**《数据结构》**

**实验报告**

专 业 软件工程

年 级 2018级

姓 名 谢梓聪

学 号 201824100445

实 验 室 信工北414

使用日期 2019年12月26日

郑州大学信息工程学院

# 实验九 排序算法的实现

**一、实验目的及要求：**

熟悉排序的方法、过程及原则。掌握插入排序、快速排序、选择排序、归并排序的算法思想及实现方法，掌握其时间复杂度的分析方法。

**二、实验方法：**

1．使用windows平台。

2．使用DEV-C或Visual C软件。

**三、实验内容：**

定义待排序序列的存储结构。验证以下排序算法：

插入类排序：直接插入排序、希尔排序

交换类排序：冒泡排序、快速排序

选择类排序：简单选择排序、堆排序

归并类排序：递归和非递归形式的归并排序

**四、实验结果：**

#include<stdio.h>

#include<stdlib.h>

#define OK 1

#define ERROR 0

void Create(int \*array);

void Output(int \*array);

void InsertSort(int \*array); //直接插入排序

void BInsertSort(int \*array); //折半插入排序

void ShellInsert(int \*array,int increment); //希尔排序

void ShellSort(int \*array);

void BubbleSort(int \*array); //冒泡排序

int Partition(int \*array,int low,int high); //快速排序

void QSort(int \*array,int low,int high);

void QuickSort(int \*array);

void SelectSort(int \*array); //简单选择排序

void Merge(int \*TempMem,int \*SrcMem,int left,int mid,int right); //归并排序

void MSort(int \*TempMem,int \*SrcMem,int left,int right); //二路划分在归并

void MergeSort(int \*array);

void MaxHeapify(int \*array, int start, int end); //堆大顶化调整

void HeapSort(int \*array); //堆排序

void order(void);

void Sort(int \*array,int num);

void MergeSortURec(int \*array);

void MergeOnce(int \*array,int \*temp,int step);

int count=1; //每次创建后的数组元素个数,下标0用来设置各排序方法中的哨兵

/\*

4 7 1 0 2 18 24 15 q 8个数

\*/

int main(void)

{

int array[100]={0},num;

Create(array);

order();

scanf("%d",&num);

Sort(array,num);

Output(array);

return 0;

}

void order(void)

{

printf(

"1. 直接插入排序\n"

"2. 折半插入排序\n"

"3. 希尔排序\n"

"4. 冒泡排序\n"

"5. 快速排序\n"

"6. 简单选择排序\n"

"7. 归并排序\n"

"8. 堆排序\n");

printf("请输入排序序号：");

}

void Sort(int \*array,int num)

{

switch(num)

{

case 1: InsertSort(array);break;

case 2: BInsertSort(array);break;

case 3: ShellSort(array);break;

case 4: BubbleSort(array);break;

case 5: QuickSort(array);break;

case 6: SelectSort(array);break;

case 7: MergeSort(array);break;

case 8: HeapSort(array);break;

}

}

void MergeOnce(int \*array,int \*temp,int step)

{

int i,j;

for(i=1;i<=count-2\*step+1;i+=2\*step)

{

Merge(array,temp,i,i+step,i+2\*step);

}

if(i+step<count+2)

Merge(array,temp,i,i+step,count);

else

for(j=i;j<=count;j++) temp[j]=array[j];

for(i=1;i<=count;i++)

array[i]=temp[i];

}

void MergeSortURec(int \*array)

{

int \*temp,step=1;

temp=(int \*)malloc(sizeof(int)\*count);

while(step<=count)

{

MergeOnce(array,temp,step);

step\*=2;

}

}

void MaxHeapify(int \*array, int start, int end)

{

int dad=start,temp,son=dad\*2;; //父亲节点和孩子节点下标

while (son<=end)

{

if(son+1<=end&&array[son]<array[son+1]) //先比较两个子节点大小，选择最大的

son++;

if(array[dad]>array[son]) //父节点大于孩子节点代表调整完毕，直接跳出函数

return;

else //否则交换双亲和孩子的数据再继续子节点和孙节点比较

{

temp=array[dad];

array[dad]=array[son];

array[son]=temp;

dad=son;

son=dad\*2;

}

}

}

void HeapSort(int \*array)

{

int i,temp;

for (i=count/2;i>0;i--) //从最后一个父节点开始调整

MaxHeapify(array,i,count);

//先将第一个元素和已排好元素前一位做交换，再重新调整，直到排序完毕

for (i=count;i>1;i--)

{

temp=array[1];

array[1]=array[i];

array[i]=temp;

MaxHeapify(array,1,i-1); //将array[1..i-1]重新调整为大顶堆

}

}

void Merge(int \*TempMem,int \*SrcMem,int left,int mid,int right)//将TempMem数组中的两部分left-mid和mid+1-right合并

{

int i,j,k; //左右两个区间的小游标和存放到的原存储的游标

for(i=left,j=mid+1,k=i;i<=mid&&j<=right;k++)

{

if(TempMem[i]<TempMem[j])

SrcMem[k]=TempMem[i++];

else

SrcMem[k]=TempMem[j++];

}

if(i<=mid)

{

for(;i<=mid;i++)

SrcMem[k++]=TempMem[i];

}

if(j<=right)

{

for(;j<=right;j++)

SrcMem[k++]=TempMem[j];

}

}

void MSort(int \*TempMem,int \*SrcMem,int left,int right) //二路划分再归并

{

int mid=(left+right)/2;

int TempMem0[100];

if(left==right) SrcMem[left]=TempMem[left];

else

{

MSort(TempMem,TempMem0,left,mid);

MSort(TempMem,TempMem0,mid+1,right);

Merge(TempMem0,SrcMem,left,mid,right);

}

}

void MergeSort(int \*array)

{

MSort(array,array,1,count);

}

void SelectSort(int \*array)

{

int i,j,min,temp;

for(i=1;i<=count-1;i++)

{

min=i; //若第i个已为最小的，则参与比较时min的初值为i

for(j=i+1;j<=count;j++)

{

if(array[j]<array[min])

{

min=j;

}

}

temp=array[min];

array[min]=array[i];

array[i]=temp;

}

}

int Partition(int \*array,int low,int high)

{

int pivotKey=array[low];

array[0]=array[low];

while(low<high)

{

while(low<high&&array[high]>=pivotKey) high--;

array[low]=array[high];

while(low<high&&array[low]<=pivotKey) low++;

array[high]=array[low];

}

array[low]=array[0];

return low;

}

void QSort(int \*array,int low,int high)

{

int pivot;

if(low<high)

{

pivot=Partition(array,low,high);

QSort(array,low,pivot-1);

QSort(array,pivot+1,high);

}

}

void QuickSort(int \*array)

{

QSort(array,1,count);

}

void BubbleSort(int \*array)

{

int i,j,temp;

for(i=count;i>=2;i--) //总共需要沉n-1次序列中最大的

{

for(j=1;j<=i-1;j++)

{

if(array[j]>array[j+1])

{

temp=array[j];

array[j]=array[j+1];

array[j+1]=temp;

}

}

}

}

void ShellInsert(int \*array,int increment)

{

int i,j;

for(i=1+increment;i<=count;i++)

{

if(array[i]<array[i-increment])

{

array[0]=array[i];

for(j=i-increment;j>0&&array[0]<array[j];j-=increment)

array[j+increment]=array[j];

array[j+increment]=array[0]; //插入到正确的位置

}

}

}

void ShellSort(int \*array)

{ //内部生成增量序列

int delta[10]={0},num=0,i=0,j;

num=count/2;

while(num>=1)

{

delta[i]=num; num-=2; i++;

if(num==2)

{

delta[i]=num;

delta[i+1]=num-1; i+=2; break;

}

}

for(j=0;j<i;j++)

ShellInsert(array,delta[j]);

}

void BInsertSort(int \*array)

{

int i,j;

int high,low,mid;

for(i=2;i<=count;i++)

{

if(array[i]<array[i-1])

{

low=1;

high=i-1; //有序序列的高位和低位游标

array[0]=array[i];

while(low<=high) //二分查找确定插入位置

{

mid=(low+high)/2;

if(array[mid]>array[0])

high=mid-1;

else

low=mid+1;

}

for(j=i-1;j>=high+1;j--) //注意此时必定查找不成功，因此每次查找后插入位置均为high+1;

array[j+1]=array[j];

array[j+1]=array[0]; //插入到正确的位置

}

}

}

void InsertSort(int \*array)

{

int i,j;

for(i=2;i<=count;i++)

{

if(array[i]<array[i-1])

{

array[0]=array[i];

for(j=i-1;array[j]>array[0];j--)

array[j+1]=array[j];

array[j+1]=array[0]; //插入到正确的位置

}

}

}

void Output(int \*array)

{

int i;

printf("排序后序列为：\n");

for(i=1;i<=count;i++)

printf("%d\t",array[i]);

}

void Create(int \*array)

{

count=1;

printf("请输入要创建的数组各元素值,q键结束输入：");

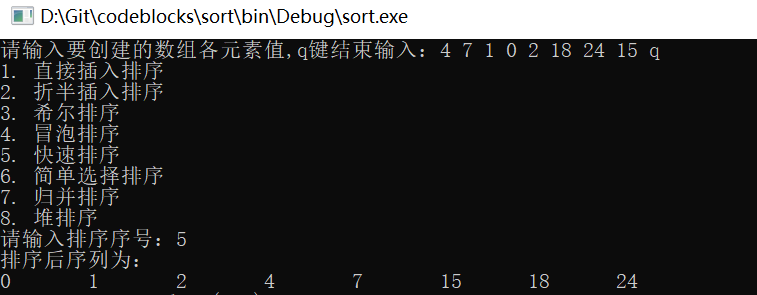
while(scanf("%d",array+count))

count++;

count--;

getchar();

}



**五、实验体会：**

该程序以简单的一维数组为数据结构来进行排序算法的测验，主要分为简单类排序包括直接插入排序、冒泡排序、选择排序、希尔排序等，这类算法由于思想简单而导致时间复杂度均为O(n\*n)，另一类为优化的排序算法包括选择特性的堆排序、交换特性的快速排序和归并排序，这些算法时间复杂度均为O(nlogn)，并且这些排序算法某些与数据正逆序有关，比如冒泡排序正序时时间复杂度为O(n)，而逆序时却为O(n\*n)。