1] = iTemp;
            }
}

void main()
{
int data[] = {10,9,8,7,6,5,4};
InsertSort(data,7);
for (int i=0;i<7;i++)
          cout<<data[i]<<" ";
cout<<"\n";
}
```

倒序(最糟情况)
第一轮：10,9,8,7->9,10,8,7(交换1次)(循环1次)
第二轮：9,10,8,7->8,9,10,7(交换1次)(循环2次)
第一轮：8,9,10,7->7,8,9,10(交换1次)(循环3次)
循环次数：6次
交换次数：3次

其他：
第一轮：8,10,7,9->8,10,7,9(交换0次)(循环1次)
第二轮：8,10,7,9->7,8,10,9(交换1次)(循环2次)
第一轮：7,8,10,9->7,8,9,10(交换1次)(循环1次)
循环次数：4次
交换次数：2次

上面结尾的行为分析事实上造成了一种假象，让我们认为这种算法是简单算法中最好的，其实不是，因为其循环次数虽然并不固定，我们仍可以使用O方法。从上面的结果可以看出，循环的次数f(n)<= 1/2*n*(n-1)<=1/2*n*n。所以其复杂度仍为O(n*n)（这里说明一下，其实如果不是为了展示这些简单排序的不同，交换次数仍然可以这样推导）。现在看交换，从外观上看，交换次数是O(n)（推导类似选择法），但我们每次要进行与内层循环相同次数的‘=’操作。正常的一次交换我们需要三次‘=’ 而这里显然多了一些，所以我们浪费了时间。

最终，我个人认为，在简单排序算法中，选择法是最好的。

二、高级排序算法：
高级排序算法中我们将只介绍这一种，同时也是目前我所知道（我看过的资料中）的最快的。它的工作看起来仍然象一个二叉树。首先我们选择一个中间值middle程序中我们使用数组中间值，然后把比它小的放在左边，大的放在右边（具体的实现是从两边找，找到一对后交换）。然后对两边分别使 用这个过程（最容易的方法——递归）。
1.快速排序：

```
#include <iostream.h>
void run(int* pData,int left,int right)
{
int i,j;
int middle,iTemp;
i = left;
j = right;
```

```
middle = pData[(left+right)/2]; //求中间值
  do{
       while((pData[i]<middle) && (i<right))//从左扫描大于中值的数
                  i++;
       while((pData[j]>middle) && (j>left))//从右扫描大于中值的数
                  j--;
       if(i<=j)//找到了一对值
                  {
                   //交换
                    iTemp = pData[i];
                    pData[i] = pData[j];
                    pData[j] = iTemp;
                    i++;
                    j--;
                  }
}while(i<=j);//如果两边扫描的下标交错，就停止（完成一次）

//当左边部分有值(left<j)，递归左半边
if(left<j)
         run(pData,left,j);
//当右边部分有值(right>i)，递归右半边
if(right>i)
         run(pData,i,right);
}
void QuickSort(int* pData,int Count)
{
run(pData,0,Count-1);
}
void main()
{
int data[] = {10,9,8,7,6,5,4};
QuickSort(data,7);
for (int i=0;i<7;i++)
          cout<<data[i]<<" ";
cout<<"\n";
}
```

这里我没有给出行为的分析，因为这个很简单，我们直接来分析算法：首先我们考虑最理想的情况
1.数组的大小是2的幂，这样分下去始终可以被2整除。假设为2的k次方，即k=log2(n)。
2.每次我们选择的值刚好是中间值，这样，数组才可以被等分。
第一层递归，循环n次，第二层循环2*(n/2).....
所以共有n+2(n/2)+4(n/4)+...+n*(n/n) = n+n+n+...+n=k*n=log2(n)*n
所以算法复杂度为O(log2(n)*n)
其他的情况只会比这种情况差，最差的情况是每次选择到的middle都是最小值或最大值，那么他将变成交换法（由于使用了递归，情况更糟）。但是你认为这种情况发生的几率有多大？？呵呵，你完全不必担心这个问题。实践证明，大多数的情况，快速排序总是最好的。如果你担心这个问题，你可以使用堆排序，这是一种稳定的O(log2(n)*n)算法，但是通常情况下速度要慢 于快速排序（因为要重组堆）。

三、其他排序
1.双向冒泡：
通常的冒泡是单向的，而这里是双向的，也就是说还要进行反向的工作。 代码看起来复杂，仔细理一下就明白了，是一个来回震荡的方式。写这段代码的作者认为这样可以在冒泡的基础上减少一些交换（我不这么认为，也许我错了）。 反正我认为这是一段有趣的代码，值得一看。

```
#include <iostream.h>
void Bubble2Sort(int* pData,int Count)
```

```
{
int iTemp;
int left = 1;
int right =Count -1;
int t;
do {
        //正向的部分
       for(int i=right;i>=left;i--)
                {
                  if(pData[i]<pData[i-1])
                                {
                                  iTemp = pData[i];
                                  pData[i] = pData[i-1];
                                  pData[i-1] = iTemp;
                                  t = i;
                                }
                }
        left = t+1;
        //反向的部分
        for(i=left;i<right+1;i++)
                {
                  if(pData[i]<pData[i-1])
                                {
                                  iTemp = pData[i];
                                  pData[i] = pData[i-1];
                                  pData[i-1] = iTemp;
                                  t = i;
                                }
                }
        right = t-1;
     }while(left<=right);
}

void main()
{
int data[] = {10,9,8,7,6,5,4};
Bubble2Sort(data,7);
for (int i=0;i<7;i++)
            cout<<data[i]<<" ";
cout<<"\n";
}
```

2.SHELL排序

这个排序非常复杂，看了程序就知道了。 首先需要一个递减的步长，这里我们使用的是9、5、3、1（最后的步长必须是1）。 工作原理是首先对相隔9-1个元素的所有内容排序，然后再使用同样的方法对相隔5-1个元素的排序，以次类推。

```
#include <iostream.h>
void ShellSort(int* pData,int Count)
{
int step[4];
step[0] = 9;
step[1] = 5;
step[2] = 3;
step[3] = 1;
int i,Temp;
int k,s,w;
```

```
for(int i=0;i<4;i++)
        {
         k = step[i];
         s = -k;
        for(int j=k;j<Count;j++)
                {
                 iTemp = pData[j];
                 w = j-k;//求上step个元素的下标
                 if(s ==0)
                        {
                         s = -k;
                         s++;
                         pData[s] = iTemp;
                        }
                while((iTemp<pData[w]) && (w>=0) && (w<=Count))
                        {
                         pData[w+k] = pData[w];
                         w = w-k;
                        }
                pData[w+k] = iTemp;
               }
        }
}

void main()
{
int data[] = {10,9,8,7,6,5,4,3,2,1,-10,-1};
ShellSort(data,12);
for (int i=0;i<12;i++)
          cout<<data[i]<<" ";
cout<<"\n";
}
```

呵呵，程序看起来有些头疼。不过也不是很难，把s==0的块去掉就轻松多了，这里是避免使用0 步长造成程序异常而写的代码。这个代码我认为很值得一看。这个算法的得名是因为其发明者的名字D.L.SHELL。依照参考资料上的说法：“由于复杂的数学原因避免使用2的幂次步长，它能降低算法效率。”另外算法的复杂度为n的1.2次幂。同样因为非常复杂并 “超出本书讨论范围”的原因（我也不知道过程），我们只有结果了.