**数据结构上机实验题报告**  
题目：内部排序算法比较  
姓名：王小龙

班级：2020211310

学号：2020211502

提交日期：2021.12.15  
一.题目  
1、本程序对以下6种常用的内部排序算法进行比较：起泡排序、直接插入排序、简单选择排序、快速排序、希尔排序、堆排序。

1. 待排序表的表的表长不小于100；其中的数据要用伪随机数产生程序产生；至少要用5组不同的输入数据作比较；比较的指标为有关关键字参加的比较次数和关键字的移动次数（关键字交换计为3次移动）。

3、最后对结果做出简单分析，包括对各组数据得出结果波动大小给予解释

二．程序设计  
基本操作：

InitList(SqList &L)//初始化线性表

CreatList(SqList &L,int n)//构造一个长度为n的线性表。

ListLength（L）//返回表L中数据元素的个数。

ListTraverse(SqList L)//依次访问并输出线性表L中的元素。

RandNum(SqList &L,int n)//随机乱码构建线性表。

DestroyList(SqList &L)//销毁线性表L。

InitStack（SqStack &S）//构造一个空栈。

PushStack（SqStack &S,SElemtype e）//插入元素e为新的栈顶元素

PopStack（&S,SElemtype &e）//输出S的栈顶元素，并用e返回其值

SaveList(&L)//通过栈的操作来实现线性表的存储。

BubbleSort（SqList &L,int n）//进行起泡排序，计算关键字比较次数c和移动次数s。

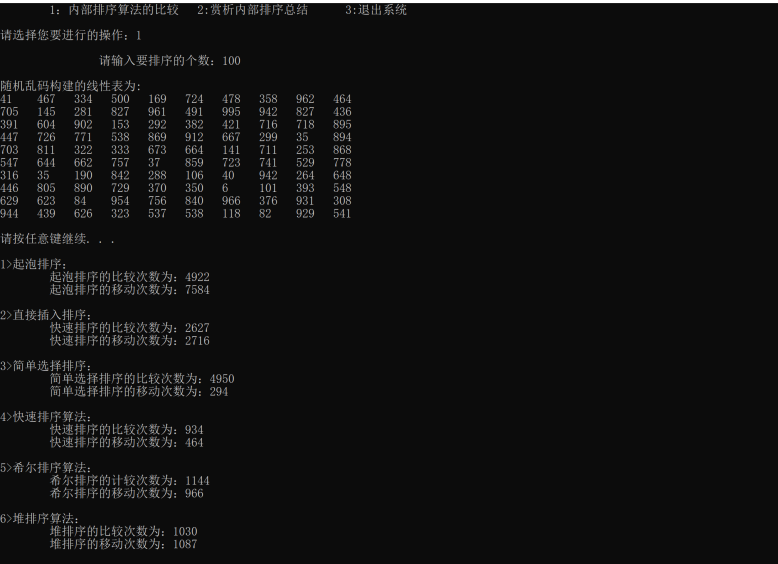
InsertSort(SqList &L,int n)进行插入排序，计算关键字比较次数c和移动次数s。

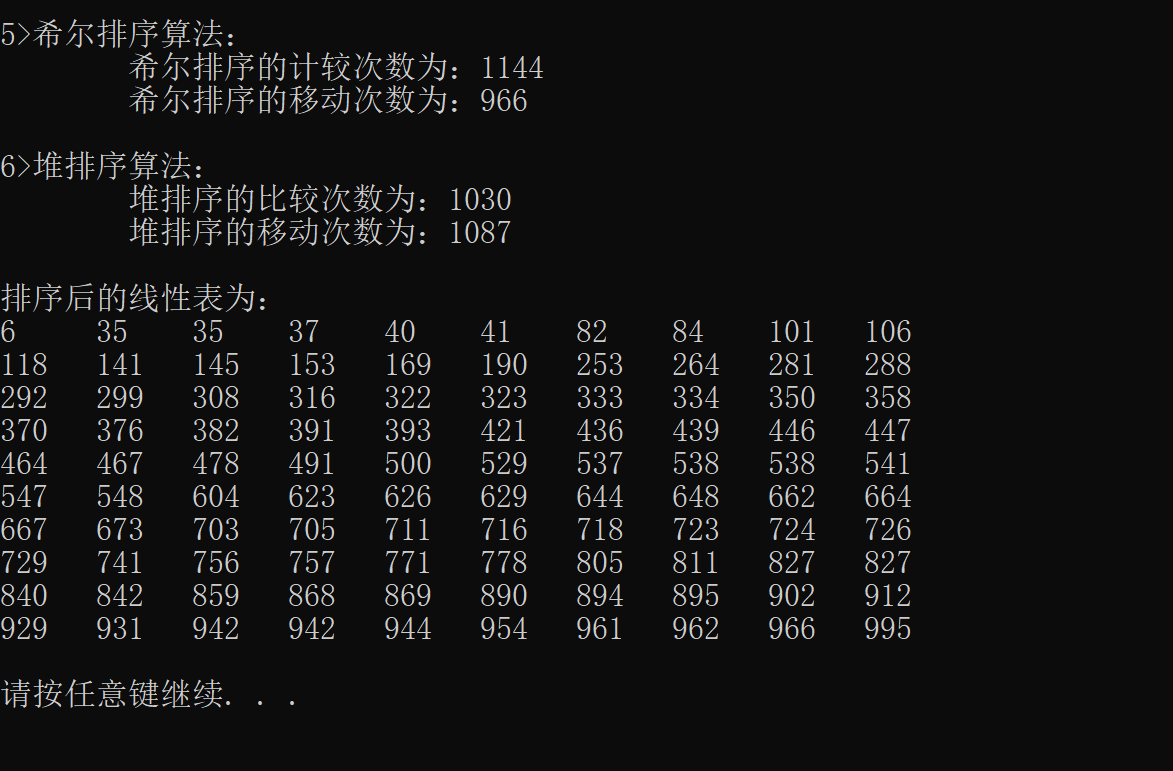
SelectSort(SqList &L,int n)//进行选择排序，计算关键字比较次数c和移动次数s。

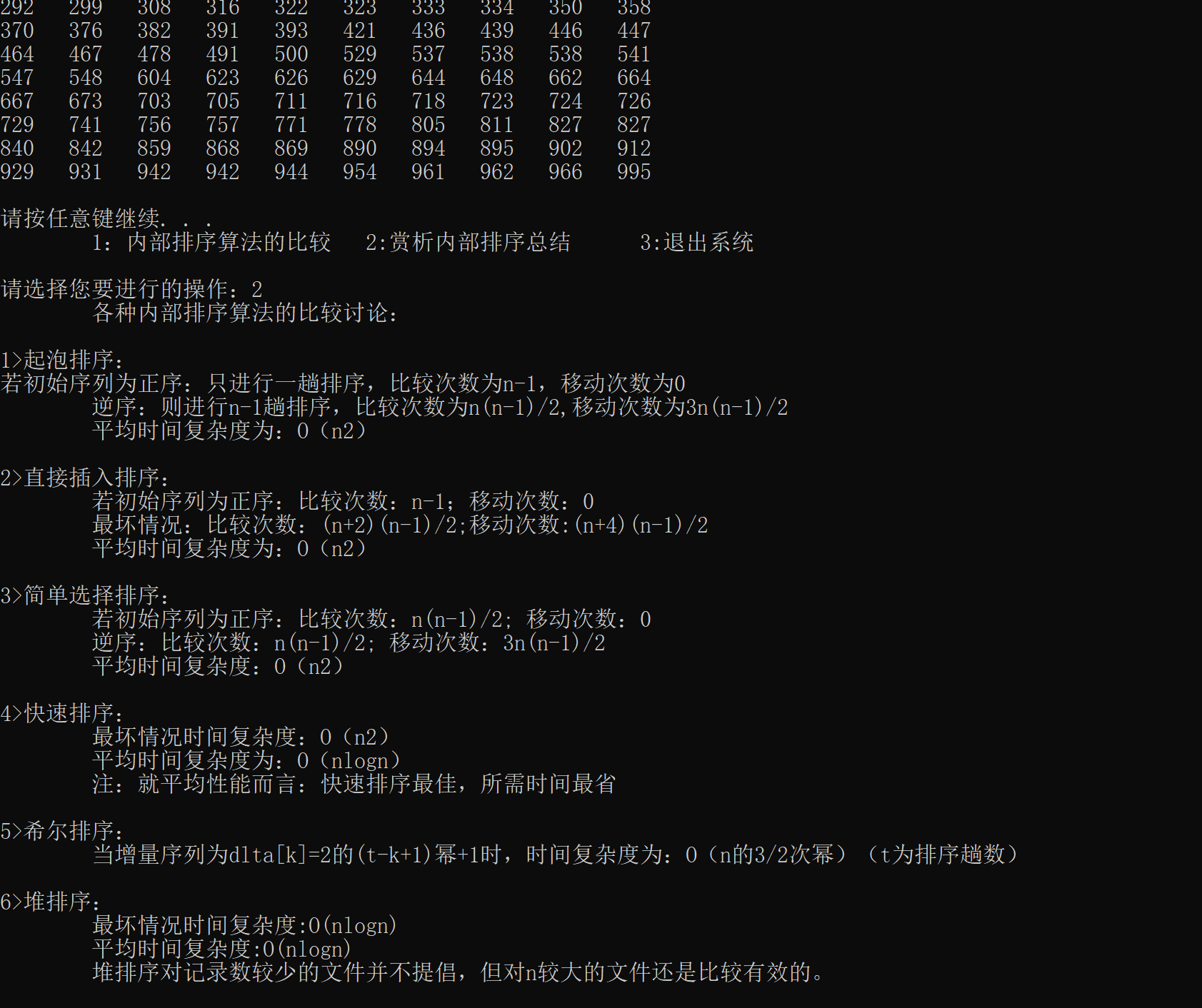
QuickSort(SqList &L,int n)//进行快速排序，计算关键字比较次数c和移动次数s。

ShellSort(SqList &L,int n)//进行希尔排序，计算关键字比较次数和移动次数s。

HeapSort(SqList &L,int n)//进行堆排序，计算关键字比较次数c和移动次数s

三、程序测试运行  




  
四、编程工作总结  
通过本次实验，对各种算法有了深刻的了解。并且打代码前要理一下思路，找到最佳入口来写代码，模块之间的调用也要先构思好，不然后面代码之间过于混乱，当出现bug时，会很难找到，要充分利用调试的功能。

五、程序源代码

#include<stdio.h>

#include<stdlib.h>

#include<malloc.h>

#include<math.h>

#define MAXSIZE 1000 //可排序表的最大长度

#define TRUE 1

#define FALSE 0

typedef int Elemtype;

typedef struct{ //可排序表的存储结构（顺序结构）

Elemtype \*elem;

int listsize;

int length;

}SqList;

int InitList(SqList &L){ //线性表初始化

L.elem=(Elemtype\*)malloc(MAXSIZE\*sizeof(Elemtype));

if(!L.elem)

exit(FALSE);

L.length=0;

L.listsize=MAXSIZE;

return TRUE;

}

int CreatList(SqList &L,int n){ //创建线性表

void RandNum(SqList &L,int n);

RandNum(L,n);

return TRUE;

}

int ListLength(SqList L){ //线性表L的当前长度

return L.length;

}

int ListTraverse(SqList L){ //依次输出线性表L中的元素

if(L.length<=0)

return FALSE;

for(int i=1;i<L.length;i++)

{

if((i-1)%10==0&&(i-1)!=0)

printf("\n");

printf("%-6d",L.elem[i]);

}

printf("\n");

return TRUE;

}

void RandNum(SqList &L,int n){ //随机乱码构建线性表

Elemtype e;

L.elem[0]=0; //L.elem[0]做暂存单元

L.length++;

printf("\n随机乱码构建的线性表为:\n");

for(int i=1;i<=n;i++)

{

if((i-1)%10==0&&i>1)

printf("\n");

e=rand()%1000;

printf("%-6d",e);

L.elem[i]=e;

L.length++;

}

putchar('\n');

}

void DestroyList(SqList &L){

free(L.elem);

L.elem=NULL;

L.length=0;

L.listsize=0;

}

// 栈的基本操作 //

#define STACK\_INIT\_SIZE 1000 //存储空间初始分配量

#define STACKINCREMENT 10 //存储空间分配增量

typedef int SElemtype;

typedef struct{

SElemtype \*base;

SElemtype \*top;

int stacksize;

}SqStack;

int InitStack(SqStack &S){ //初始化栈

S.base=(SElemtype\*)malloc(STACK\_INIT\_SIZE\*sizeof(SElemtype));

if(!S.base)

exit(0);

S.top=S.base;

S.stacksize=STACK\_INIT\_SIZE;

return TRUE;

}

int PushStack(SqStack &S,SElemtype e){ //插入元素e为新的栈顶元素

if(S.top-S.base>=S.stacksize)

{

S.base=(SElemtype\*)realloc(S.base,(S.stacksize+STACKINCREMENT)\*sizeof(SElemtype));

if(!S.base)

exit(FALSE);

S.top=S.base+S.stacksize;

S.stacksize+=STACKINCREMENT;

}

\*S.top=e;

S.top++;

return TRUE;

}

int PopStack(SqStack &S,SElemtype &e){ //若栈不空，则删除栈顶元素，并用e返回其值

if(S.base==S.top)

return FALSE;

e=\*(S.top-1);

S.top--;

return TRUE;

}

void SaveList(SqStack &S,SqList L){ //通过栈的操作来实现线性表的存储

SElemtype e;

for(int i=1;i<L.length;i++)

{

e=L.elem[i];

PushStack(S,e);

}

}

void DestroyStack(SqStack &S){

free(S.base);

S.base=NULL;

S.top=NULL;

S.stacksize=0;

}

// 内部排序基本算法 //

//起泡排序

void BubbleSort(SqList &L,int n){

int i,j,flag,c=0,s=0;

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

{

flag=FALSE;

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

{

c++;

if(L.elem[j]>L.elem[j+1])

{

L.elem[0]=L.elem[j];

L.elem[j]=L.elem[j+1];

L.elem[j+1]=L.elem[0];

s+=3;

flag=TRUE;

}

}

if(!flag)

break;

}

printf("\t起泡排序的比较次数为：%d\n\t起泡排序的移动次数为：%d\n",c,s);

}

//直接插入排序

void InsertSort(SqList &L,int n){

int c=0,s=0,i,j;

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

{

c++;

if(L.elem[i]<L.elem[i-1])

{

L.elem[0]=L.elem[i];

L.elem[i]=L.elem[i-1];

s+=2;

for(j=i-2;L.elem[0]<L.elem[j];j--)

{

L.elem[j+1]=L.elem[j];

c++;

s++;

}

c++;

L.elem[j+1]=L.elem[0];

s++;

}

}

printf("\t快速排序的比较次数为：%d\n\t快速排序的移动次数为：%d\n",c,s);

}

//简单选择排序

void SelectSort(SqList &L,int n){

int i,j,t,c=0,s=0;

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

{

t=i;

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

{

c++;

if(L.elem[t]>L.elem[j])

t=j;

}

if(t!=i)

{

L.elem[0]=L.elem[i];

L.elem[i]=L.elem[t];

L.elem[t]=L.elem[0];

s+=3;

}

}

printf("\t简单选择排序的比较次数为：%d\n\t简单选择排序的移动次数为：%d\n",c,s);

}

//快速排序

int Partition(SqList &L,int low,int high,int &c,int &s){

L.elem[0]=L.elem[low];

s++;

while(low<high)

{

while(low<high&&L.elem[0]<=L.elem[high])

{

c++;

high--;

}

if(low<high)

c++;

L.elem[low]=L.elem[high];

s++;

while(low<high&&L.elem[0]>=L.elem[low])

{

c++;

low++;

}

if(low<high)

c++;

L.elem[high]=L.elem[low];

s++;

}

L.elem[low]=L.elem[0];

s++;

return low;

}

void QSort(SqList &L,int low,int high,int &c,int &s){

int pivotkey;

if(low<high)

{

pivotkey=Partition(L,low,high,c,s);

QSort(L,low,pivotkey-1,c,s);

QSort(L,pivotkey+1,high,c,s);

}

}

void QuickSort(SqList &L,int n){

int low=1,high=n;

static int c=0,s=0;

QSort(L,low,high,c,s);

printf("\t快速排序的比较次数为：%d\n\t快速排序的移动次数为：%d\n",c,s);

}

//希尔排序

void ShellInsert(SqList &L,int n,int dk,int &c,int &s){

int i,j;

for(i=dk+1;i<=n;i++)

{

c++;

if(L.elem[i]<L.elem[i-dk])

{

L.elem[0]=L.elem[i];

s++;

for(j=i-dk;j>0&&(L.elem[0]<L.elem[j]);j-=dk)

{

L.elem[j+dk]=L.elem[j];

c++;

s++;

}

c++;

L.elem[j+dk]=L.elem[0];

s++;

}

}

}

void ShellSort(SqList &L,int n){

static int c=0,s=0;

int t=4; //t为排序趟数

int dlta[4]={7,5,3,1};

for(int m=0;m<t;m++)

ShellInsert(L,n,dlta[m],c,s);

printf("\t希尔排序的计较次数为：%d\n\t希尔排序的移动次数为：%d\n",c,s);

}

//堆排序

void HeapAdjust(SqList &L,int r,int m,int &c,int &s){

L.elem[0]=L.elem[r];

s++;

for(int j=2\*r;j<=m;j\*=2)

{

if(j<m&&(L.elem[j]<L.elem[j+1]))

j++;

if(j<m)

c+=2;

else

c++;

if(L.elem[0]>=L.elem[j])

break;

L.elem[r]=L.elem[j];

s++;

r=j;

}

L.elem[r]=L.elem[0];

s++;

}

void HeapSort(SqList &L,int n){

static int c=0,s=0,i;

for(i=n/2;i>0;i--)

HeapAdjust(L,i,n,c,s);

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

{

L.elem[0]=L.elem[i];

L.elem[i]=L.elem[1];

L.elem[1]=L.elem[0];

s+=3;

HeapAdjust(L,1,i-1,c,s);

}

printf("\t堆排序的比较次数为：%d\n\t堆排序的移动次数为：%d\n",c,s);

}

void Sort() //对内部排序算法函数的调用

{

int n,k,i;

SElemtype e;

SqList L;

SqStack S;

InitStack(S);

InitList(L);

printf("\t\t请输入要排序的个数：");

scanf("%d",&n);

CreatList(L,n);

k=ListLength(L);

SaveList(S,L);

putchar('\n');

system("pause");

putchar('\n');

//1>测试起泡排序

printf("1>起泡排序：\n");

BubbleSort(L,n);

putchar('\n');

//2>恢复排序前线性表，测试直接插入排序

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

{

e=\*(S.top-k+i);

L.elem[i]=e;

}

printf("2>直接插入排序：\n");

InsertSort(L,n);

putchar('\n');

//3>恢复排序前线性表，测试简单选择排序

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

{

e=\*(S.top-k+i);

L.elem[i]=e;

}

printf("3>简单选择排序：\n");

SelectSort(L,n);

putchar('\n');

//4>恢复排序前线性表，测试快速排序

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

{

e=\*(S.top-k+i);

L.elem[i]=e;

}

printf("4>快速排序算法：\n");

QuickSort(L,n);

putchar('\n');

//5>恢复排序前线性表，测试希尔排序

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

{

e=\*(S.top-k+i);

L.elem[i]=e;

}

printf("5>希尔排序算法：\n");

ShellSort(L,n);

putchar('\n');

//6>恢复排序前线性表，测试堆排序

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

{

e=\*(S.top-k+i);

L.elem[i]=e;

}

printf("6>堆排序算法：\n");

HeapSort(L,n);

putchar('\n');

//排序后的线性表

printf("排序后的线性表为：\n");

ListTraverse(L);

//对线性表和栈进行销毁，便于执行下次测试

DestroyList(L);

DestroyStack(S);

}

void Read()

{

FILE \*fp;

char \*buffer,str[2000];

buffer=str;

fp=fopen("lv.txt","r");

if(!fp)

exit(0);

fread(buffer,sizeof(char)\*850,2,fp);

\*(str+850)='\0';

printf("%s\n",str);

fclose(fp);

}

int main()

{

int cmd;

printf("\t1：内部排序算法的比较\t2:赏析内部排序总结\t3:退出系统\n\n");

printf("请选择您要进行的操作：");

scanf("%d",&cmd);

printf("\n");

while(cmd<1||cmd>3)

{

getchar();

printf("\n输入有错，请重新输入：\n");

scanf("%d",&cmd);

}

for(;;)

{

switch(cmd)

{

case 1:

{

Sort();

break;

}

case 2:

{

Read();

break;

}

case 3:

{

exit(0);

break;

}

}

putchar('\n');

system("pause");

printf("\t1：内部排序算法的比较\t2:赏析内部排序总结\t3:退出系统\n\n");

printf("请选择您要进行的操作：");

scanf("%d",&cmd);

while(cmd<1||cmd>3)

{

getchar();

printf("\n输入有错，请重新输入：\n");

scanf("%d",&cmd);

}

}

return 0;

}