内部排序算法比较课程设计

1. 问题描述

排序是计算机经常进行的一种操作。排序算法有直接插入排序，快速排序，选择排序，希尔排序，堆排序等等。排序算法的效率是影响计算机性能的重要因素。在教科书中，各种排序算法只给了执行时间，在这里将比较各个排序算法的比较和移动次数，对各种排序算法有更直观的感受。

1. 程序设计

（1）变量说明

全局变量cmp,mov记录排序算法的比较和移动次数，tmp数组在归并算法中临时储存关键字，is\_all用来在下文中判断是否对所有算法运行，并且记录在对所有算法运行时储存要测试的数据个数，mem二维数组用来记录在对所有算法运行时的比较和移动次数。

typedef struct

{

int length;

int data[200000];

}SqList;

SqList 同来储存要排序的表，其中Length表示长度，Data数组储存数据。

（2）函数说明

generate(SqList \*L)用来生成随机数并储存在L中

print(SqList \*L) 用来打印L

reverse(SqList \*L) 将列表转成逆序

以下十个函数用来实现这十个排序算法

BubbleSort(SqList \*L); 冒泡

QuickSort(SqList \*L); 快速排序

InsertSort(SqList \*L); 直接插入

SelectSort(SqList \*L); 选择排序

shellSort(SqList \*L); 希尔排序

HeapSort(SqList \*L); 堆排序

MergeSort(SqList \*L); 归并排序

BInsertSort(SqList \*L); 折半插入排序

Path2Insert(SqList \*L); 二路插入排序

radixsort(SqList \*L); 基数排序

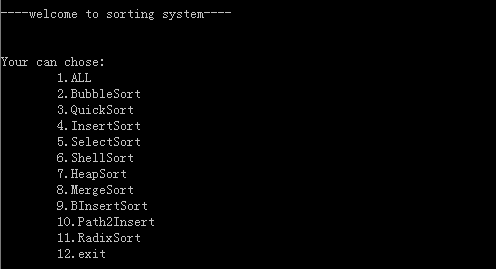
All（SqList \*L）

用来封装这十个函数并且统计正序和逆序的结果。

MAIN函数用一个while循环询问用户是否需要进行哪种服务，并调用相应的函数。如果为12则退出循环，并退出程序。

1. 使用方法

运行程序后界面如下



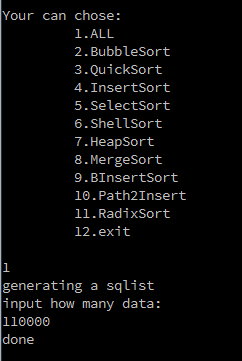
输入1用所有算法对表进行排序

输入2~11对单个算法进行排序

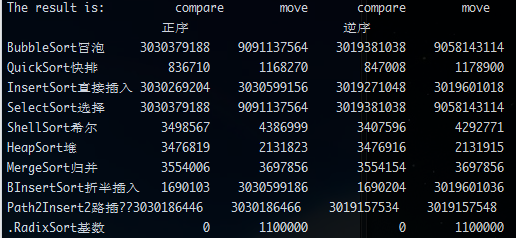
输入12退出

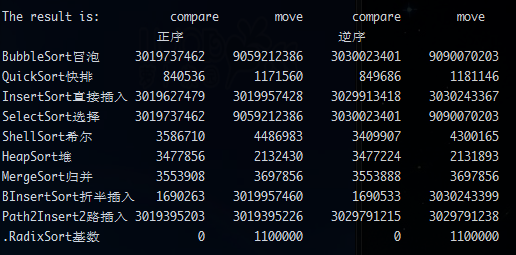
1. 同时对多个算法进行比较

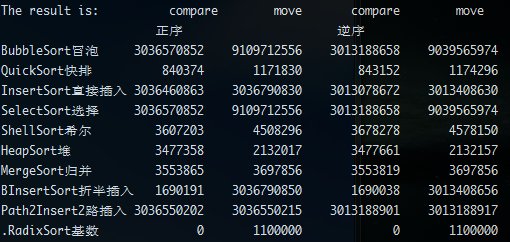
运行程序，输入1对所有算法进行测试，再输入110000对110000个数据进行测试。

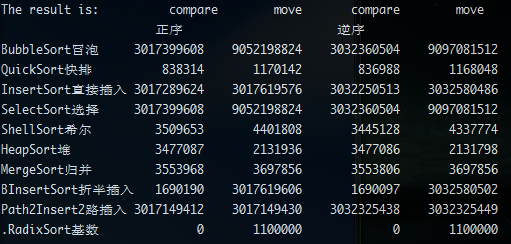


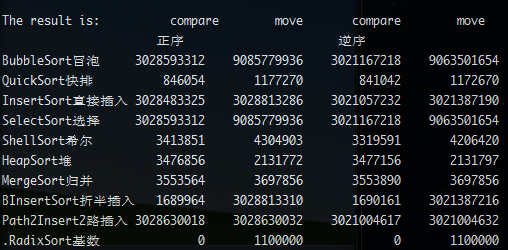
连续测试测试5次，结果如下











1. 数据处理

对五次正序和逆序的结果求和取平均如下：

|  |  |  |
| --- | --- | --- |
| 算法 | 比较次数 | 移动次数 |
| 冒泡 | 3024880124 | 9074640372 |
| 快排 | 841986 | 1173413 |
| 直接插入 | 3024770137 | 3025100096 |
| 选择 | 3024880124 | 9074640372 |
| 希尔 | 3487648 | 4380426 |
| 堆 | 3477201 | 2131953 |
| 归并 | 3553886 | 3697856 |
| 折半插入 | 1690174 | 3025100122 |
| 2路插入 | 3024737898 | 3024737915 |
| 基数 | 0 | 1100000 |
|  |  |  |

结论：

由上表可以看出，冒泡、直接插入、选择，二路插入的比较和移动次数最多，快排的比较和移动次数最少。基数排序比较次数为0，移动次数也较少。

除了基数排序与其无关，比较次数为0，移动次数是数据个数的十倍。

结论：

从以上图表可以看出，希尔排序的波动性最大，快速排序波动性较大，冒泡排序、选择排序、堆排序、基数排序、归并排序的波动性最小，二路插入排序、折半插入排序、直接插入排序算法波动性波动性较小

分析：

希尔排序，当排序序列为正序时，比较次数为n，移动次数为0，当为逆序时，移动次数增加，复杂度增加，因此波动性较大。

快速排序最好的情况是正序序列，只需一趟排序，有n-1次比较，不移动数据，最差需进行n-1趟排序，进行n(n-1)/2次比较，并移动n(n-1)/2次，因此快速排序的波动性很大。

冒泡排序，选择排序比较次数相同，均为n(n-1)/2，移动次数变化也趋于固定值，所以冒泡和选择排序波动性较小。

基数排序不进行比较，移动次数与关键字的个数有关，为d\*n,移动次数固定,故波动性较小。

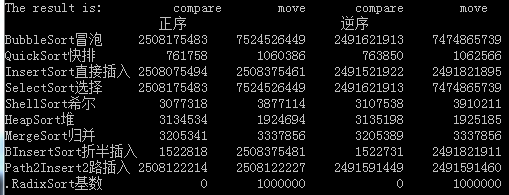
归并排序：MSort函数中递归将表平分，次数是固定的，一趟归并排序的操作是调用n/2h次算法merge,将两个有序段两两归并，并且在Merge算法中比较和移动的次数比较固定。

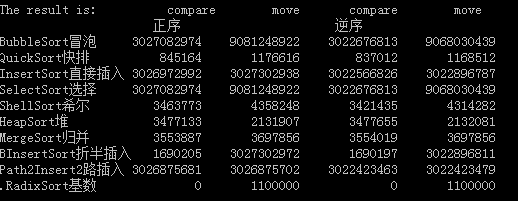
堆排序相对于快速排序在最坏的情况下，比较次数不超过2n(log2(n))，所以堆排序波动性较小。

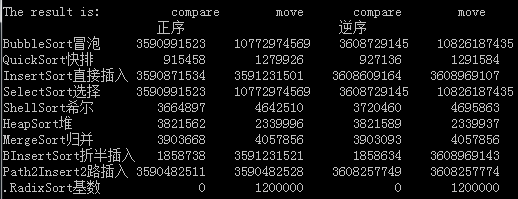
直接插入排序、2路插入排序、折半插入排序为正序时移动次数为0，比较次数小于等于n，逆序时在向后移动时复杂度增加。波动性较大。

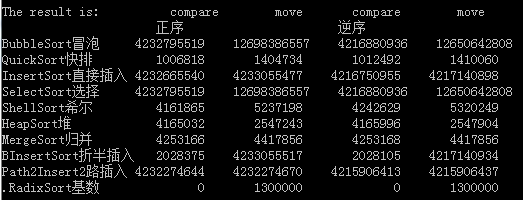
六． 不同表长对实验结果的影响

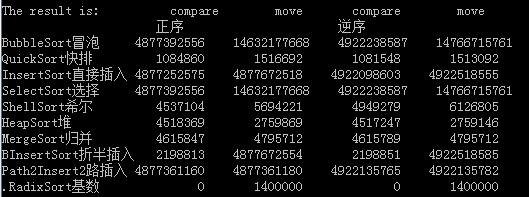
分别用表长为100000、110000、120000、130000、140000的数据进行测试，结果如下：











结论:从以上图表来看，除了希尔排序波动比较大，其他排序算法与要比较的数据量成大致成正比。

1. 附源代码

#include<stdio.h>

#include<stdlib.h>

#include<math.h>

long long int cmp;

long long int mov;

int tmp[200000];

int is\_all;

long long int mem[10][4];

typedef struct

{

int length;

int data[200000];

}SqList;

void generate(SqList \*L)

{

int i,n;

int r;

if(is\_all == 0)

{

puts("input how many data:");

scanf("%d", &n);

}

else

n = is\_all;

srand((unsigned)time(NULL));

L->length = n;

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

{

r = rand();

L->data[i] = r;

}

}

void print(SqList \*L)

{

int i;

char c;

if( !is\_all)

{

puts("print the Sqlist? <Y/N>");

scanf("%c", &c);

scanf("%c", &c);

if(c == 'y' || c == 'Y')

{

puts("now the SqList looks like");

for(i=1 ; i<L->length; i++)

printf("%d\n", L->data[i]);

}

else if(c == 'n' || c == 'N')

;

else

puts("invalid input");

}

}

void reverse(SqList \*L)

{

int n = L->length;

int i;

int tmp;

for(i = 1; i<n/2; i++)

{

tmp = L->data[i];

L->data[i] = L->data[n+1-i];

L->data[n+1-i] = tmp;

}

}

//-------------------------bubble-----------------------

void BubbleSort(SqList \*L)

{

int i,j;

int tmp;

char c;

puts("\nBubble sorting");

if( is\_all == 0 )

{

puts("generating a sqlist");

generate(L);

puts("done");

}

puts("sorting");

mov = 0; cmp = 0;

for (i=1; i<=L->length; i++)

{

for( j = L->length; j > i; j--)

{

if(L->data[j-1] > L->data[j] )

{

cmp ++;

tmp = L->data[j-1];

L->data[j-1] = L ->data[j];

L->data[j] = tmp;

mov += 3;

}

}

}

puts("done");

printf("total compare :%lld, total move :%lld\n", cmp, mov);

print(L);

}

//-----------------------Quicksort------------------------

int Partition(SqList \*L, int a\_low, int a\_high)

{

int pivotkey;

int low = a\_low,high = a\_high;

L->data[0] = L->data[low];

mov ++;

pivotkey = L->data[low];

mov ++;

while(low<high)

{

while(low<high && L->data[high] >= pivotkey) --high;

cmp ++;

L->data[low] = L->data[high];

mov ++;

while(low<high && L->data[low] <= pivotkey) ++low;

cmp ++;

L->data[high] = L->data[low];

mov ++;

}

L->data[low] = L->data[0];

mov ++;

return low;

}

void QSort(SqList \*L, int low, int high)

{

int pivotloc;

if(low < high)

{

pivotloc = Partition(L, low, high);

mov++;

QSort(L, low, pivotloc - 1);

QSort(L, pivotloc + 1, high);

}

}

void QuickSort(SqList \*L)

{

char c;

puts("\nQuick sorting");

if( is\_all == 0 )

{

puts("generating a sqlist");

generate(L);

puts("done");

}

puts("sorting");

cmp = 0; mov = 0;

QSort(L, 1, L->length);

puts("done");

printf("total compare :%lld, total move :%lld\n", cmp, mov);

print(L);

}

//--------------------insertsort----------------------

void InsertSort(SqList \*L)

{

char c;

cmp = 0; mov = 0;

puts("\ninsert sorting");

puts("sorting");

int i,j;

for (i=2; i<=L->length; i++)

{

if(L->data[i]<L->data[i - 1])

{

L->data[0] = L->data[i];

L->data[i] = L->data[i-1]; //move\_last

mov += 2;

for ( j = i - 2; L->data[0] < L->data[j]; j--)

{

cmp++;

L->data[j+1] = L->data[j];

mov++;

}

L->data[j+1] = L->data[0];

mov++;

}

}

puts("done");

printf("total compare :%lld, total move :%lld\n", cmp, mov);

print(L);

}

//---------------------select sorting-----------------------

void SelectSort(SqList \*L)

{

char c;

cmp = 0; mov = 0;

puts("\nselect sorting");

if( is\_all == 0 )

{

puts("generating a sqlist");

generate(L);

puts("done");

}

puts("sorting");

int i,j;

for (i=1; i<=L->length; i++)

{

int s\_i = i;

for(j = i+1; j <= L->length; j++) //select min

{

if(L->data[s\_i] > L->data[j] )

{

int tmp = L->data[i];

L->data[i] = L->data[s\_i];

L->data[s\_i] = tmp;

cmp++;

mov +=3;

}

}

}

puts("done");

printf("total compare :%lld, total move :%lld\n", cmp, mov);

print(L);

}

//--------------------------shell sorting------------------

void shellinsert(SqList \*L, int dk)

{

int i,j;

for( i= dk +1; i<=L->length; i++)

{

if(L->data[i] < L->data[i-dk])

{

cmp++;

L->data[0] = L->data[i];

mov++;

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

{

cmp ++;

L->data[j+dk] = L->data[j];

mov++;

}

L->data[j+dk] = L->data[0];

mov++;

}

}

}

void shellSort(SqList \*L)

{

int t = log(110000)/log(2);

int dlta[50];

int i,k;

char c;

cmp = 0; mov = 0;

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

dlta[i] = pow(2, t - i) + 1;

puts("\nshell sorting");

if( is\_all == 0 )

{

puts("generating a sqlist");

generate(L);

puts("done");

}

puts("sorting");

for(k = 0; k<t; k++)

{

shellinsert(L,dlta[k]);

}

puts("done");

printf("total compare :%lld, total move :%lld\n", cmp, mov);

print(L);

}

//---------------------heap sort-------------------------------

void HeapAdjust(SqList \*L, int s, int m)

{

int rc,j;

rc = L->data[s];

mov++;

for( j = 2\*s; j<=m; j\*=2)

{

if(j<m && L->data[j] < L->data[j+1] ) ++j;

cmp ++;

if( rc > L->data[j] ) break;

cmp ++;

L->data[s] = L->data[j]; s = j;

mov++;

}

L->data[s] = rc;

cmp++;

}

void HeapSort(SqList \*L)

{

int i;

char c;

cmp = 0; mov = 0;

puts("done");

puts("\nheap sorting");

if( is\_all == 0 )

{

puts("generating a sqlist");

generate(L);

puts("done");

}

puts("sorting");

for( i = L->length/2; i>0; i--)

HeapAdjust(L, i , L->length);

for( i = L->length; i>1; i--)

{

int tmp = L->data[1];

L->data[1] = L->data[i];

L->data[i] = tmp;

mov += 3;

HeapAdjust(L, 1, i-1);

}

puts("done");

printf("total compare :%lld, total move :%lld\n", cmp, mov);

print(L);

}

//----------------------------------------merge---------------------------------------

void Merge(int \*array, int i, int m, int n)

{

int j,k,l;

l = i;

for( j = m+1, k = i;l<=m && j<=n; k++)

{

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

{

tmp[k] = array[l++];

}

else

{

tmp[k] = array[j++];

}

cmp++;

mov++;

}

while( l<= m)

{

tmp[k] = array[l++];

k++;

mov++;

}

while( j<= n)

{

tmp[k] = array[j++];

k++;

mov++;

}

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

{

array[l] = tmp[l];

cmp++;

mov++;

}

}

void MSort(int \*array, int s, int t)

{

int m;

if(s == t)

{

;

}

else

{

m = (s+t)/2;

MSort(array, s, m);

MSort(array, m+1, t);

Merge(array, s, m, t);

}

}

void MergeSort(SqList \*L)

{

char c;

cmp = 0; mov = 0;

puts("\nMerge sorting");

if( is\_all == 0 )

{

puts("generating a sqlist");

generate(L);

puts("done");

}

puts("sorting");

MSort(L->data, 1, L->length);

puts("done");

printf("total compare :%lld, total move :%lld\n", cmp, mov);

print(L);

}

//---------------------------------BInsertSort------------------------------

void BInsertSort( SqList \*L)

{

int i,j,m,high,low;

char c;

cmp = 0; mov = 0;

puts("\nBInsert sorting");

if( is\_all == 0 )

{

puts("generating a sqlist");

generate(L);

puts("done");

}

puts("sorting");

for(i=2; i<=L->length; i++)

{

L->data[0] = L->data[i];

mov++;

low = 1;

high = i-1;

while(low <= high)

{

m = (low + high)/2;

if(L->data[0] < L->data[m])

high = m-1;

else

low = m + 1;

cmp++;

}

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

{

L->data[j+1] = L->data[j];

mov++;

}

L->data[high+1] = L->data[0];

mov++;

}

puts("done");

printf("total compare :%lld, total move :%lld\n", cmp, mov);

print(L);

}

//-----------------------------Path2Insert ----------------------------------

void Path2Insert (SqList \*L)

{

int i,k;

char c;

cmp = 0; mov = 0;

puts("\nPath2Insert sorting");

if( is\_all == 0 )

{

puts("generating a sqlist");

generate(L);

puts("done");

}

int length = L->length - 1;

int d[length];

puts("sorting");

int final = 0;

int first = 0;

int temp[length];

temp[0] = L->data[0];

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

{

if(L->data[i] > temp[final])

{

cmp++;

final++;

temp[final] = L->data[i];

mov++;

}

if(L->data[i]< temp[first])

{

cmp++;

first = (first-1+length)%length;

temp[first] = L->data[i];

mov++;

}

if(L->data[i] < temp[final]&&L->data[i] > temp[first])

{

cmp++;

cmp++;

int j = final++;

while (L->data[i] < temp[j])

{

cmp++;

temp[(j+1)%length] = temp[j];

j = (j-1+length)%length;

mov++;

}

temp[j+1] = L->data[i];

mov++;

}

}

for (k = 0; k < length; k++)

{

mov++;

L->data[k+1] = temp[(first++)%length];

}

puts("done");

printf("total compare :%lld, total move :%lld\n", cmp, mov);

print(L);

}

//------------------------Radix----------------------------

int maxbit(int data[],int n)

{

int d = 1;

int p =10;

int i;

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

{

while(data[i] >= p)

{

cmp++;

p \*= 10;

++d;

}

}

return d;

}

void radixsort(SqList \*L)

{

puts("\nRadix sorting");

if( is\_all == 0 )

{

puts("generating a sqlist");

generate(L);

puts("done");

}

int n = L->length;

int \*data = L->data;

int d = maxbit(data,n);

int tmp[n];

int count[10];

int i,j,k;

int radix = 1;

char c;

cmp = 0; mov = 0;

puts("sorting");

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

{

for(j = 0;j < 10;j++) //clear count

count[j] = 0;

for(j = 0;j < n; j++) //write count

{

k = (data[j]/radix)%10;

count[k]++;

}

for(j = 1;j < 10;j++)

count[j] = count[j-1] + count[j];

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

{

k = (data[j]/radix)%10;

tmp[count[k]-1] = data[j];

mov++;

count[k]--;

}

for(j = 0;j < n;j++) //copy to data

{

data[j] = tmp[j];

mov++;

}

radix = radix\*10;

}

puts("done");

printf("total compare :%lld, total move :%lld\n", cmp, mov);

print(L);

}

void All(SqList \*L)

{

int n;

int i=0;

SqList \*t=(SqList \*)malloc(sizeof(SqList));

puts("generating a sqlist");

puts("input how many data:");

scanf("%d", &is\_all);

generate(L);

puts("done");

\*t=\*L;

int sig = 2;

while(sig !=0)

{

i = 0;

int j = 0;

if(sig == 1)

{

reverse(L);

\*t = \*L;

j = 2;

}

while(i != 10)

{

if(i == 0)

BubbleSort(L);

if(i == 1)

QuickSort(L);

if(i == 2)

InsertSort(L);

if(i == 3)

SelectSort(L);

if(i == 4)

shellSort(L);

if(i == 5)

HeapSort(L);

if(i == 6)

MergeSort(L);

if(i == 7)

BInsertSort(L);

if(i == 8)

Path2Insert(L);

if(i == 9)

radixsort(L);

mem[i][0 + j] = cmp;

mem[i][1 + j]=mov;

i++;

\*L=\*t;

}

sig --;

}

i = 0;

printf("\nThe result is: compare move compare move\n");

printf(" 正序 逆序\n");

printf("BubbleSort冒泡 %10lld %10lld %10lld %10lld \n", mem[i][0], mem[i][1], mem[i][2], mem[i][3]);

i++;

printf("QuickSort快排 %10lld %10lld %10lld %10lld \n", mem[i][0], mem[i][1], mem[i][2], mem[i][3]);

i++;

printf("InsertSort直接插入 %10lld %10lld %10lld %10lld \n", mem[i][0], mem[i][1], mem[i][2], mem[i][3]);

i++;

printf("SelectSort选择 %10lld %10lld %10lld %10lld \n", mem[i][0], mem[i][1], mem[i][2], mem[i][3]);

i++;

printf("ShellSort希尔 %10lld %10lld %10lld %10lld \n", mem[i][0], mem[i][1], mem[i][2], mem[i][3]);

i++;

printf("HeapSort堆 %10lld %10lld %10lld %10lld \n", mem[i][0], mem[i][1], mem[i][2], mem[i][3]);

i++;

printf("MergeSort归并 %10lld %10lld %10lld %10lld \n", mem[i][0], mem[i][1], mem[i][2], mem[i][3]);

i++;

printf("BInsertSort折半插入%10lld %10lld %10lld %10lld \n", mem[i][0], mem[i][1], mem[i][2], mem[i][3]);

i++;

printf("Path2Insert2路插入 %10lld %10lld %10lld %10lld \n", mem[i][0], mem[i][1], mem[i][2], mem[i][3]);

i++;

printf(".RadixSort基数 %10lld %10lld %10lld %10lld \n", mem[i][0], mem[i][1], mem[i][2], mem[i][3]);

i++;

is\_all = 0;

}

int main()

{

SqList L;

int chose;

int i;

is\_all = 0;

puts("\n----welcome to sorting system----");

while(1)

{

puts("\n\nYour can chose: \n\

1.ALL\n\

2.BubbleSort\n\

3.QuickSort\n\

4.InsertSort\n\

5.SelectSort\n\

6.ShellSort\n\

7.HeapSort\n\

8.MergeSort\n\

9.BInsertSort\n\

10.Path2Insert\n\

11.RadixSort\n\

12.exit\n");

scanf("%d", &chose);

if(chose == 1 )

All(&L);

else if(chose == 2)

BubbleSort(&L);

else if(chose == 3)

QuickSort(&L);

else if(chose == 4)

InsertSort(&L);

else if(chose == 5)

SelectSort(&L);

else if(chose == 6)

shellSort(&L);

else if(chose == 7)

HeapSort(&L);

else if(chose == 8)

MergeSort(&L);

else if(chose == 9)

BInsertSort(&L);

else if(chose == 10)

Path2Insert(&L);

else if(chose == 11)

radixsort(&L);

else if(chose == 12)

exit(0);

else

puts("invalid input");

}

}