实验报告电子版

**数据结构实训**

**—— 多维数字信号的排序**

姓 名： 刘远明

学 号： 2220212113

指导老师： 韩凤

实验日期： 2024年4月29日

**大连海事大学电子信息科学技术专业**

**Academic Honesty Violations in Practice of Data Structures (PDS)**

In **PDS**, the MINIMUM penalty recommended for a violation of the Academic Honesty Policy will be a ZERO ON THE ASSIGNMENT, PROJECT or EXAM and a LOWERING OF YOUR FINAL GRADE for below what is otherwise earned. You may NOT withdraw from the course if found guilty. Some examples of academic misconduct in **PDS** include but are not limited to the following actions:

1. Picking up and using or discarding another student's written or computer output;

2. Using the computer account of another student;

3. Representing as one's own the work of another on assignments, quizzes, and projects;

4. Giving another student a copy of one's work on an assignment before the due date.

5. Copying work from online resources (Baidu,Chegg, google forums, etc.)

6. Posting work to online resources where other students can view your work.

All submissions will be checked for similarity. This code will check each submission for similarity between other student submissions, past student submissions, the solution manual, and online resources and postings. If your submission is flagged for a high level of similarity, it will be turned in for an academic honesty violation if deemed appropriate.

NOTE: Changing variable names, adding comments, or spacing will l result in a violation.

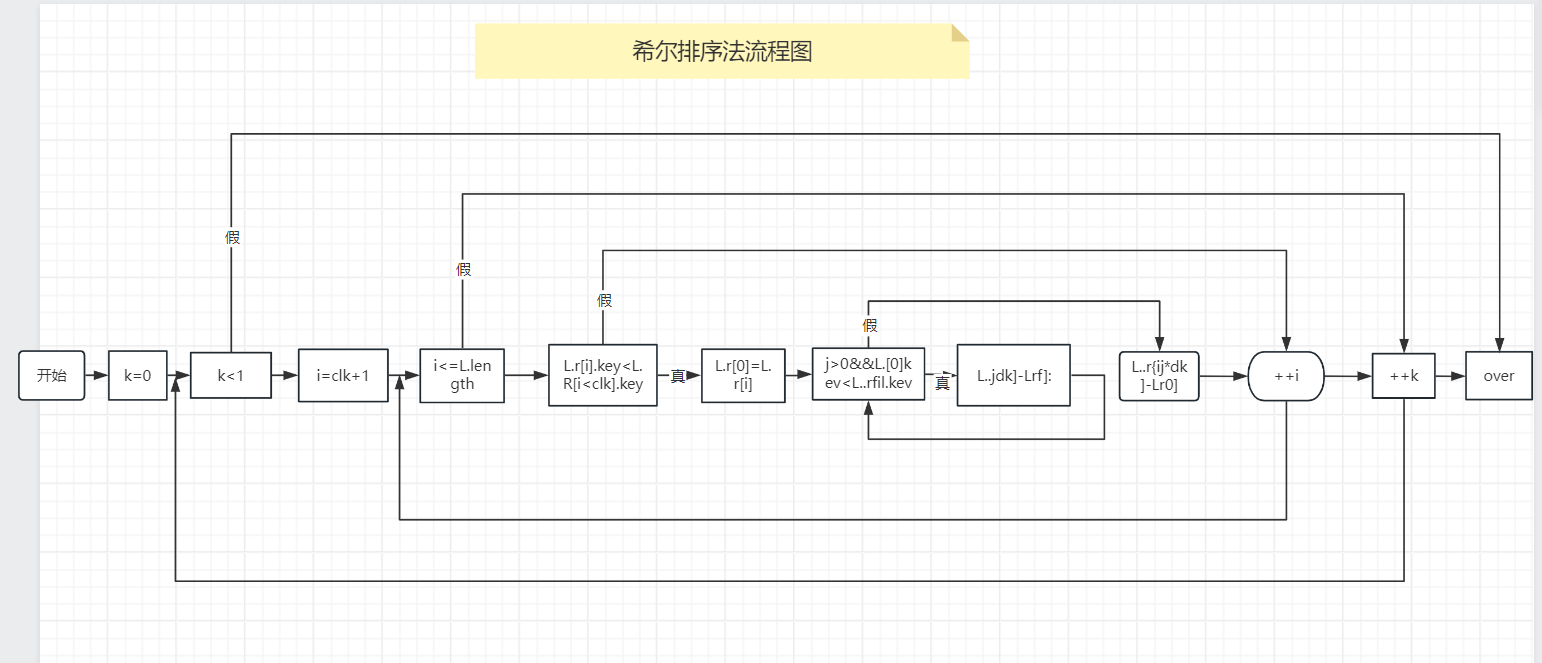
1. **实验目的**
2. 深入理解各种排序的算法思想、方法、稳定性及时间和空间复杂度。
3. 掌握插入排序、交换排序、选择排序的算法实现。
4. 能够实现多维信号的排序操作。
5. 提高实际动手进行程序设计的能力。
6. **实验内容与要求**

**实验内容：**

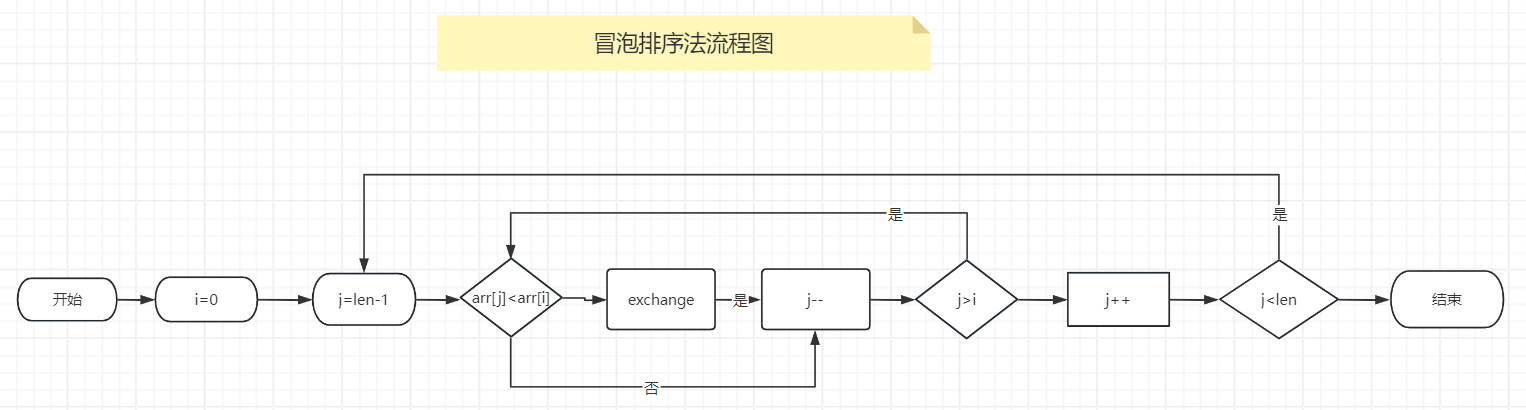
1. 设计函数实现计算多维信号中各元素到参考点的距离（欧几里得距离、街区距离或者余弦距离）
2. 实现对给信号中各元素依据1）中计算的距离进行排序，排序算法分别为冒泡排序、快速排序、堆排序、归并排序；
3. 比较分析冒泡排序、速排序、堆排序、归并四种算法的优缺点和适用范围。

**实验要求：**

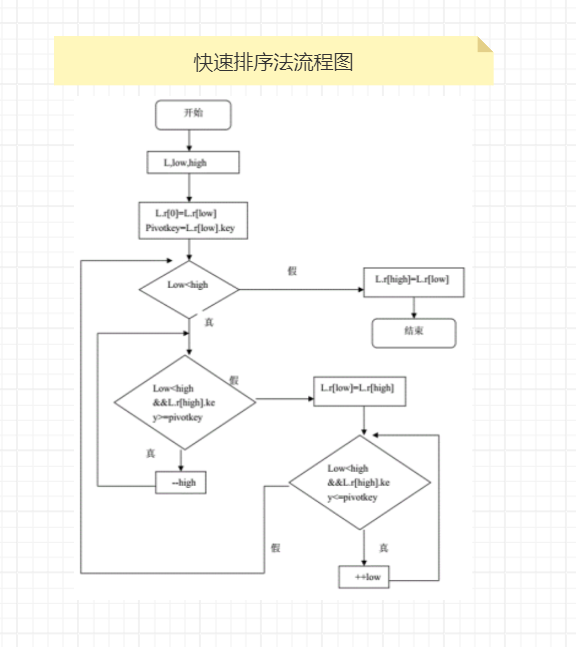
1. 遵守工程伦理和学术规范。
2. 解决的问题要来源于实际工程：要对日常生活中的一个具体应用进行抽象建模与求解，鼓励对国家重大工程中的一个具体点进行抽象建模与求解并融入工程管理理念。
3. 深入理解各种排序的算法思想、方法、稳定性及时间和空间复杂度，要对各种方案进行分析比较。
4. 要对实验结果进行和解释，并通过信息综合得到合理有效地结论。
5. **算法描述与流程**
6. 希尔排序法流程图



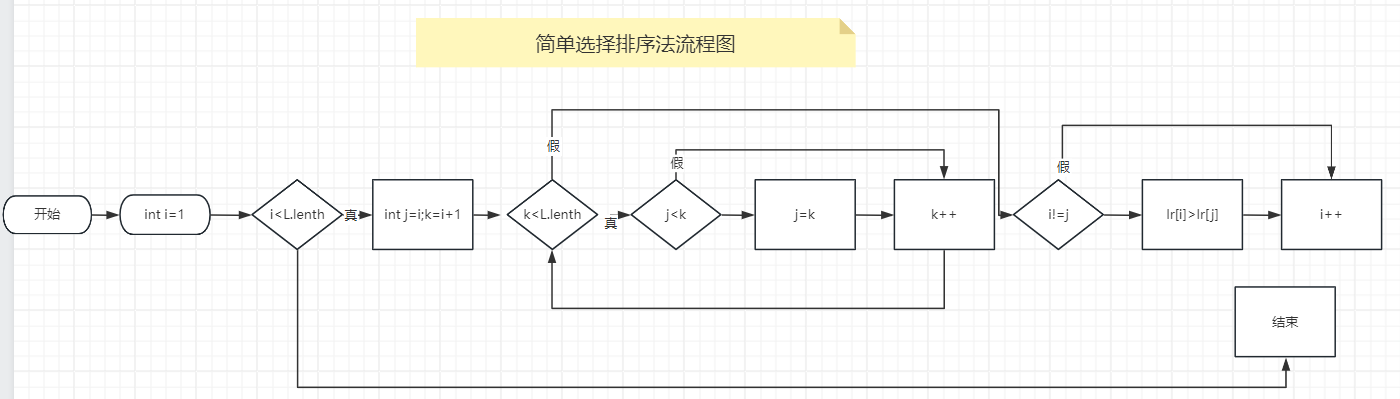
1. 冒泡排序法流程图



1. 快速排序法流程图



1. 简单选择排序法流程图



本次实验使用了希尔排序法、冒泡排序法、快速排序法、选择排序法对随机生成的数据集进行排序。随机生成的数据集的关键字值使用余弦距离来计算。

1. **算法实现**

**初始定义**

**#include <stdio.h>**

**#include "malloc.h"**

**#include "math.h"**

**#include <stdlib.h> //srand()、rand()所需**

**#include <time.h> //time()所需**

**#include <string.h>**

**#define MAXL 100 //最大长度**

**#define LIMIT 200 //生成随机数的范围**

**#define NUM 200 //数据点个数**

**typedef float KeyType; //定义关键字类型为int**

**typedef int InfoType;**

**typedef struct**

**{**

**KeyType key; //关键字项**

**InfoType ID; //其他数据项，类型为InfoType**

**float Item1;**

**float Item2;**

**float Item3;**

**} RecType; //查找元素的类型**

**typedef struct NDLNode**

**{**

**KeyType key; //关键字项**

**InfoType ID; //其他数据项，类型为InfoType**

**float Item1;**

**float Item2;**

**float Item3;**

**struct NDLNode\* next; //指向后继结点**

**} NDLinkNode;**

**计时专用函数：**

**void RandomSeed(); //生成随机数并写入文件中（初始化数据）**

**double InsertSort\_Count(RecType R[], int n); //直接插入排序的计时版本**

**double BinInsertSort\_Count(RecType R[], int n); //折半插入排序计时版本**

**double ShellSort\_Count(RecType R[], int n); //希尔排序计时版本**

**double BubbleSort\_Count(RecType R[], int n); //冒泡排序计时版本**

**double QuickSort\_Count(RecType R[], int s, int t);//快速排序计时版本**

**double SelectSort\_Count(RecType R[], int n); //选择排序计时版本**

**double MergeSort\_Count(RecType R[], int n); //二路归并排序计时版本**

**排序函数：**

**void InsertSort(RecType R[], int n); //对R[0..n-1]按递增有序进行直接插入排序**

**void BinInsertSort(RecType R[], int n); //对R[0..n-1]按递增有序进行折半插入排序**

**void ShellSort(RecType R[], int n); //对R[0..n-1]按递增有序进行希尔排序**

**void BubbleSort(RecType R[], int n); //对R[0..n-1]按递增有序进行冒泡排序**

**void disppart(RecType R[], int s, int t);//显示一趟划分后的结果**

**int partition(RecType R[], int s, int t);//一趟划分**

**void QuickSort(RecType R[], int s, int t); //对R[s..t]的元素进行递增快速排序**

**void SelectSort(RecType R[], int n);//简单选择排序算法**

**/\*两路归并排序\*/**

**void Merge(RecType R[], int low, int mid, int high);**

**void MergePass(RecType R[], int length, int n); //实现一趟归并**

**void MergeSort(RecType R[], int n); //二路归并排序算法;**

**计算距离（计算Key值）算法**

**RecType\* ReadNDSignalFromFile(const char\* FileName,int \*nNum,int nFlag)**

**{**

**RecType\* signal;**

**int D = 3;//数据维数**

**FILE\* fps, \* fpr;**

**fopen\_s(&fps,FileName, "r");**

**NDLinkNode\* h;**

**NDInitList(h);**

**int i = 0;**

**KeyType e[3],key=0;**

**int ID;**

**while (!feof(fps))**

**{**

**fscanf\_s(fps, "%d,", &ID, sizeof(float));**

**if(nFlag)**

**fscanf\_s(fps, "%f,", &key, sizeof(KeyType));**

**fscanf\_s(fps, "%f,", &e[0], sizeof(KeyType));**

**fscanf\_s(fps, "%f,", &e[1], sizeof(KeyType));**

**fscanf\_s(fps, "%f\n", &e[2], sizeof(KeyType));**

**NDListAppend(h,e,ID,nFlag,key);**

**}**

**fclose(fps);**

**//printf(" (3)输出单链表h:\n");**

**//NDDispList(h);**

**KeyType AnchorPoint[3];**

**AnchorPoint[0] = 0; AnchorPoint[1] = 0; AnchorPoint[2] = 0;**

**int nPointNum = NDListLength(h);**

**signal = (RecType\*)malloc(sizeof(RecType) \* nPointNum);**

**//float\* data = (float\*)malloc(sizeof(float) \* nPointNum);**

**NDLinkNode\* tmpp = h->next; //p指向头结点,n置为0(即头结点的序号为0)**

**float tmpD,tmpH = 0;**

**int k = 0;**

**while (tmpp != NULL)**

**{**

**tmpD = 0;**

**if (nFlag==1)**

**signal[k].key = tmpp->key;**

**else**

**{**

**//for (i = 0; i < tmpp->D; i++)**

**//余弦距离**

**tmpH = AnchorPoint[0] \* tmpp->Item1 + AnchorPoint[1] \* tmpp->Item2 + AnchorPoint[2] \* tmpp->Item3;**

**tmpD = (tmpp->Item1 \* tmpp->Item1 + tmpp->Item2 \* tmpp->Item2 + tmpp->Item3 \* tmpp->Item3) \* (AnchorPoint[0] \* AnchorPoint[0] + AnchorPoint[1] \* AnchorPoint[1] + AnchorPoint[2] \* AnchorPoint[2]);**

**signal[k].key = tmpH / sqrt(tmpD);**

**//欧几里得距离**

**//tmpD = (AnchorPoint[0] - tmpp->Item1) \* (AnchorPoint[0] - tmpp->Item1)+ (AnchorPoint[1] - tmpp->Item2) \* (AnchorPoint[1] - tmpp->Item2)+ (AnchorPoint[2] - tmpp->Item3) \* (AnchorPoint[2] - tmpp->Item3);**

**//signal[k].key = sqrt(tmpD); }**

**signal[k].ID = tmpp->ID;**

**signal[k].Item1 = tmpp->Item1;**

**signal[k].Item2 = tmpp->Item2;**

**signal[k].Item3 = tmpp->Item3;**

**k++;**

**tmpp = tmpp->next;**

**}**

**\*nNum = k;**

**NDDestroyList(h);**

**return signal;**

**}**

**排序具体算法**

**void InsertSort(RecType R[],int n) //对R[0..n-1]按递增有序进行直接插入排序**

**{ int i, j; RecType tmp;**

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

**{**

**printf(" i=%d，插入%f，插入结果: \n",i,R[i].key);**

**if (R[i].key<R[i-1].key) //反序时**

**{ tmp=R[i];**

**j=i-1;**

**do //找R[i]的插入位置**

**{ R[j+1]=R[j]; //将关键字大于R[i].key的记录后移**

**j--;**

**} while (j>=0 && R[j].key>tmp.key);**

**R[j+1]=tmp; //在j+1处插入R[i]**

**}**

**DispList(R,n,0);**

**}**

**}**

**double InsertSort\_Count(RecType R[], int n) //直接插入排序的计时版本**

**{**

**int i, j; RecType tmp;**

**clock\_t start, finish; //定义第一次调用CPU时钟单位的实际，可以理解为定义一个计数器**

**double Total\_time; //定义一个double类型的变量，用于存储时间单位**

**start = clock();**

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

**{**

**if (R[i].key < R[i - 1].key) //反序时**

**{**

**tmp = R[i];**

**j = i - 1;**

**do //找R[i]的插入位置**

**{**

**R[j + 1] = R[j]; //将关键字大于R[i].key的记录后移**

**j--;**

**} while (j >= 0 && R[j].key > tmp.key);**

**R[j + 1] = tmp; //在j+1处插入R[i]**

**}**

**}**

**finish = clock();**

**Total\_time = (double)(finish - start) / CLOCKS\_PER\_SEC;**

**return Total\_time;**

**}**

**void BinInsertSort(RecType R[], int n) //对R[0..n-1]按递增有序进行折半插入排序**

**{**

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

**RecType tmp;**

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

**{**

**if (R[i].key < R[i - 1].key) //反序时**

**{**

**printf(" i=%d，插入%f，插入结果: ", i, R[i].key);**

**tmp = R[i]; //将R[i]保存到tmp中**

**low = 0; high = i - 1;**

**while (low <= high) //在R[low..high]中查找插入的位置**

**{**

**mid = (low + high) / 2; //取中间位置**

**if (tmp.key < R[mid].key)**

**high = mid - 1; //插入点在左半区**

**else**

**low = mid + 1; //插入点在右半区**

**} //找位置high**

**for (j = i - 1; j >= high + 1; j--) //集中进行元素后移**

**R[j + 1] = R[j];**

**R[high + 1] = tmp; //插入tmp**

**}**

**DispList(R, n,0);**

**}**

**}**

**double BinInsertSort\_Count(RecType R[], int n) //折半插入排序计时版本**

**{**

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

**RecType tmp;**

**clock\_t start, finish; //定义第一次调用CPU时钟单位的实际，可以理解为定义一个计数器**

**double Total\_time; //定义一个double类型的变量，用于存储时间单位**

**start = clock();**

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

**{**

**if (R[i].key < R[i - 1].key) //反序时**

**{**

**tmp = R[i]; //将R[i]保存到tmp中**

**low = 0; high = i - 1;**

**while (low <= high) //在R[low..high]中查找插入的位置**

**{**

**mid = (low + high) / 2; //取中间位置**

**if (tmp.key < R[mid].key)**

**high = mid - 1; //插入点在左半区**

**else**

**low = mid + 1; //插入点在右半区**

**} //找位置high**

**for (j = i - 1; j >= high + 1; j--) //集中进行元素后移**

**R[j + 1] = R[j];**

**R[high + 1] = tmp; //插入tmp**

**}**

**}**

**finish = clock();**

**Total\_time = (double)(finish - start) / CLOCKS\_PER\_SEC;**

**return Total\_time;**

**}**

**void ShellSort(RecType R[], int n) //对R[0..n-1]按递增有序进行希尔排序**

**{**

**int i, j, d;**

**RecType tmp;**

**d = n / 2; //增量置初值**

**while (d > 0)**

**{**

**for (i = d; i < n; i++) //对所有组采用直接插入排序**

**{**

**tmp = R[i]; //对相隔d个位置一组采用直接插入排序**

**j = i - d;**

**while (j >= 0 && tmp.key < R[j].key)**

**{**

**R[j + d] = R[j];**

**j = j - d;**

**}**

**R[j + d] = tmp;**

**}**

**printf(" d=%d: ", d); DispList(R, n,0);**

**d = d / 2; //减小增量**

**}**

**}**

**double ShellSort\_Count(RecType R[], int n) //希尔排序计时版本**

**{**

**int i, j, d;**

**RecType tmp;**

**d = n / 2; //增量置初值**

**clock\_t start, finish; //定义第一次调用CPU时钟单位的实际，可以理解为定义一个计数器**

**double Total\_time; //定义一个double类型的变量，用于存储时间单位**

**start = clock();**

**while (d > 0)**

**{**

**for (i = d; i < n; i++) //对所有组采用直接插入排序**

**{**

**tmp = R[i]; //对相隔d个位置一组采用直接插入排序**

**j = i - d;**

**while (j >= 0 && tmp.key < R[j].key)**

**{**

**R[j + d] = R[j];**

**j = j - d;**

**}**

**R[j + d] = tmp;**

**}**

**d = d / 2; //减小增量**

**}**

**finish = clock();**

**Total\_time = (double)(finish - start) / CLOCKS\_PER\_SEC;**

**return Total\_time;**

**}**

**void BubbleSort(RecType R[], int n) //对R[0..n-1]按递增有序进行冒泡排序**

**{**

**int i, j;**

**bool exchange;**

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

**{**

**exchange = false; //一趟前exchange置为假**

**for (j = n - 1; j > i; j--) //归位R[i],循环n-i-1次**

**if (R[j].key < R[j - 1].key) //相邻两个元素反序时**

**{**

**swap(R[j], R[j - 1]); //将这两个元素交换**

**exchange = true; //一旦有交换，exchange置为真**

**}**

**printf(" i=%d: 归位元素%f，排序结果：", i, R[i].key);**

**DispList(R, n, 0);**

**if (!exchange) //本趟没有发生交换，中途结束算法**

**return;**

**}**

**}**

**double BubbleSort\_Count(RecType R[], int n) //对R[0..n-1]按递增有序进行冒泡排序**

**{**

**int i, j;**

**bool exchange;**

**clock\_t start, finish; //定义第一次调用CPU时钟单位的实际，可以理解为定义一个计数器**

**double Total\_time; //定义一个double类型的变量，用于存储时间单位**

**start = clock();**

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

**{**

**exchange = false; //一趟前exchange置为假**

**for (j = n - 1; j > i; j--) //归位R[i],循环n-i-1次**

**if (R[j].key < R[j - 1].key) //相邻两个元素反序时**

**{**

**swap(R[j], R[j - 1]); //将这两个元素交换**

**exchange = true; //一旦有交换，exchange置为真**

**}**

**if (!exchange) //本趟没有发生交换，中途结束算法**

**break;**

**}**

**finish = clock();**

**Total\_time = (double)(finish - start) / CLOCKS\_PER\_SEC;**

**return Total\_time;**

**}**

**void disppart(RecType R[], int s, int t) //显示一趟划分后的结果**

**{**

**static int i = 1;**

**int j;**

**printf("第%d次划分:", i);**

**for (j = 0; j < s; j++)**

**printf(" ");**

**for (j = s; j <= t; j++)**

**printf("%.1f,", R[j].key);//只打印一位小数**

**printf("\n");**

**i++;**

**}**

**int partition(RecType R[], int s, int t) //一趟划分**

**{**

**int i = s, j = t;**

**RecType tmp = R[i]; //以R[i]为基准**

**while (i < j) //从两端交替向中间扫描,直至i=j为止**

**{**

**while (j > i&& R[j].key >= tmp.key)**

**j--; //从右向左扫描,找一个小于tmp.key的R[j]**

**R[i] = R[j]; //找到这样的R[j],放入R[i]处**

**while (i < j && R[i].key <= tmp.key)**

**i++; //从左向右扫描,找一个大于tmp.key的R[i]**

**R[j] = R[i]; //找到这样的R[i],放入R[j]处**

**}**

**R[i] = tmp;**

**//如果要计时删掉disppart函数**

**disppart(R, s, t);**

**return i;**

**}**

**void QuickSort(RecType R[], int s, int t) //对R[s..t]的元素进行递增快速排序**

**{**

**int i;**

**if (s < t) //区间内至少存在两个元素的情况**

**{**

**i = partition(R, s, t);**

**QuickSort(R, s, i - 1); //对左区间递归排序**

**QuickSort(R, i + 1, t); //对右区间递归排序**

**}**

**}**

**double QuickSort\_Count(RecType R[], int s, int t) //对R[s..t]的元素进行递增快速排序**

**{**

**int i;**

**clock\_t start, finish; //定义第一次调用CPU时钟单位的实际，可以理解为定义一个计数器**

**double Total\_time; //定义一个double类型的变量，用于存储时间单位**

**start = clock();**

**if (s < t) //区间内至少存在两个元素的情况**

**{**

**i = partition(R, s, t);**

**QuickSort(R, s, i - 1); //对左区间递归排序**

**QuickSort(R, i + 1, t); //对右区间递归排序**

**}**

**finish = clock();**

**Total\_time = (double)(finish - start) / CLOCKS\_PER\_SEC;**

**return Total\_time;**

**}**

**void SelectSort(RecType R[], int n) //简单选择排序算法**

**{**

**int i, j, k;**

**for (i = 0; i < n - 1; i++) //做第i趟排序**

**{**

**k = i;**

**for (j = i + 1; j < n; j++) //在当前无序区R[i..n-1]中选key最小的R[k]**

**if (R[j].key < R[k].key)**

**k = j; //k记下目前找到的最小关键字所在的位置**

**if (k != i) //交换R[i]和R[k]**

**swap(R[i], R[k]);**

**printf(" i=%d,选择关键字:%f,排序结果为:", i, R[i].key);**

**DispList(R, n,0); //输出每一趟的排序结果**

**}**

**}**

**double SelectSort\_Count(RecType R[], int n) //简单选择排序算法**

**{**

**int i, j, k;**

**clock\_t start, finish; //定义第一次调用CPU时钟单位的实际，可以理解为定义一个计数器**

**double Total\_time; //定义一个double类型的变量，用于存储时间单位**

**start = clock();**

**for (i = 0; i < n - 1; i++) //做第i趟排序**

**{**

**k = i;**

**for (j = i + 1; j < n; j++) //在当前无序区R[i..n-1]中选key最小的R[k]**

**if (R[j].key < R[k].key)**

**k = j; //k记下目前找到的最小关键字所在的位置**

**if (k != i) //交换R[i]和R[k]**

**swap(R[i], R[k]);**

**}**

**finish = clock();**

**Total\_time = (double)(finish - start) / CLOCKS\_PER\_SEC;**

**return Total\_time;**

**}**

**void Merge(RecType R[], int low, int mid, int high)**

**//一次归并：将两个有序表R[low..mid]和R[mid+1..high]归并为一个有序表R[low..high]中**

**{**

**RecType\* R1;**

**int i = low, j = mid + 1, k = 0; //k是R1的下标,i、j分别为第1、2段的下标**

**R1 = (RecType\*)malloc((high - low + 1) \* sizeof(RecType)); //动态分配空间**

**while (i <= mid && j <= high) //在第1段和第2段均未扫描完时循环**

**if (R[i].key <= R[j].key) //将第1段中的记录放入R1中**

**{**

**R1[k] = R[i];**

**i++; k++;**

**}**

**else //将第2段中的记录放入R1中**

**{**

**R1[k] = R[j];**

**j++; k++;**

**}**

**while (i <= mid) //将第1段余下部分复制到R1**

**{**

**R1[k] = R[i];**

**i++; k++;**

**}**

**while (j <= high) //将第2段余下部分复制到R1**

**{**

**R1[k] = R[j];**

**j++; k++;**

**}**

**for (k = 0, i = low; i <= high; k++, i++) //将R1复制回R中**

**R[i] = R1[k];**

**}**

**int Mcount = 1; //全局变量**

**void MergePass(RecType R[], int length, int n) //实现一趟归并**

**{**

**int i;**

**printf("第%d趟归并:", Mcount++);**

**for (i = 0; i + 2 \* length - 1 < n; i = i + 2 \* length) //归并length长的两相邻子表**

**{**

**printf("R[%d,%d]和R[%d,%d]归并 ", i, i + length - 1, i + length, i + 2 \* length - 1);**

**Merge(R, i, i + length - 1, i + 2 \* length - 1);**

**}**

**if (i + length - 1 < n - 1) //余下两个子表,后者长度小于length**

**{**

**printf("\*R[%d,%d]和R[%d,%d]归并 ", i, i + length - 1, i + length, n - 1);**

**Merge(R, i, i + length - 1, n - 1); //归并这两个子表**

**}**

**printf("\n归并结果：");**

**DispList(R, n,0); //输出该趟的排序结果**

**}**

**void MergeSort(RecType R[], int n) //二路归并排序算法**

**{**

**int length;**

**for (length = 1; length < n; length = 2 \* length)**

**MergePass(R, length, n);**

**}**

**double MergeSort\_Count(RecType R[], int n) //二路归并排序算法**

**{**

**int length;**

**clock\_t start, finish; //定义第一次调用CPU时钟单位的实际，可以理解为定义一个计数器**

**double Total\_time; //定义一个double类型的变量，用于存储时间单位**

**start = clock();**

**for (length = 1; length < n; length = 2 \* length)**

**MergePass(R, length, n);**

**finish = clock();**

**Total\_time = (double)(finish - start) / CLOCKS\_PER\_SEC;**

**return Total\_time;**

**}**

**5、实验结果与分析**

**5.1测试用例**

测试程序形式为：

main()

{

//初始化现场；

//读数据

如：signal=ReadNDSignalFromFile(SourceFileName,nNum, nFlag

//输出读入的数据： R= CloneArr(signal,n,0);printf("排序前: "); DispList(R, n,0);

//计时：start = clock();

//调用排序算法：InsertSort(R, n);

//计时结束： finish = clock();Total\_time = (double)(finish - start) / CLOCKS\_PER\_SEC;printf("插入排序后(耗时=%f秒):\n", Total\_time);

//显示并输出结果：DispList(R, n,0);WriteNDSignalToFile(ResultFileName, R, n,1); free(R);R=NULL;

//调用其他排序类似上述步骤

//清理现场

**}5.2测试程序**

**用于测试排序结果：**

**取消了计时函数，只用来测试排序结果的程序，选用个数为30的数据集**

int main()

{

//srand((float)time(NULL)); //用于生成输入数据文件

//RandomSeed();

clock\_t start, finish; //定义第一次调用CPU时钟单位的实际，可以理解为定义一个计数器

double Total\_time; //定义一个double类型的变量，用于存储时间单位

RecType\* signal=NULL;

int nNum[1] ;

int nFlag=2;//1表示直接读取key，2表示通过计算得到，如实验示例。

const char\* SourceFileName = "d:\\sortinputs.txt";

const char\* ResultFileName = "d:\\sortoutput.txt";

signal=ReadNDSignalFromFile(SourceFileName,nNum, nFlag);//文件格

//式两种，nFlag=1表示key值已有，格式为ID号，Key值，数据项；nFlag=0表示没有key值，需要计算，格式为ID号，数据项；

int n = \*nNum;

RecType\* R=NULL;

R= CloneArr(signal,n,0);

printf("排序前: \n");

DispList(R, n,0);

InsertSort(R, n);//可以换成其他排序函数，注意heap和radius稍有不同，具体见后面的示例

DispList(R, n, 0);

WriteNDSignalToFile(ResultFileName, R, n, 1);

printf("插入排序结束\n\n");

//Total\_time = InsertSort\_Count(R,n);

//printf("插入排序后(耗时=%f秒):\n", Total\_time);

//free(R);

//R=NULL;

R = CloneArr(signal, n, 0);

BinInsertSort(R, n);

DispList(R, n, 0);

printf("折半排序结束\n\n");

//Total\_time = BinInsertSort\_Count(R, n);

//printf("折半排序后(耗时=%f秒):\n", Total\_time);

//free(R);

//R=NULL;

R = CloneArr(signal, n, 0);

ShellSort(R, n); //对R[0..n-1]按递增有序进行希尔排序

DispList(R, n, 0);

printf("希尔排序结束\n\n");

//Total\_time = ShellSort\_Count(R, n);

//printf("希尔排序后(耗时=%f秒):\n", Total\_time);

//free(R);

// R=NULL;

R = CloneArr(signal, n, 0);

BubbleSort(R, n);

DispList(R, n, 0);

printf("冒泡排序结束\n\n");

//Total\_time = BubbleSort\_Count(R, n);

//printf("冒泡排序后(耗时=%f秒):\n", Total\_time);

//free(R);

//R=NULL;

R = CloneArr(signal, n, 0);

QuickSort(R, 0, n - 1);

DispList(R, n, 0);

printf("快速排序结束\n\n");

//Total\_time = QuickSort\_Count(R, 0, n - 1);

//printf("快速排序后(耗时=%f秒):\n", Total\_time);

//free(R);

//R=NULL;

R = CloneArr(signal, n, 0);

SelectSort(R, n);

DispList(R, n, 0);

//Total\_time = SelectSort\_Count(R, n);

//printf("简单选择排序后(耗时=%f毫秒):\n", Total\_time);

printf("选择排序结束\n\n");

R = CloneArr(signal, n, 0);

MergeSort(R, n);

DispList(R, n, 0);

//Total\_time = MergeSort\_Count(R, n);

//printf("两路归并排序后(耗时=%f毫秒):\n", Total\_time);

printf("归并排序结束\n\n");

free(R);

R=NULL;

return 1;

}

**用于测试运行时间：**

**因为print函数的运行时间远大于排序本身消耗的时间，如果直接在排序算法两端加上clock（）函数不能代表真实的运行时间。所以我将各个排序函数重写：返回值为所需时间并且删除printf函数和DispList函数。用XX\_Count函数来统计每一种排序的时间。使用个数为10000的数据集**

int main()

{

clock\_t start, finish; //定义第一次调用CPU时钟单位的实际，可以理解为定义一个计数器

double Total\_time; //定义一个double类型的变量，用于存储时间单位

RecType\* signal=NULL;

int nNum[1] ;

int nFlag=2;//1表示直接读取key，2表示通过计算得到，如实验示例。

const char\* SourceFileName = "d:\\sortinput.txt";

const char\* ResultFileName = "d:\\sortoutput.txt";

signal=ReadNDSignalFromFile(SourceFileName, nNum, nFlag);

int n = \*nNum;

RecType\* R=NULL;

R= CloneArr(signal,n,0);

printf("排序前: \n");

Total\_time = InsertSort\_Count(R,n);

printf("插入排序后(耗时=%f秒):\n", Total\_time);

R = CloneArr(signal, n, 0);

Total\_time = BinInsertSort\_Count(R, n);

printf("折半排序后(耗时=%f秒):\n", Total\_time);

R = CloneArr(signal, n, 0);

Total\_time = ShellSort\_Count(R, n);

printf("希尔排序后(耗时=%f秒):\n", Total\_time);

R = CloneArr(signal, n, 0);

Total\_time = BubbleSort\_Count(R, n);

printf("冒泡排序后(耗时=%f秒):\n", Total\_time);

R = CloneArr(signal, n, 0);

Total\_time = QuickSort\_Count(R, 0, n - 1);

printf("快速排序后(耗时=%f秒):\n", Total\_time);

R = CloneArr(signal, n, 0);

Total\_time = SelectSort\_Count(R, n);

printf("简单选择排序后(耗时=%f毫秒):\n", Total\_time);

//数组的存储位于data[1]------ - data[n]

free(R);

R=NULL;

return 1;

}

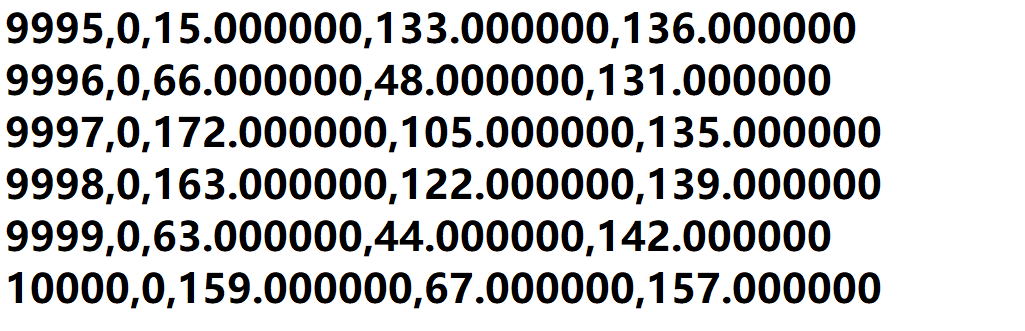
**5.3初始参数设置**

用以下函数生成了两个数据集：

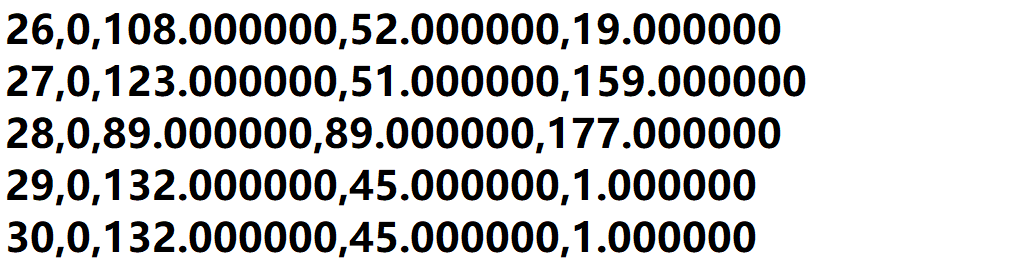
srand((float)time(NULL)); //用于生成输入数据文件

RandomSeed();

用Srand函数和rand函数生成了1-200的随机数，分为两种情况：1、计算运行时间是生成10000组数据，保存在sortinput1.txt中。因为在足够大的数据才能显示出运行时间的差别，且不包含显示函数，不会因为数据量大造成翻阅麻烦。每组数据包含关键字、ID、三个维度的数据。

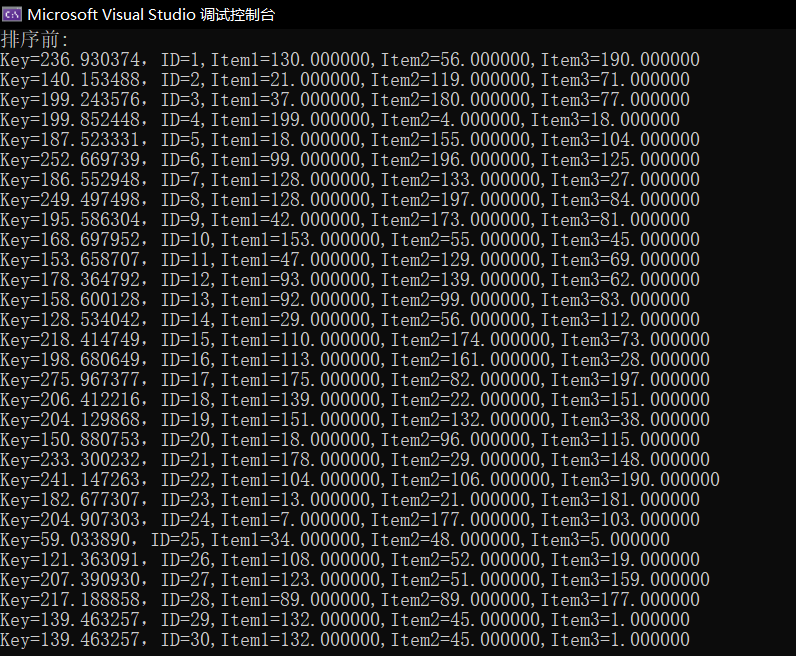


2、观察排序结果则生成30组数据，保存在sortinput.txt中。主要目的在于观察排序之后的结果的差别，数据量大容易造成翻阅的麻烦。每组数据包含关键字、ID、三个维度的数据。

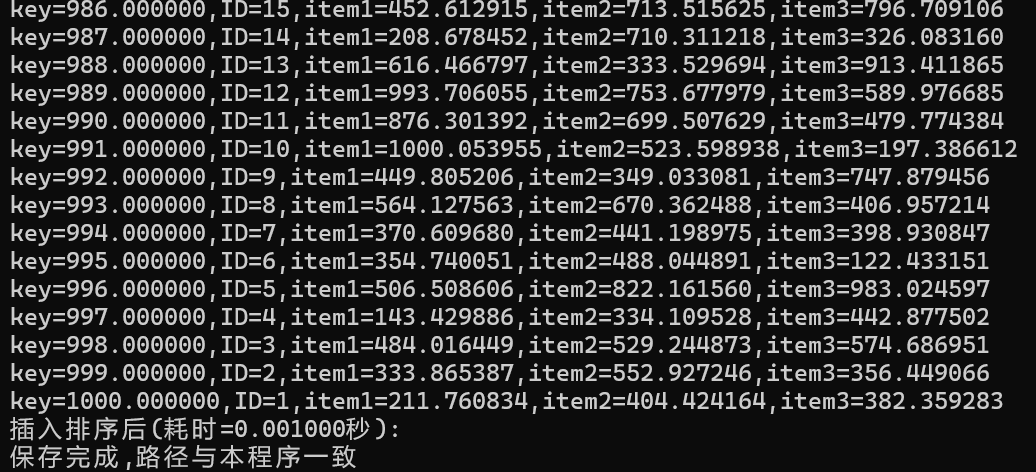


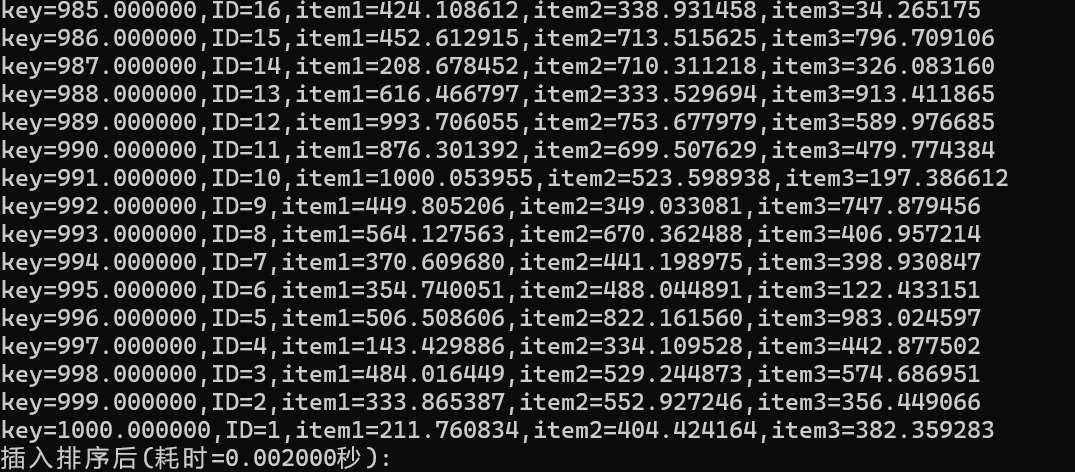
**5.4实验结果**

实验结果截图：

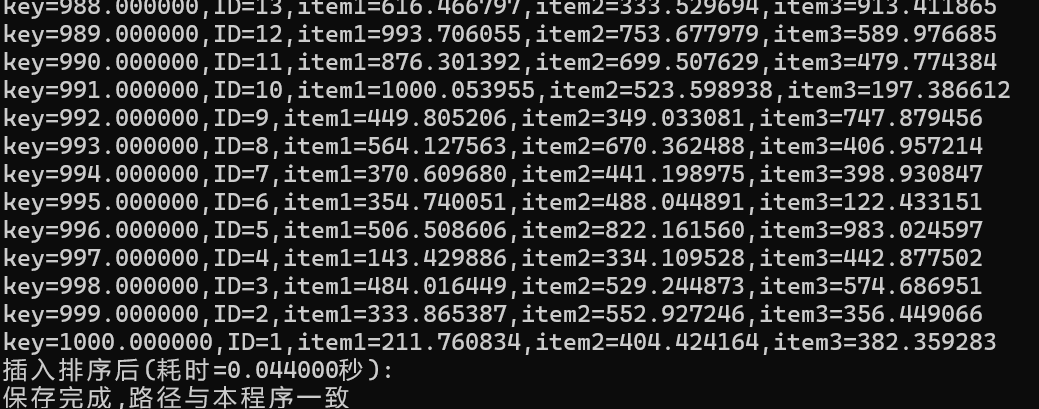


快速排序过程及结果：

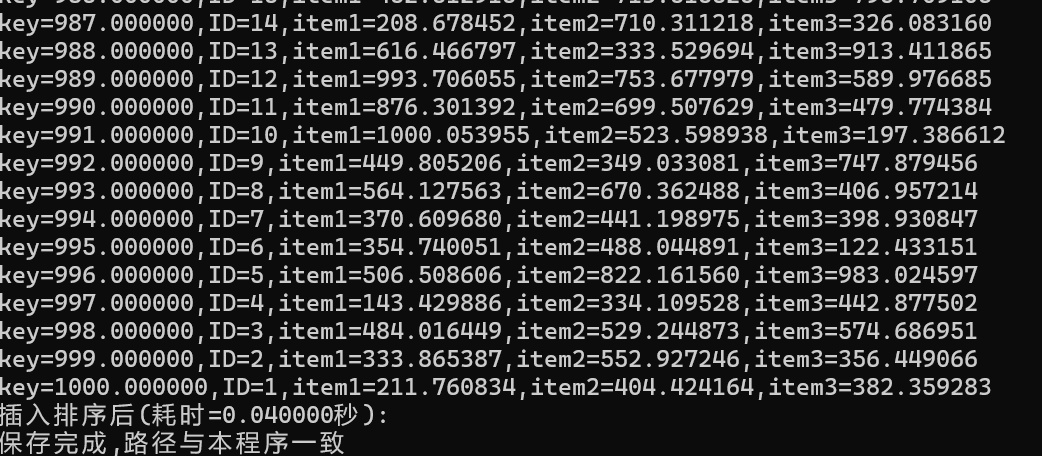


希尔排序结果：

冒泡排序结果：



简单选择排序



**理论上时间复杂度为：冒泡排序=简单选择>快速排序>希尔排序**

**实验结果定性来说是符合理论的**

**5.5实验分析**

1. **时间复杂度：**

多维数字信号的排序算法的时间复杂度取决于算法的选择、数据的维度以及数据的规模。对于常见的排序算法，如冒泡排序、插入排序和选择排序，它们的时间复杂度通常较高，特别是在处理多维数据时。相比之下，一些高效的排序算法，如归并排序、快速排序和堆排序，在处理多维数字信号时可能具有更好的时间复杂度表现。这些算法通常具有较低的时间复杂度，并且能够在较短的时间内完成排序任务。

需要注意的是，多维数字信号的排序算法可能需要考虑额外的计算开销，如维度之间的比较和排序。这些额外的计算步骤可能会增加算法的时间复杂度。

1. **空间复杂度：**

空间复杂度是指算法在执行过程中所需的额外存储空间。对于多维数字信号的排序算法，空间复杂度的分析需要考虑算法所需的临时存储空间以及数据结构的大小。一些简单的排序算法，如冒泡排序和插入排序，可能只需要常量级的额外空间来存储临时变量，因此其空间复杂度相对较低。然而，一些更复杂的排序算法可能需要使用额外的数据结构来辅助排序操作，如递归调用栈、临时数组或优先队列等。

在处理多维数字信号时，如果算法需要存储多个维度的数据或进行多次排序操作，可能会导致空间复杂度的增加。因此，在设计多维数字信号的排序算法时，需要综合考虑算法的空间效率和实际应用的需求。

1. **边界条件分析：**
2. 数据维度为0或1的情况：当数据的维度为0时，实际上没有数据可供排序。当数据的维度为1时，排序问题退化为一维数据的排序问题。在这些情况下，算法应该能够正确处理并返回正确的结果。
3. 数据规模极小或极大的情况：当数据规模极小（如只有几个数据点）时，排序算法的性能可能受到影响，特别是对于那些需要多次迭代或比较的算法。当数据规模极大时，算法的性能和内存消耗可能成为一个问题。因此，算法需要能够适应不同规模的数据输入。
4. 数据的特殊分布：在某些情况下，数据的分布可能具有特殊性，如所有维度的值都相同或大部分维度的值非常接近。这些特殊情况可能导致某些排序算法失效或性能下降。因此，算法需要能够处理各种分布特性的数据。
5. **运行结果错误原因分析：**
6. 算法实现错误：可能是算法实现过程中的逻辑错误或编码错误，导致排序结果不正确。这需要对算法代码进行仔细检查和调试，以找到并修复错误。
7. 边界条件处理不当：如前面所述，边界条件的处理对算法的正确性至关重要。如果算法在边界条件下的处理不当，可能导致排序结果错误。因此，需要对边界条件进行仔细考虑和测试。