**第三部分 第1题： 内部排序算法比较**

**实验报告**

题目：对8种常见的内部排序算法进行比较。

班级：F1703407 姓名：赵伟基 学号：517021910883 完成日期：2018-12-18

1. **需求分析**

课本中给出了八种内部排序算法的时间复杂度分析，任务要求是通过随机数据来比较各算法的关键字比较次数和关键字移动速度，来对各种排序算法取得更直观的感受。主要工作在于对各种排序算法中插入对关键字比较和移动次数的统计操作。

（1）对以下8种常用的内部排序算法进行比较：冒泡排序、直接插入排序、简单选择 排序、快速排序、希尔排序、堆排序、折半插入排序、归并排序。

（2）待排序表的表长不小于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；比较的指标为有关键字参与的比较次数和关键字的移动次数（关键字一次交换计为3次移动）。

（3）最后要对结果做出简单分析，包括对各组数据得出结果波动大小的解释。本实验采用matlab对结果进行画图分析。

1. **概要设计**

本实验要求对输入的数据进行排序，故选取顺序结构存储数据；由于需要对八种算法来对数组排序，故需要定义八个（及以上）的排序功能函数；此外对数据典型性（如正序、逆序）等进行考虑，故设置排序正序(zhengxu())，逆序(fanxu())和随机数据（total\_ram()）的函数。

1. 主程序模块

int main()

{

初始化；

Total\_ram(); //完全随机的乱序排序统计

zhengxu(); //正序的排序统计

fanxu(); //反序的排序统计

}

1. 数据存储结构

int A[N],B[N],C[N],D[N],E[N],F[N],G[N],H[N];

//N为数据条数 本实验设定为1000

//SR 限制了随机数的范围 本实验设定为100000

1. 排序功能函数模块：
2. BubbleSort ( ):

初始条件：输入待排序数组A[N]及数据条数n；

操作过程：利用冒泡排序算法，对有关键字参与的比较次数和关键字的移动次数进行统计,返回比较次数a1及交换次数aa；

1. simpleInsert():

初始条件：输入待排序数组B[N]及数据条数n；

操作过程：利用直接插入排序算法，对有关键字参与的比较次数和关键字的移动次数进行统计,返回比较次数b及交换次数bb；

1. simpleSelect():

初始条件：输入待排序数组C[N]及数据条数n；

操作过程：利用简单选择排序算法，对有关键字参与的比较次数和关键字的移动次数进行统计,返回比较次数c及交换次数cc；

1. quicksort():

初始条件：输入待排序数组D[N]及数据条数n；

操作过程：利用快速排序算法，对有关键字参与的比较次数和关键字的移动次数进行统计,返回比较次数d及交换次数dd；

1. shellsort():

初始条件：输入待排序数组E[N]及数据条数n；

操作过程：利用希尔排序算法，对有关键字参与的比较次数和关键字的移动次数进行统计,返回比较次数e及交换次数ee；

1. heapSort():

初始条件：输入待排序数组F[N]及数据条数n；

操作过程：利用堆排序算法，对有关键字参与的比较次数和关键字的移动次数进行统计,返回比较次数f及交换次数ff；

1. mergeSort():

初始条件：输入待排序数组G[N]及数据条数n；

操作过程：利用归并排序算法，对有关键字参与的比较次数和关键字的移动次数进行统计,返回比较次数g及交换次数gg；

1. insertSortHalf():

初始条件：输入待排序数组H[N]及数据条数n；

操作过程：利用折半插入排序算法，对有关键字参与的比较次数和关键字的移动次数进行统计,返回比较次数h及交换次数hh；

1. fanBubbleSort ():

初始条件：输入待排序数组a[N]及数据条数n；

操作过程：利用冒泡排序算法，对a[N]数组逆序排序，即递减序；

1. 功能函数模块
   * 1. change()：

初始条件：输入两个int类型数地址

返回结果：交换两数的值，同时返回交换次数3；

* + 1. 包裹函数：quicksort（int a[],int low,int high）；

mergeSort ( int a[],int left,int right);

* + 1. 算法相关函数：
       1. divide(int a[],int low,int high)

初始条件：输入数组和划分的区间大小

操作结果：实现一次划分返回划分后的low值

* + - 1. percolateDown(int a[],int hole,int n)

初始条件：输入数组，向下过滤的初始位置，目前堆的大小

返回结果：完成一次堆的详细过滤

* + - 1. merge(int a[],int left,int mid,int right)

初始条件：输入数组和两个数组的始末点

返回结果：返回归并的两个有序数列

* + 1. 打印模块

zhengxu（）：产生递增数列数组进行排序，打印各排序方式的统计结果

fanxu（）：产生递减数列数组进行排序，打印各排序方式的统计结果

total\_ram():产生完全乱序的数组进行排序，打印各排序方式的统计结果

1. **详细设计**
2. paixu.h

#ifndef PAIXU\_H

#define PAIXU\_H

# include <cstdio>

# include <cstdlib>

# include <iostream>

# include <ctime>

# include <iomanip>

# define N 1000

# define SR 100000

using namespace std;

int A[N],B[N],C[N],D[N],E[N],F[N],G[N],H[N];

int change(int \*a,int \*b);

// a1,b,c,d,e,f,g,h 为比较次数统计

//aa,bb,cc,dd,ee,ff,gg,hh 为交换次数统计

void BubbleSort(int a[],int n); //a1 aa 冒泡排序 正序

void simpleInsert(int a[],int n); //b,bb 简单插入排序 正序

void simpleSelect(int a[],int n); //c cc 简单选择排序 正序

void quicksort(int a[],int low,int high); //d dd 快速排序 正序

void quicksort(int a[],int n);

int divide(int a[],int low,int high);

void shellsort(int a[],int n); //e ee 希尔排序正序

void heapSort(int a[],int n); //f ff 堆排序 正序

void percolateDown(int a[],int hole,int n);

void mergeSort(int a[],int left,int right);

void merge(int a[],int left,int mid,int right);

void mergeSort(int a[],int n); //g gg 归并排序 正序

void insertSortHalf(int a[],int n); //h hh 折半插入排序 正序

void fanBubbleSort(int a[],int n); // 冒泡排序 反序

void total\_ram(); //对随机数组排序

void zhengxu(); //对正序数组排序

void fanxu(); //对反序数组排序

int a1=0,aa=0,b=0,bb=0,c=0,cc=0,d=0,dd=0,e=0,ee=0,f=0,ff=0,g=0,gg=0,h=0,hh=0;

#endif // PAIXU\_H

1. main.cpp

# include "paixu.h"

using namespace std;

int main()

{

int i=0,num=N,case1=0;

srand(time(0));

total\_ram(); //随机排序 包含产生随机数8种算法排序和打印结果的操作

zhengxu();//正序排序 包含产生随机数8种算法排序和打印结果的操作

fanxu();//逆序排序 包含产生随机数8种算法排序和打印结果的操作

return 0;

}

void zhengxu()

{

int i=0,case1=0,num=N;

cout<<endl<<"------------------数据为正序--------------------";

for (case1=0;case1<5;case1++) //产生五组数据

{

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

{

A[i]=B[i]=C[i]=D[i]=E[i]=F[i]=G[i]=H[i]=rand()%SR; //产生随机数

}

/\* for(i=0;i<num;i++)

{

cout<<A[i]<<" "<<B[i]<<" "<<C[i]<<" "<<D[i]<<" "<<F[i]<<" "<<G[i]<<" "<<H[i]<<endl;

}\*/

quicksort(A,num);quicksort(B,num);quicksort(C,num);quicksort(D,num);quicksort(E,num);quicksort(F,num);quicksort(G,num);quicksort(H,num); //产生正序数组

cout<<endl<<"case"<<case1+1<<":"<<endl;

BubbleSort(A,num);

cout<<endl<<"BubbleSort:"<<"compare times"<<a1<<" movetimes"<<aa;

simpleInsert(B,num);

cout<<endl<<"simpleInsert:"<<"compare times"<<b<<" movetimes"<<bb;

simpleSelect(C,num);

cout<<endl<<"simpleSelect:"<<"compare times"<<c<<" movetimes"<<cc;

quicksort(D,num);

cout<<endl<<"quicksort:"<<"compare times"<<d<<" movetimes"<<dd;

shellsort(E,num);

cout<<endl<<"shellsort:"<<"compare times"<<e<<" movetimes"<<ee;

heapSort(F,num);

cout<<endl<<"heapSort:"<<"compare times"<<f<<" movetimes"<<ff;

mergeSort(G,num);

cout<<endl<<"mergeSort:"<<"compare times"<<g<<" movetimes"<<gg;

insertSortHalf(H,num);

cout<<endl<<"insertSortHalf:"<<"compare times"<<h<<" movetimes"<<hh;

a1=0;d=0;dd=0;e=0;ee=0;

aa=0;b=0;bb=0;c=0;cc=0;

g=0;gg=0;f=0;ff=0;

h=0;hh=0;

/\* for(i=0;i<num;i++)

{

cout<<endl<<A[i]<<" "<<B[i]<<" "<<C[i]<<" "<<D[i]<<" "<<F[i]<<" "<<G[i]<<" "<<H[i];

}\*/

}

cout<<endl<<"------------------------------------------";

}

void fanxu()

{

int case1=0,i=0,num=N;

cout<<endl<<"-----------------数据为逆序---------------------";

for (case1=0;case1<5;case1++)

{

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

{

A[i]=B[i]=C[i]=D[i]=E[i]=F[i]=G[i]=H[i]=rand()%SR;

}

/\* for(i=0;i<num;i++)

{

cout<<A[i]<<" "<<B[i]<<" "<<C[i]<<" "<<D[i]<<" "<<F[i]<<" "<<G[i]<<" "<<H[i]<<endl;

}\*/

fanBubbleSort(A,num); //产生逆序数组

fanBubbleSort(B,num);

fanBubbleSort(C,num);

fanBubbleSort(D,num);

fanBubbleSort(E,num);

fanBubbleSort(F,num);

fanBubbleSort(G,num);

fanBubbleSort(H,num);

cout<<endl<<"case"<<case1+1<<":"<<endl;

BubbleSort(A,num);

cout<<endl<<"BubbleSort:"<<"compare times"<<a1<<" movetimes"<<aa;

simpleInsert(B,num);

cout<<endl<<"simpleInsert:"<<"compare times"<<b<<" movetimes"<<bb;

simpleSelect(C,num);

cout<<endl<<"simpleSelect:"<<"compare times"<<c<<" movetimes"<<cc;

quicksort(D,num);

cout<<endl<<"quicksort:"<<"compare times"<<d<<" movetimes"<<dd;

shellsort(E,num);

cout<<endl<<"shellsort:"<<"compare times"<<e<<" movetimes"<<ee;

heapSort(F,num);

cout<<endl<<"heapSort:"<<"compare times"<<f<<" movetimes"<<ff;

mergeSort(G,num);

cout<<endl<<"mergeSort:"<<"compare times"<<g<<" movetimes"<<gg;

insertSortHalf(H,num);

cout<<endl<<"insertSortHalf:"<<"compare times"<<h<<" movetimes"<<hh;

a1=0;d=0;dd=0;e=0;ee=0;

aa=0;b=0;bb=0;c=0;cc=0;

g=0;gg=0;f=0;ff=0;

h=0;hh=0;

/\* for(i=0;i<num;i++)

{

cout<<endl<<A[i]<<" "<<B[i]<<" "<<C[i]<<" "<<D[i]<<" "<<F[i]<<" "<<G[i]<<" "<<H[i];

}\*/

}

cout<<endl<<"------------------------------------------";

}

void total\_ram()

{

int i=0,num=N,case1=0;

cout<<endl<<"---------------数据排序完全随机分布-------------------";

for (case1=0;case1<5;case1++)

{

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

{

A[i]=B[i]=C[i]=D[i]=E[i]=F[i]=G[i]=H[i]=rand()%SR;

}

cout<<endl<<"case"<<case1+1<<":"<<endl;

BubbleSort(A,num);

cout<<endl<<"BubbleSort:"<<"compare times"<<a1<<" movetimes"<<aa;

simpleInsert(B,num);

cout<<endl<<"simpleInsert:"<<"compare times"<<b<<" movetimes"<<bb;

simpleSelect(C,num);

cout<<endl<<"simpleSelect:"<<"compare times"<<c<<" movetimes"<<cc;

quicksort(D,num);

cout<<endl<<"quicksort:"<<"compare times"<<d<<" movetimes"<<dd;

shellsort(E,num);

cout<<endl<<"shellsort:"<<"compare times"<<e<<" movetimes"<<ee;

heapSort(F,num);

cout<<endl<<"heapSort:"<<"compare times"<<f<<" movetimes"<<ff;

mergeSort(G,num);

cout<<endl<<"mergeSort:"<<"compare times"<<g<<" movetimes"<<gg;

insertSortHalf(H,num);

cout<<endl<<"insertSortHalf:"<<"compare times"<<h<<" movetimes"<<hh;

a1=0;d=0;dd=0;e=0;ee=0;

aa=0;b=0;bb=0;c=0;cc=0;

g=0;gg=0;f=0;ff=0;

h=0;hh=0;

}

cout<<endl<<"------------------------------------------";

}

//交换函数

int change(int \*a,int \*b)

{

int c=\*a;

\*a=\*b;

\*b=c;

return 3;

}

//冒泡排序

void BubbleSort(int a[],int n)

{

int i,j;

bool flag=true;

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

{

flag=false;

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

{

if (a[j+1]<a[j])

{

aa+=change(&a[j+1],&a[j]);

flag=true;

}

a1++;

}

}

}

//简单插入排序

void simpleInsert(int a[],int n)

{

int k;

int tmp;

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

{

tmp=a[j];

for(b++,bb++,k=j-1;tmp<a[k]&&k>=0;--k,b++,bb++)

a[k+1]=a[k];

a[k+1]=tmp;bb++;

}

}

//简单选择排序

void simpleSelect(int a[],int n)

{

int i,j,min;

for(i=0;i<n-1;i++)

{

min=i;

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

if(a[j]<a[min]) min=j;

cc+=change(&a[i],&a[min]);

}

}

//快速排序

void quicksort(int a[],int low,int high)

{

int mid;

if(low>=high) return;

mid=divide(a,low,high);

quicksort(a,low,mid-1);

quicksort(a,mid+1,high);

}

void quicksort(int a[],int n)

{

quicksort(a,0,n-1);

}

int divide(int a[],int low,int high)

{

int k=a[low];

dd++;

do{

while(low<high&&a[high]>=k) {--high;d++;}

d++;

if(low<high){a[low]=a[high];++low;dd++;}

while (low<high&&a[low]<=k) {++low;d++;}

d++;

if(low<high) {a[high]=a[low];--high;dd++;}

}while(low!=high);

a[low]=k;

dd++;

return low;

}

//希尔排序

void shellsort(int a[],int n)

{

int step,j,i;

int temp;

for(step=n/2;step>0;step/=2)

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

{

temp=a[i];ee++;

for(j=i-step;j>=0&&a[j]>temp;j-=step)

{

a[j+step]=a[j];

e++;

ee++;

}

e++;

a[j+step]=temp;

ee++;

}

}

void percolateDown(int a[],int hole,int n)

{

int child;

int tmp=a[hole];

ff++;

for(;hole\*2+1<n;hole=child)

{

child=hole\*2+1;

f+=2;

if(child!=n-1&&a[child+1]>a[child])

{child++;}

if(a[child]>tmp) {a[hole]=a[child];ff++;}

else break;

}

a[hole]=tmp;ff++;

}

//堆排序

void heapSort(int a[],int n)

{

int i;

int tmp;

for(i=n/2-1;i>=0;i--)

percolateDown(a,i,n);

for(i=n-1;i>0;--i)

{

tmp=a[0];

a[0]=a[i];

a[i]=tmp;

ff+=3;

percolateDown(a,0,i);

}

}

//归并排序

void merge(int a[],int left,int mid,int right)

{

int \*tmp=new int [right-left+1];

int i=left,j=mid,k=0;

while(i<mid&&j<=right)

{if(a[i]<a[j]) tmp[k++]=a[i++];

else tmp[k++]=a[j++];

g++;gg++;}

while(i<mid) {tmp[k++]=a[i++];gg++;}

while(j<=right) {tmp[k++]=a[j++];gg++;}

for(i=0,k=left;k<=right;) {a[k++]=tmp[i++];gg++;}

delete []tmp;

}

void mergeSort(int a[],int left,int right)

{

int mid=(left+right)/2;

if(left==right) return;

mergeSort(a,left,mid);

mergeSort(a,mid+1,right);

merge(a,left,mid+1,right);

}

void mergeSort(int a[],int n)

{

mergeSort(a,0,n-1);

}

// 折半插入排序

void insertSortHalf(int a[],int n){

int i,j,low,high,mid,temp;

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

temp =a[i]; hh++;

low= 0;

high = i-1;

while(low<=high){

mid = (low+high)/2;

h++;

if(temp>a[mid])

low= mid+1;

else

high = mid-1;

}

for(j= i-1;j>=high+1;j--)

{a[j+1] = a[j];hh++;}

a[high+1] = temp;hh++;

}

}

//反向冒泡排序

void fanBubbleSort(int a[],int n)

{

int i,j;

bool flag=true;

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

{

flag=false;

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

{

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

{

change(&a[j+1],&a[j]);

flag=true;

}

}

}

}

1. **调试分析**
2. 该实验参数传输类型失当，可以在函数中直接对待排序数组进行操作，无需传参。
3. 该算法未对用户界面进行很好的设计，只满足了实验的要求
4. 算法复杂度分析

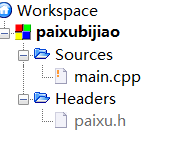
（1）：时间复杂度

1. O(n^2):BubbleSort() ;InsertSortHalf();SimpleSelect();fanBubbleSort();SimpleInsert();zhengxu();fanxu();total\_ram();
2. O(n^(3/2)):shellsort();
3. O(nlog2(n)):heapsort();mergesort();quicksort();

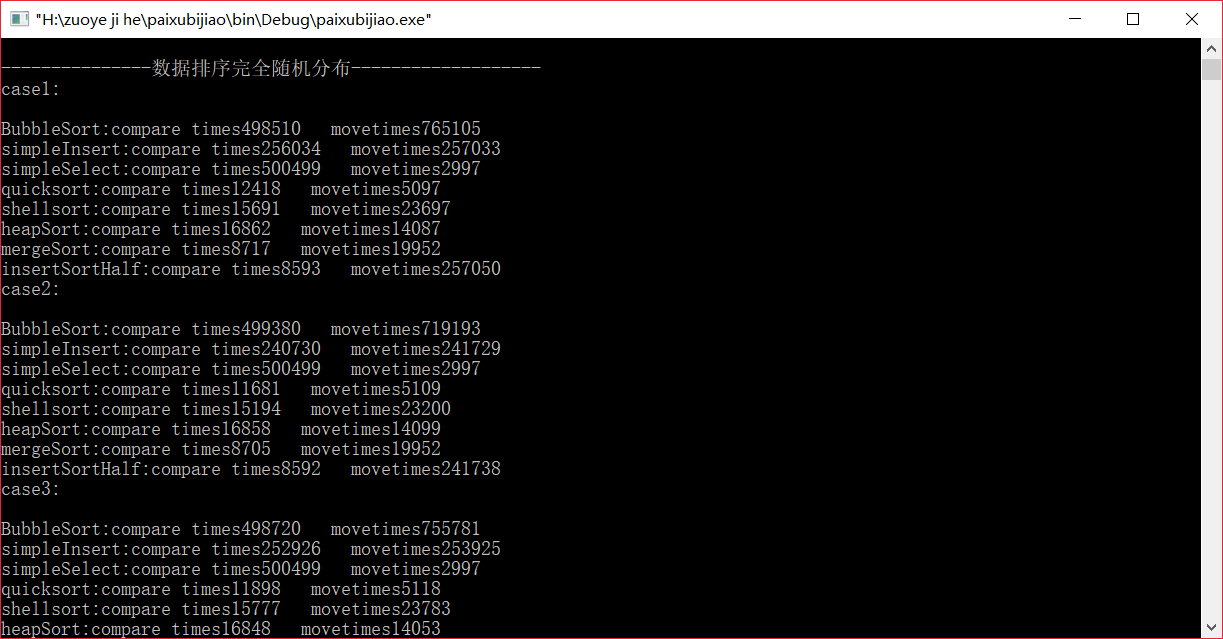
（2）：空间复杂度

1. O(1):InsertSortHalf();SimpleSelect();fanBubbleSort();SimpleInsert();BubbleSort() ; shellsort();heapsort()
2. O(log2(n)):quicksort();
3. O(n): mergesort();
4. **用户手册**

1．本程序使用的 Code::Blocks 16.01 IDE，程序以项目（project）方式组织，如图 1 所示



2． 用户点击main.cpp->build and run 即可获得默认N=1000条数据的随机排序 正序排序 逆序排序的五次数据结果。



实验结果如图所示

3. 如果用户需要改变数据条数或数据范围，在paixu.h文件中改变N与SR的定义即可：（N：数据条数； [0,SR]: 随机数范围）



1. **测试结果**
2. **比较各排序算法排列数组（初始状态随机）下的compare times 和move times**

注：N取1000 随机数范围在[0,100000]之间

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| compare times | case1 | case2 | case3 | case4 | case5 | average |
| BubbleSort | 499269 | 499247 | 499680 | 497789 | 499269 | 499050.8 |
| simpleInsert | 247728 | 252361 | 250756 | 247344 | 250619 | 249761.6 |
| simpleSelect | 500499 | 500499 | 500499 | 500499 | 500499 | 500499 |
| quicksort | 11963 | 12008 | 12129 | 11760 | 11784 | 11928.8 |
| shellSort | 15945 | 15421 | 15568 | 16382 | 15905 | 15844.2 |
| heapSort | 16826 | 16870 | 16892 | 16846 | 16866 | 16860 |
| mergeSort | 8682 | 8709 | 8701 | 8734 | 8721 | 8709.4 |
| insertSortHalf | 8583 | 8591 | 8588 | 8580 | 8580 | 8584.4 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| movetimes | case1 | case2 | case3 | case4 | case5 | average |
| BubbleSort | 740187 | 754086 | 749271 | 739035 | 748860 | 746287.8 |
| simpleInsert | 248727 | 253360 | 251755 | 248343 | 251618 | 250760.6 |
| simpleSelect | 2997 | 2997 | 2997 | 2997 | 2997 | 2997 |
| quicksort | 5030 | 5058 | 5090 | 5103 | 5078 | 5071.8 |
| shellSort | 23951 | 23427 | 23574 | 24388 | 23911 | 23850.2 |
| heapSort | 14055 | 14076 | 14108 | 14073 | 14065 | 70377 |
| mergeSort | 19952 | 19952 | 19952 | 19952 | 19952 | 19952 |
| insertSortHalf | 248741 | 253375 | 251774 | 248360 | 251633 | 250776.6 |

在原数据乱序的情况下，由上述数据可以验证各排序算法复杂度与比较次数和实际交换次数之间的关系，其比较次数与交换次数的数量级反映了算法的平均时间复杂度。

1. **比较各排序算法排列数组（正序）下的compare times 和move times**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| comparetimes | case1 | case2 | case3 | case4 | case5 | average |
| BubbleSort | 999 | 999 | 999 | 999 | 999 | 999 |
| simpleInsert | 999 | 999 | 999 | 999 | 999 | 999 |
| simpleSelect | 500499 | 500499 | 500499 | 500499 | 500499 | 500499 |
| quicksort | 597146 | 599514 | 596570 | 599434 | 598866 | 598306 |
| shellSort | 8006 | 8006 | 8006 | 8006 | 8006 | 8006 |
| heapSort | 17624 | 17620 | 17610 | 17630 | 17632 | 17623.2 |
| mergeSort | 5055 | 5054 | 5051 | 5048 | 5051 | 5051.8 |
| insertSortHalf | 8977 | 8977 | 8977 | 8977 | 8977 | 8977 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| movetimes | case1 | case2 | case3 | case4 | case5 | average |
| BubbleSort | 0 | 0 | 0 | 0 | 0 | 0 |
| simpleInsert | 1998 | 1998 | 1998 | 1998 | 1998 | 1998 |
| simpleSelect | 2997 | 2997 | 2997 | 2997 | 2997 | 2997 |
| quicksort | 42390 | 42630 | 43110 | 42590 | 42278 | 42599.6 |
| shellSort | 16012 | 16012 | 16012 | 16012 | 16012 | 16012 |
| heapSort | 14703 | 14701 | 14695 | 14701 | 14712 | 14702.4 |
| mergeSort | 19952 | 19952 | 19952 | 19952 | 19952 | 19952 |
| insertSortHalf | 2015 | 2017 | 2012 | 2012 | 2014 | 2014 |

原数据正序的情况下，由上述数据可以验证各排序算法复杂度与比较次数和实际交换次数之间的关系，其比较次数与交换次数的数量级反映了算法的最好情况下的时间复杂度。

正序情况下（数据为1000条）排序效率：

冒泡排序>简单插入排序>折半插入排序>希尔排序>归并排序>堆排序>简单选择排序>快速排序

**3.比较各排序算法排列数组（逆序）下的compare times 和move times**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| compare times | case1 | case2 | case3 | case4 | case5 | average |
| BubbleSort | 499500 | 499500 | 499500 | 499500 | 499500 | 499500 |
| simpleInsert | 500485 | 500488 | 501478 | 501483 | 501480 | 501482.4 |
| simpleSelect | 500499 | 500499 | 500499 | 500499 | 500499 | 500499 |
| quicksort | 500998 | 500998 | 500998 | 500998 | 500998 | 500998 |
| shellSort | 12695 | 12703 | 12693 | 12699 | 12694 | 12696.8 |
| heapSort | 15982 | 15984 | 16004 | 15992 | 15982 | 15988.8 |
| mergeSort | 4932 | 4932 | 4932 | 4932 | 4932 | 4932 |
| insertSortHalf | 7987 | 7987 | 7987 | 7987 | 7987 | 7987 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| movetimes | case1 | case2 | case3 | case4 | case5 | average |
| BubbleSort | 1498458 | 1498467 | 1498440 | 1498455 | 1498446 | 1498453 |
| simpleInsert | 501484 | 501487 | 501478 | 501483 | 501480 | 501482.4 |
| simpleSelect | 2997 | 2997 | 2997 | 2997 | 2997 | 2997 |
| quicksort | 2498 | 2498 | 2498 | 2498 | 2498 | 2498 |
| shellSort | 20701 | 20709 | 20699 | 20705 | 20700 | 20702.8 |
| heapSort | 13312 | 13315 | 13326 | 13316 | 13305 | 13314.8 |
| mergeSort | 19952 | 19952 | 19952 | 19952 | 19952 | 19952 |
| insertSortHalf | 501498 | 501498 | 501498 | 501498 | 501498 | 501498 |

原数据逆序的情况下，由上述数据可以验证各排序算法复杂度与比较次数和实际交换次数之间的关系，其比较次数与交换次数的数量级反映了算法的最坏情况下的时间复杂度。

逆序情况下（数据为1000条）排序效率：

快速排序>简单选择排序>堆排序>归并排序>希尔排序>折半插入排序>插入排序>冒泡排序

**4.由上述三种情况的比较次数与交换次数关系，各算法的时间复杂度可以由下图来表示：**

|  |  |  |  |
| --- | --- | --- | --- |
| 排序方法 | | 平均时间 | 最坏情况 |
| 插入排序 | 直接插入排序 | O（n2） | O（n2） |
|
| 折半插入排序 | O（n2） | O（n2） |
| 希尔排序 | O（n3/2） | O（n2） |
| 选择排序 | 直接选择排序 | O（n2） | O（n2） |
| 堆排序 | O（nlog2n） | O（nlog2n） |
| 交换排序 | 冒泡排序 | O（n2） | O（n2） |
| 快速排序 | O（nlog2n） | O（n2） |
| 归并排序 | 2-路归并排序 | O（nlog2n） | O（nlog2n） |
|

**5.关于数据波动性的解释**

1.在乱序情况下，数组的分布不同直接会直接导致排序比较次数和交换次数出现较大差异，从实验结果来看各次实验的误差均在可接受范围内，数据波动属于正常现象；

2.在正序和逆序情况下，数组分布近似相同，但其中存在的相同数字会对稳定性较差的算法产生一定影响，如对快速排序中的划分，极大堆对堆排序的过滤等均有一定影响，导致数据在小范围内波动，属于正常现象。

**6.比较各排序算法不同表长排列数组（随机）下的compare times 和move times**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| comparetimes | 100 | 500 | 1000 | 2000 | 5000 | 10000 | 50000 |
| BubbleSort | 4895 | 124705 | 499175 | 1997289 | 12493405 | 49984269 | 1249909659 |
| simpleInsert | 2419 | 66475 | 254514 | 992127 | 6255630 | 25288698 | 622988017 |
| simpleSelect | 5049 | 125249 | 500499 | 2000999 | 12502499 | 50004999 | 1250024999 |
| quicksort | 710 | 5085 | 11166 | 26770 | 74273 | 167999 | 994303 |
| shellSort | 877 | 6445 | 15597 | 37861 | 116651 | 266134 | 1880149 |
| heapSort | 1038 | 7414 | 16876 | 37666 | 107712 | 235340 | 1409884 |
| mergeSort | 551 | 3858 | 8680 | 19433 | 55250 | 120542 | 718262 |
| insertSortHalf | 533 | 3788 | 8571 | 19173 | 54543 | 119013 | 711203 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| movetimes | 100 | 500 | 1000 | 2000 | 5000 | 10000 | 50000 |
| BubbleSort | 7140 | 191862 | 728940 | 3020757 | 18532923 | 74926746 | 1879796736 |
| simpleInsert | 2578 | 64952 | 258523 | 1010917 | 6187639 | 24995580 | 626698910 |
| simpleSelect | 297 | 1497 | 2997 | 5997 | 14997 | 29997 | 149997 |
| quicksort | 372 | 2378 | 5174 | 11160 | 30980 | 66779 | 380938 |
| shellSort | 1374 | 10129 | 23989 | 55595 | 169878 | 409076 | 2584823 |
| heapSort | 1090 | 6540 | 14086 | 30190 | 82142 | 174102 | 987524 |
| mergeSort | 1344 | 8976 | 19952 | 43904 | 123616 | 267232 | 1568928 |
| insertSortHalf | 2578 | 64955 | 258535 | 1010959 | 618803 | 24997065 | 626736924 |

结论① 在对随机数据排序时，我们可以发现随数据条数增加，各算法的比较次数统计量和交换次数统计量呈不同的比例增加，与其算法复杂度有一定的关系。

结论② 综合两项指标我们可以给出排序效率在不同情况下的最优选择：

数据量小时：快速排序>堆排序>归并排序>希尔排序

数据量大时：快速排序>堆排序>归并排序

结论③ 简单选择在减少数据交换上有奇效

简单插入在数据量较少时也有奇效

结论④ 稳定性：

综合对正逆序数据排序及不同表长数据的排序结果：

稳定性较好的算法有：直接插入 折半插入 冒泡排序 归并排序

稳定性较差的算法有：快速排序 直接选择排序 希尔排序 堆排序

**七、 附录**

main.cpp //主函数 功能函数模块

paixu.h //头文件 定义函数