实验报告电子版

实训四

班级：2

选课序号：04

**数据结构实训**

**—— 多维数字信号的排序**

姓 名： 江一诺

学 号： 2220220821

指导老师： 韩凤

实验日期： 2025年4月24日

**大连海事大学电子信息工程专业**

**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**

### (1) 基本数据结构

**SignalElement**结构体，包含一个double类型的distance字段

信号元素数组：SignalElement \*signals

原始距离值数组：double \*distanceValues

数据量：int size

### (2) 读取距离数据

**readDistancesFromFile(filename, values)**

输入：文件名filename，距离值数组引用values

输出：读取的数据量（失败则返回-1）

步骤：

1. 打开文件，失败则返回-1
2. 统计文件中的有效数据行数count
3. 如果count为0，关闭文件并返回-1
4. 重置文件指针到开头
5. 为values分配内存，失败则关闭文件并返回-1
6. 读取所有浮点数据到values数组
7. 关闭文件并返回数据量count

### (3) 插入排序(insertionSort)

**insertionSort(array, size)**

输入：数组array，大小size

输出：无（修改原数组）

步骤：

1. 从第二个元素开始遍历(i=1)到最后一个元素
2. 保存当前元素key = array[i]
3. 从当前位置向前查找(j=i-1)，条件是j≥0且array[j].distance > key.distance
4. 每次将较大元素向后移动：array[j+1] = array[j]
5. j递减
6. 将key插入到合适位置：array[j+1] = key

### (4) 希尔排序 (shellSort)

**shellSort(array, size)**

输入：数组array，大小size

输出：无（修改原数组）

步骤：

1. 从间隔gap=size/2开始，每次gap减半直到gap为0
2. 对每个gap，从第gap个元素开始向后遍历
3. 对每个元素，保存为temp
4. 向前以gap为间隔，比较并移动元素，直到找到合适的位置
5. 将temp插入到合适位置

### (5) 快速排序 (quickSort)

**quickSort(array, size)**

输入：数组array，大小size

输出：无（修改原数组）

步骤：调用递归函数quickSortRecursive(array, 0, size-1)

**quickSortRecursive(array, low, high)**

输入：数组array，起始索引low，结束索引high

输出：无（修改原数组）

步骤：

1. 如果low >= high则返回（递归终止条件）
2. 调用partition函数找到基准元素位置pi
3. 递归排序小于基准的部分：quickSortRecursive(array, low, pi-1)
4. 递归排序大于基准的部分：quickSortRecursive(array, pi+1, high)

**partition(array, low, high)**

输入：数组array，起始索引low，结束索引high

输出：基准元素的最终位置

步骤：

1. 选择最后一个元素作为基准pivot = array[high].distance
2. 初始化指针i = low-1
3. 遍历从low到high-1的元素
4. 如果元素小于基准，则i++并交换array[i]和array[j]
5. 交换array[i+1]和array[high]
6. 返回i+1（基准元素的最终位置）

### (6) 选择排序 (selectionSort)

**selectionSort(array, size)**

输入：数组array，大小size

输出：无（修改原数组）

步骤：

1. 对每个位置i从0到size-2
2. 找到从i到size-1中最小元素的索引min\_idx
3. 如果min\_idx不等于i，交换array[i]和array[min\_idx]

### (7) 性能测试函数 (testSortingPerformance)

**testSortingPerformance()**

输入：无

输出：无（写入到文件）

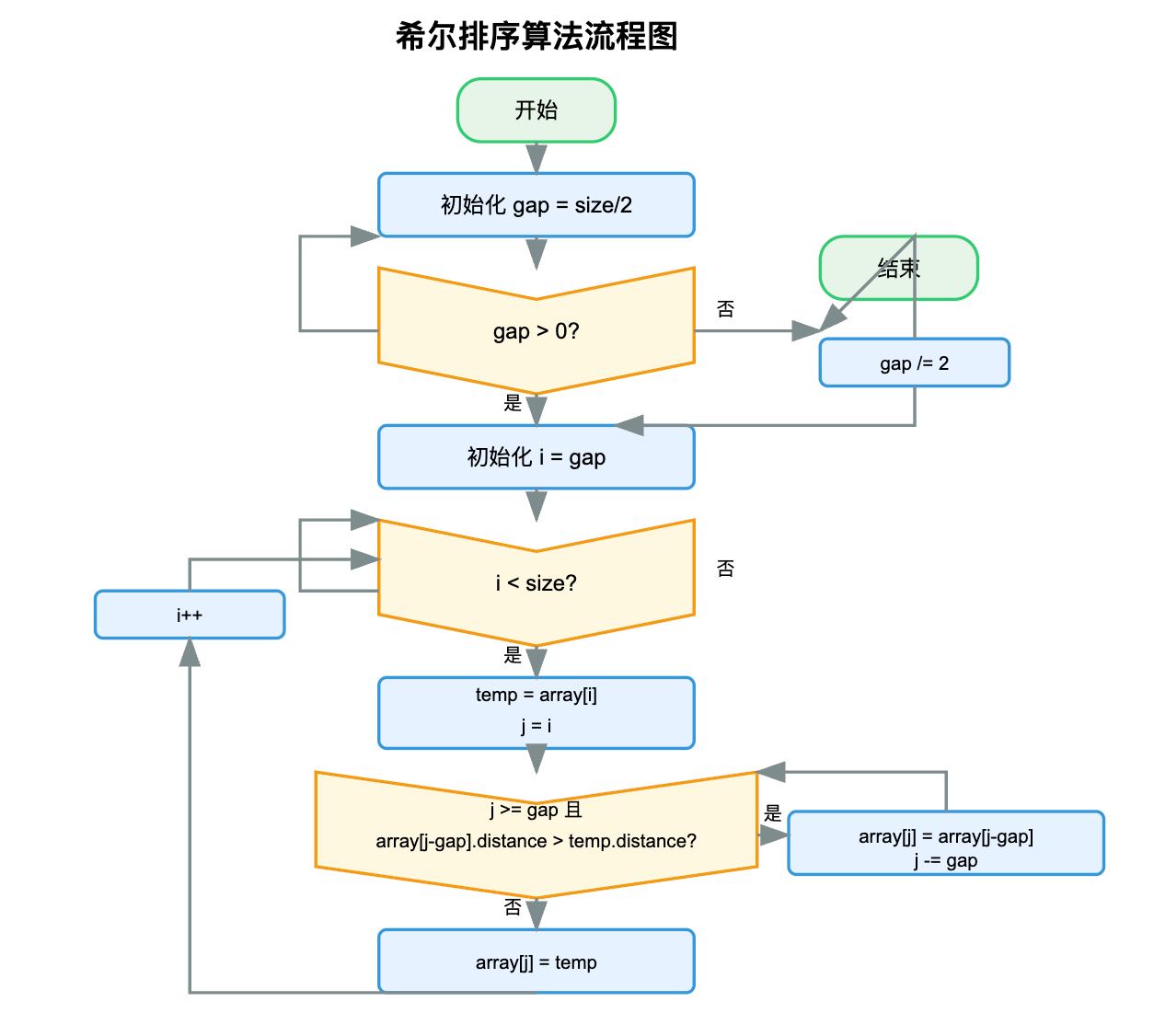
步骤：

1. 创建输出文件"运行时间对比.txt"
2. 从"计算得出的距离.txt"读取距离数据
3. 创建原始数组和三个副本用于不同的排序算法
4. 对每种排序算法（希尔、快速、选择）：
   1. 记录开始时间
   2. 执行排序
5. 记录结束时间并计算用时
6. 将结果写入输出文件
7. 释放所有分配的内存并关闭文件

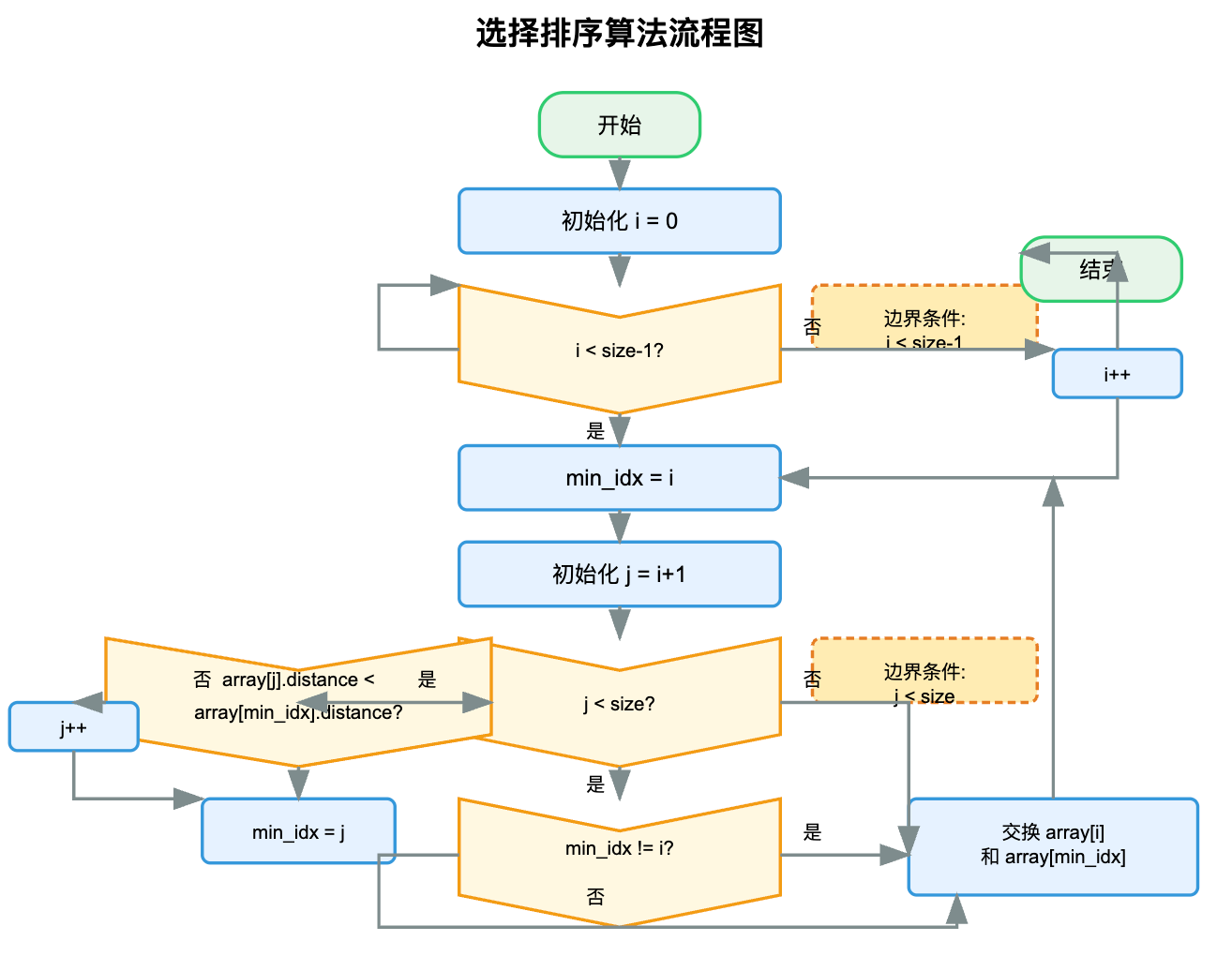
### (8) 主函数 (main)

步骤：

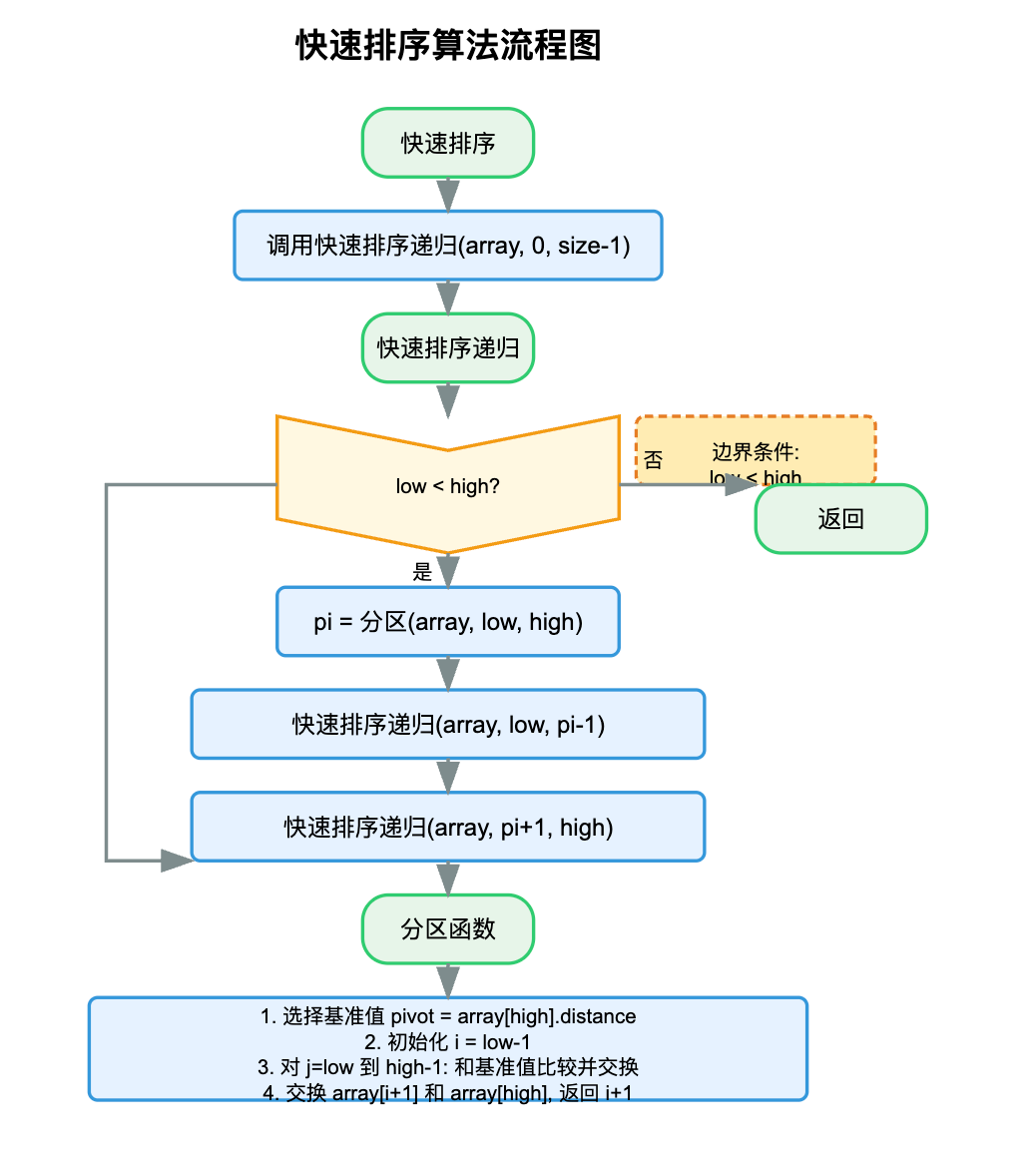
1. 初始化随机数种子
2. 从文件"计算得出的距离.txt"读取距离数据
3. 创建信号元素数组并初始化
4. 创建三个数组副本用于不同的排序算法
5. 显示排序前的数据（前10个元素）
6. 分别执行希尔排序、快速排序和选择排序，记录并显示执行时间
7. 每次排序后显示排序结果（前10个元素）
8. 释放所有分配的内存
9. 执行性能测试并保存结果到文件
10. 返回0表示程序执行成功
11. 希尔排序法流程图



1. 选择排序法流程图



1. 快速排序法流程图



1. **算法实现**

**各元素到参考点的距离.c**

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

*// 计算欧几里得距离*

*//两个点在多维空间中各维度差值的平方和的平方根*

double euclideanDistance(double \*point1, double \*point2, int dimension) {

    double sum = 0.0;

    for (int i = 0; i < dimension; i++) {

        double diff = point1[i] - point2[i];

        sum += diff \* diff;

    }

    return sqrt(sum);

}

*// 计算街区距离（曼哈顿距离）*

*//两个点在多维空间中各维度差值的绝对值之和。*

double manhattanDistance(double \*point1, double \*point2, int dimension) {

    double sum = 0.0;

    for (int i = 0; i < dimension; i++) {

        sum += fabs(point1[i] - point2[i]);

    }

    return sum;

}

*// 计算余弦距离*

*//两个向量之间的余弦相似度，并将其转换为距离1 - cos(θ)*

double cosineDistance(double \*point1, double \*point2, int dimension) {

    double dotProduct = 0.0;

    double norm1 = 0.0;

    double norm2 = 0.0;

    for (int i = 0; i < dimension; i++) {

        dotProduct += point1[i] \* point2[i];

        norm1 += point1[i] \* point1[i];

        norm2 += point2[i] \* point2[i];

    }

*// 边界条件：避免除以零*

    if (norm1 == 0.0 || norm2 == 0.0) {

        return 1.0; *// 最大距离*

    }

*// 余弦相似度转换为余弦距离：1 - cos(θ)*

    return 1.0 - (dotProduct / (sqrt(norm1) \* sqrt(norm2)));

}

*// 计算多维信号中各元素到参考点的距离*

*//distancesType: 0-欧几里得距离，1-曼哈顿距离，2-余弦距离*

void calculateDistances(double \*\*signals, int signalCount, int dimension,

                        double \*referencePoint, double \*results, int distanceType) {

    for (int i = 0; i < signalCount; i++) {

        switch (distanceType) {

            case 0: *// 欧几里得距离*

                results[i] = euclideanDistance(signals[i], referencePoint, dimension);

                break;

            case 1: *// 街区距离*

                results[i] = manhattanDistance(signals[i], referencePoint, dimension);

                break;

            case 2: *// 余弦距离*

                results[i] = cosineDistance(signals[i], referencePoint, dimension);

                break;

            default:

                printf("不支持的距离类型\n");

                return;

        }

    }

}

*// 打印距离结果*

void printDistances(double \*results, int count) {

    printf("各元素到参考点的距离：\n");

    for (int i = 0; i < count; i++) {

        printf("元素%d: %.4f\n", i + 1, results[i]);

    }

}

*// 保存距离结果到文件*

void saveDistancesToFile(double \*results, int count, const char \*filename) {

    FILE \*file = fopen(filename, "w");

    if (file == NULL) {

        printf("无法创建文件: %s\n", filename);

        return;

    }

    for (int i = 0; i < count; i++) {

        fprintf(file, "%.6f\n", results[i]);

    }

    fclose(file);

    printf("距离数据已保存到文件: %s\n", filename);

}

*// 逐层释放分配的内存*

void freeMemory(double \*\*signals, int count) {

    for (int i = 0; i < count; i++) {

        free(signals[i]);

    }

    free(signals);

}

*// 从文件读取信号数据*

int readSignalData(const char \*filename, double \*\*\*signals, int \*signalCount, int \*dimension, double \*\*referencePoint) {

    FILE \*file = fopen(filename, "r");

*//边界条件：文件不存在*

    if (file == NULL) {

        printf("无法打开文件: %s\n", filename);

        return 0;

    }

*// 读取维度和信号数量*

*//边界条件：维度和信号数量不是两个整数*

    if (fscanf(file, "%d %d", dimension, signalCount) != 2) {

        printf("读取维度和信号数量失败\n");

        fclose(file);

        return 0;

    }

*// 分配参考点内存*

    \*referencePoint = (double \*)malloc(\*dimension \* sizeof(double));

*// 边界条件：参考点不存在*

    if (\*referencePoint == NULL) {

        printf("内存分配失败\n");

        fclose(file);

        return 0;

    }

*// 读取参考点数据*

    for (int i = 0; i < \*dimension; i++) {

*// 边界条件：参考点数据不是一个浮点数*

        if (fscanf(file, "%lf", &(\*referencePoint)[i]) != 1) {

            printf("读取参考点数据失败\n");

            free(\*referencePoint);

            fclose(file);

            return 0;

        }

    }

*// 分配信号数据内存*

    \*signals = (double \*\*)malloc(\*signalCount \* sizeof(double \*));

*// 边界条件：信号数据不存在*

    if (\*signals == NULL) {

        printf("内存分配失败\n");

        free(\*referencePoint);

        fclose(file);

        return 0;

    }

    for (int i = 0; i < \*signalCount; i++) {

        (\*signals)[i] = (double \*)malloc(\*dimension \* sizeof(double));

*// 边界条件：信号数据不存在*

        if ((\*signals)[i] == NULL) {

            printf("内存分配失败\n");

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

                free((\*signals)[j]);

            }

            free(\*signals);

            free(\*referencePoint);

            fclose(file);

            return 0;

        }

*// 读取每个信号的数据*

        for (int j = 0; j < \*dimension; j++) {

*// 边界条件：信号数据不是一个浮点数*

            if (fscanf(file, "%lf", &(\*signals)[i][j]) != 1) {

                printf("读取信号数据失败\n");

                for (int k = 0; k <= i; k++) {

                    free((\*signals)[k]);

                }

                free(\*signals);

                free(\*referencePoint);

                fclose(file);

                return 0;

            }

        }

    }

    fclose(file);

    return 1;

}

排序.c

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

#include <time.h>

*// 定义信号元素结构体*

typedef struct {

    double distance;  *// 计算的距离*

} SignalElement;

*// 从文件读取距离数据*

int readDistancesFromFile(const char\* filename, double\*\* values) {

    FILE\* file = fopen(filename, "r");

    if (file == NULL) {

        printf("无法打开文件 %s\n", filename);

        return -1;

    }

*// 先统计文件中的数据行数*

    int count = 0;

    char line[5000];

    double temp;

    while (fgets(line, sizeof(line), file) != NULL) {

        if (sscanf(line, "%lf", &temp) == 1) {

            count++;

        }

    }

    if (count <= 0) {

        printf("文件中没有有效数据\n");

        fclose(file);

        return -1;

    }

*// 重新定位到文件开头*

    rewind(file);

*// 分配内存*

    \*values = (double\*)malloc(count \* sizeof(double));

    if (\*values == NULL) {

        printf("内存分配失败\n");

        fclose(file);

        return -1;

    }

*// 读取数据*

    int i = 0;

    while (i < count && fgets(line, sizeof(line), file) != NULL) {

        if (sscanf(line, "%lf", &(\*values)[i]) == 1) {

            i++;

        }

    }

    fclose(file);

    return count;

}

*// 1. 插入排序 (Insertion Sort)*

void insertionSort(SignalElement\* array, int size) {

    int i, j;

    SignalElement key;

    for (i = 1; i < size; i++) {

        key = array[i];

        j = i - 1;

*//边界条件，如果当前元素比前一个元素小，则直接插入，否则把当前元素向后移动一位*

        while (j >= 0 && array[j].distance > key.distance) {

            array[j + 1] = array[j];

            j = j - 1;

        }

        array[j + 1] = key;

    }

}

*// 2. 希尔排序 (Shell Sort) - 插入排序的改进版*

void shellSort(SignalElement\* array, int size) {

*// 使用希尔增量序列*

    for (int gap = size / 2; gap > 0; gap /= 2) {

*// 对每个增量执行插入排序*

        for (int i = gap; i < size; i++) {

            SignalElement temp = array[i];

            int j;

            for (j = i; j >= gap && array[j - gap].distance > temp.distance; j -= gap) {

                array[j] = array[j - gap];

            }

            array[j] = temp;

        }

    }

}

*// 3. 快速排序 (Quick Sort) - 交换排序的一种*

void swap(SignalElement\* a, SignalElement\* b) {

    SignalElement temp = \*a;

    \*a = \*b;

    \*b = temp;

}

int partition(SignalElement\* array, int low, int high) {

*// 选择最后一个元素作为基准*

    double pivot = array[high].distance;

    int i = low - 1;

    for (int j = low; j <= high - 1; j++) {

        if (array[j].distance < pivot) {

            i++;

            swap(&array[i], &array[j]);

        }

    }

    swap(&array[i + 1], &array[high]);

    return i + 1;

}

void quickSortRecursive(SignalElement\* array, int low, int high) {

*// 递归终止条件：起始索引<结束索引*

    if (low < high) {

        int pi = partition(array, low, high);

        quickSortRecursive(array, low, pi - 1);

        quickSortRecursive(array, pi + 1, high);

    }

}

void quickSort(SignalElement\* array, int size) {

    quickSortRecursive(array, 0, size - 1);

}

*// 4. 选择排序 (Selection Sort)*

void selectionSort(SignalElement\* array, int size) {

    int i, j, min\_idx;

    for (i = 0; i < size - 1; i++) {

        min\_idx = i;

        for (j = i + 1; j < size; j++) {

            if (array[j].distance < array[min\_idx].distance) {

                min\_idx = j;

            }

        }

*// 交换找到的最小元素与第一个元素*

        if (min\_idx != i) {

            SignalElement temp = array[i];

            array[i] = array[min\_idx];

            array[min\_idx] = temp;

        }

    }

}

*// 打印信号元素数组*

void printArray(SignalElement\* array, int size) {

    for (int i = 0; i < size; i++) {

        printf("距离: %.6f\n", array[i].distance);

    }

    printf("\n");

}

*// 复制数组*

void copyArray(SignalElement\* dest, SignalElement\* src, int size) {

    for (int i = 0; i < size; i++) {

        dest[i] = src[i];

    }

}

*// 测试排序算法性能并输出到文件*

void testSortingPerformance() {

    FILE\* outputFile = fopen("运行时间对比.txt", "w");

    if (outputFile == NULL) {

        printf("无法创建输出文件\n");

        return;

    }

    fprintf(outputFile, "排序算法性能测试报告\n");

    fprintf(outputFile, "==========================\n\n");

*// 完整文件路径*

    const char\* filepath = "./计算得出的距离.txt";

*// 读取实际数据*

    double\* distanceValues;

    int size = readDistancesFromFile(filepath, &distanceValues);

    if (size <= 0) {

        printf("性能测试读取数据失败或数据为空，尝试使用绝对路径\n");

        filepath = "/Users/User/Desktop/Data structure experiment/PDS01-2042220220821江一诺/实验四/计算得出的距离.txt";

        size = readDistancesFromFile(filepath, &distanceValues);

        if (size <= 0) {

            printf("性能测试使用绝对路径仍然失败\n");

            fprintf(outputFile, "无法读取数据文件\n");

            fclose(outputFile);

            return;

        }

    }

    printf("\n成功从文件读取了%d个数据用于性能测试\n数据来源: %s\n\n", size, filepath);

*// 创建信号元素数组*

    SignalElement\* originalArray = (SignalElement\*)malloc(size \* sizeof(SignalElement));

    if (originalArray == NULL) {

        printf("内存分配失败\n");

        free(distanceValues);

        fclose(outputFile);

        return;

    }

*// 初始化信号元素数组*

    for (int i = 0; i < size; i++) {

        originalArray[i].distance = distanceValues[i];  *// 距离值作为距离*

    }

    fprintf(outputFile, "数据量: %d 个元素\n\n", size);

    fprintf(outputFile, "| 算法名称 | 运行时间 (ms) |\n");

    fprintf(outputFile, "|----------|---------------|\n");

*// 为每种排序算法创建副本*

    SignalElement\* shellArray = (SignalElement\*)malloc(size \* sizeof(SignalElement));

    SignalElement\* quickArray = (SignalElement\*)malloc(size \* sizeof(SignalElement));

    SignalElement\* selectArray = (SignalElement\*)malloc(size \* sizeof(SignalElement));

    if (shellArray == NULL || quickArray == NULL || selectArray == NULL) {

        printf("内存分配失败\n");

        free(originalArray);

        if (shellArray) free(shellArray);

        if (quickArray) free(quickArray);

        if (selectArray) free(selectArray);

        free(distanceValues);

        fclose(outputFile);

        return;

    }

    copyArray(shellArray, originalArray, size);

    copyArray(quickArray, originalArray, size);

    copyArray(selectArray, originalArray, size);

*// 测试希尔排序*

    clock\_t start = clock();

    shellSort(shellArray, size);

    clock\_t end = clock();

    double shellTime = ((double)(end - start)) / CLOCKS\_PER\_SEC \* 1000; *// 毫秒*

*// 测试快速排序*

    start = clock();

    quickSort(quickArray, size);

    end = clock();

    double quickTime = ((double)(end - start)) / CLOCKS\_PER\_SEC \* 1000; *// 毫秒*

*// 测试选择排序*

    start = clock();

    selectionSort(selectArray, size);

    end = clock();

    double selectTime = ((double)(end - start)) / CLOCKS\_PER\_SEC \* 1000; *// 毫秒*

*// 输出结果*

    fprintf(outputFile, "| 希尔排序 | %13.2f |\n", shellTime);

    fprintf(outputFile, "| 快速排序 | %13.2f |\n", quickTime);

    fprintf(outputFile, "| 选择排序 | %13.2f |\n", selectTime);

*// 释放内存*

    free(originalArray);

    free(shellArray);

    free(quickArray);

    free(selectArray);

    free(distanceValues);

    fprintf(outputFile, "\n注：时间单位为毫秒 (ms)。测试环境：MacOS。\n");

    fprintf(outputFile, "数据来源：%s（%d个数据点）\n", filepath, size);

    fclose(outputFile);

    printf("性能测试完成，结果已保存到运行时间对比.txt文件中\n");

}

**5、实验结果与分析**

**5.1测试用例**

**随机生成5000条数据：**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

int main() {

    int dimension = 3;  *// 维度为3*

    int signalCount = 5000;  *// 生成5000条数据*

*// 打开文件*

    FILE \*file = fopen("data\_距离.txt", "w");

    if (file == NULL) {

        printf("无法创建文件\n");

        return 1;

    }

*// 初始化随机数生成器*

    srand(time(NULL));

*// 写入维度和信号数量*

    fprintf(file, "%d %d\n", dimension, signalCount);

*// 生成并写入参考点数据（随机生成）*

    for (int i = 0; i < dimension; i++) {

        double value = (double)(rand() % 100) / 10.0;  *// 生成0到10之间的随机浮点数*

        fprintf(file, "%.1f ", value);

    }

    fprintf(file, "\n");

*// 生成并写入信号数据*

    for (int i = 0; i < signalCount; i++) {

        for (int j = 0; j < dimension; j++) {

            double value = (double)(rand() % 100) / 10.0;  *// 生成0到10之间的随机浮点数*

            fprintf(file, "%.1f ", value);

        }

        fprintf(file, "\n");

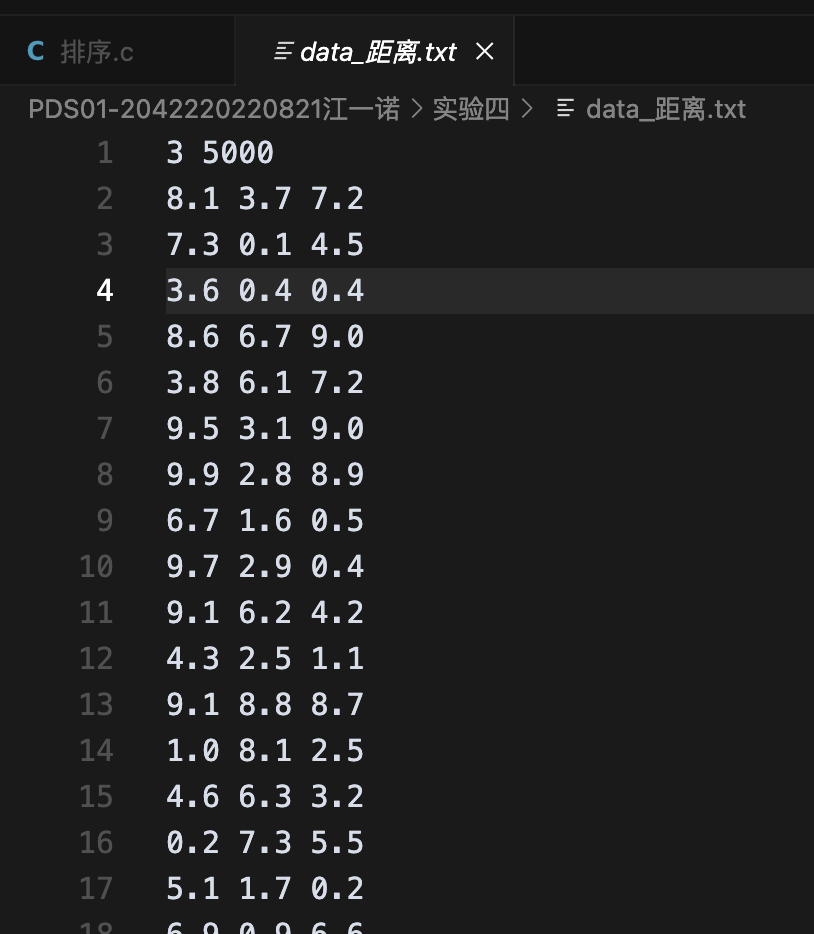
    }

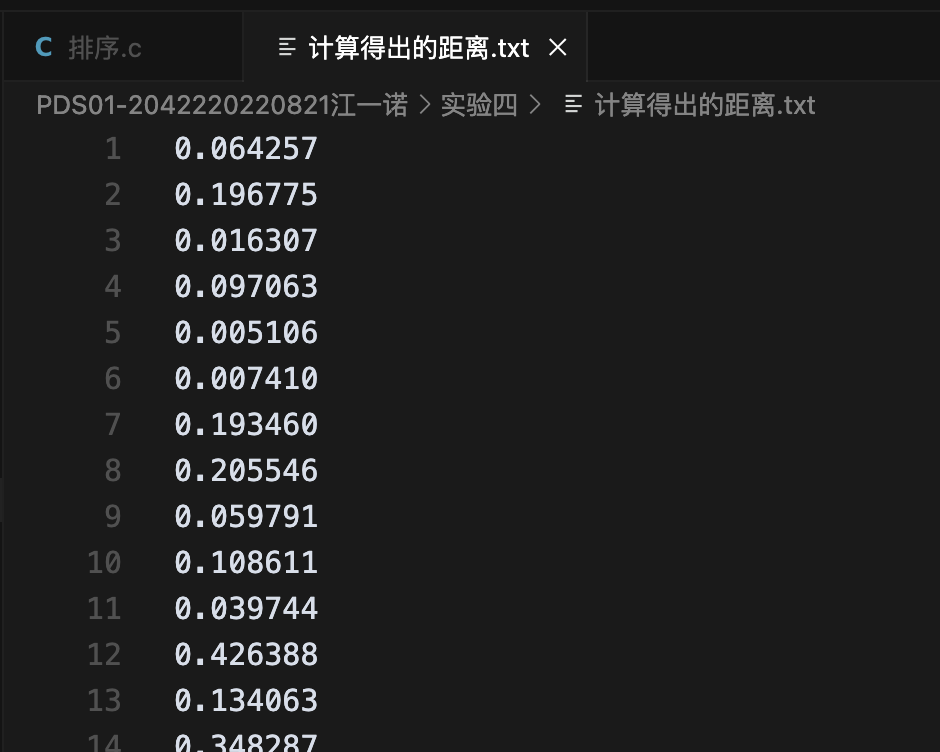
    fclose(file);

    printf("已成功生成5000条随机数据并保存到data\_距离.txt文件\n");

    return 0;

}





**5.2测试程序**

**各元素到参考点的距离.c**

*// 主函数*

int main() {

    double \*\*signals = NULL;

    double \*referencePoint = NULL;

    int dimension = 0;

    int signalCount = 0;

*// 从文件读取数据*

    if (!readSignalData("data\_距离.txt", &signals, &signalCount, &dimension, &referencePoint)) {

        return 1;

    }

    printf("\n数据读取成功\n");

*// 存储结果的数组*

    double \*results = (double \*)malloc(signalCount \* sizeof(double));

    if (results == NULL) {

        printf("内存分配失败\n");

        free(referencePoint);

        freeMemory(signals, signalCount);

        return 1;

    }

*// 显示读取的数据*

    printf("维度: %d\n", dimension);

    printf("信号数量: %d\n", signalCount);

    printf("参考点: ");

    for (int i = 0; i < dimension; i++) {

        printf("%.1f ", referencePoint[i]);

    }

    printf("\n\n信号数据:\n");

    for (int i = 0; i < signalCount; i++) {

        printf("信号 %d: ", i+1);

        for (int j = 0; j < dimension; j++) {

            printf("%.1f ", signals[i][j]);

        }

        printf("\n");

    }

    printf("\n");

    int distanceType;

    printf("请选择距离类型（0：欧几里得距离，1：街区距离，2：余弦距离）：");

    scanf("%d", &distanceType);

*// 计算距离*

    calculateDistances(signals, signalCount, dimension, referencePoint, results, distanceType);

*// 打印结果*

    printDistances(results, signalCount);

*// 保存距离结果到文件*

    saveDistancesToFile(results, signalCount, "计算得出的距离.txt");

*// 释放内存*

    freeMemory(signals, signalCount);

    free(referencePoint);

    free(results);

    return 0;

}

**排序.c**

int main() {

*// 设置随机数种子*

    srand((unsigned int)time(NULL));

*// 完整文件路径*

    const char\* filepath = "./计算得出的距离.txt";

*// 读取距离数据*

    double\* distanceValues;

    int size = readDistancesFromFile(filepath, &distanceValues);

    if (size <= 0) {

        printf("读取数据失败或数据为空，请检查文件路径: %s\n", filepath);

*// 尝试绝对路径*

        filepath = "/Users/User/Desktop/Data structure experiment/PDS01-2042220220821江一诺/实验四/计算得出的距离.txt";

        printf("尝试使用绝对路径: %s\n", filepath);

        size = readDistancesFromFile(filepath, &distanceValues);

        if (size <= 0) {

            printf("使用绝对路径仍然失败，请确认文件存在并有读取权限\n");

            return -1;

        }

    }

    printf("\n成功读取了%d个数据\n\n", size);

*// 创建信号元素数组*

    SignalElement\* signals = (SignalElement\*)malloc(size \* sizeof(SignalElement));

    if (signals == NULL) {

        printf("内存分配失败\n");

        free(distanceValues);

        return -1;

    }

*// 初始化信号元素数组*

    for (int i = 0; i < size; i++) {

        signals[i].distance = distanceValues[i];  *// 距离值作为距离*

*// 打印少量示例，避免输出过多*

        if (i < 10 || i > size - 10) {

            printf("信号[%d]: 距离=%.6f\n", i, signals[i].distance);

            if (i == 10) printf("... (省略中间数据) ...\n");

        }

    }

*// 复制数组以便使用不同排序算法*

    SignalElement\* signalsShell = (SignalElement\*)malloc(size \* sizeof(SignalElement));

    SignalElement\* signalsQuick = (SignalElement\*)malloc(size \* sizeof(SignalElement));

    SignalElement\* signalsSelect = (SignalElement\*)malloc(size \* sizeof(SignalElement));

    if (signalsShell == NULL || signalsQuick == NULL || signalsSelect == NULL) {

        printf("内存分配失败\n");

        free(signals);

        free(distanceValues);

        return -1;

    }

*// 复制数组*

    for (int i = 0; i < size; i++) {

        signalsShell[i] = signals[i];

        signalsQuick[i] = signals[i];

        signalsSelect[i] = signals[i];

    }

    printf("\n排序前 (仅显示前10个元素):\n");

    for (int i = 0; i < 10 && i < size; i++) {

        printf("距离: %.6f\n", signalsShell[i].distance);

    }

    printf("... (总共 %d 个元素) ...\n\n", size);

*// 使用希尔排序*

    printf("开始希尔排序...\n");

    clock\_t start = clock();

    shellSort(signalsShell, size);

    clock\_t end = clock();

    double shellTime = ((double)(end - start)) / CLOCKS\_PER\_SEC \* 1000; *// 毫秒*

    printf("希尔排序完成，用时 %.2f 毫秒\n", shellTime);

    printf("希尔排序后 (仅显示前10个元素):\n");

    for (int i = 0; i < 10 && i < size; i++) {

        printf("距离: %.6f\n", signalsShell[i].distance);

    }

    printf("... (总共 %d 个元素) ...\n\n", size);

*// 使用快速排序*

    printf("开始快速排序...\n");

    start = clock();

    quickSort(signalsQuick, size);

    end = clock();

    double quickTime = ((double)(end - start)) / CLOCKS\_PER\_SEC \* 1000; *// 毫秒*

    printf("快速排序完成，用时 %.2f 毫秒\n", quickTime);

    printf("快速排序后 (仅显示前10个元素):\n");

    for (int i = 0; i < 10 && i < size; i++) {

        printf("距离: %.6f\n", signalsQuick[i].distance);

    }

    printf("... (总共 %d 个元素) ...\n\n", size);

*// 使用选择排序*

    printf("开始选择排序...\n");

    start = clock();

    selectionSort(signalsSelect, size);

    end = clock();

    double selectTime = ((double)(end - start)) / CLOCKS\_PER\_SEC \* 1000; *// 毫秒*

    printf("选择排序完成，用时 %.2f 毫秒\n", selectTime);

    printf("选择排序后 (仅显示前10个元素):\n");

    for (int i = 0; i < 10 && i < size; i++) {

        printf("距离: %.6f\n", signalsSelect[i].distance);

    }

    printf("... (总共 %d 个元素) ...\n\n", size);

*// 释放内存*

    free(signals);

    free(signalsShell);

    free(signalsQuick);

    free(signalsSelect);

    free(distanceValues);

*// 执行性能测试并输出到文件*

    printf("\n开始排序算法性能测试...\n");

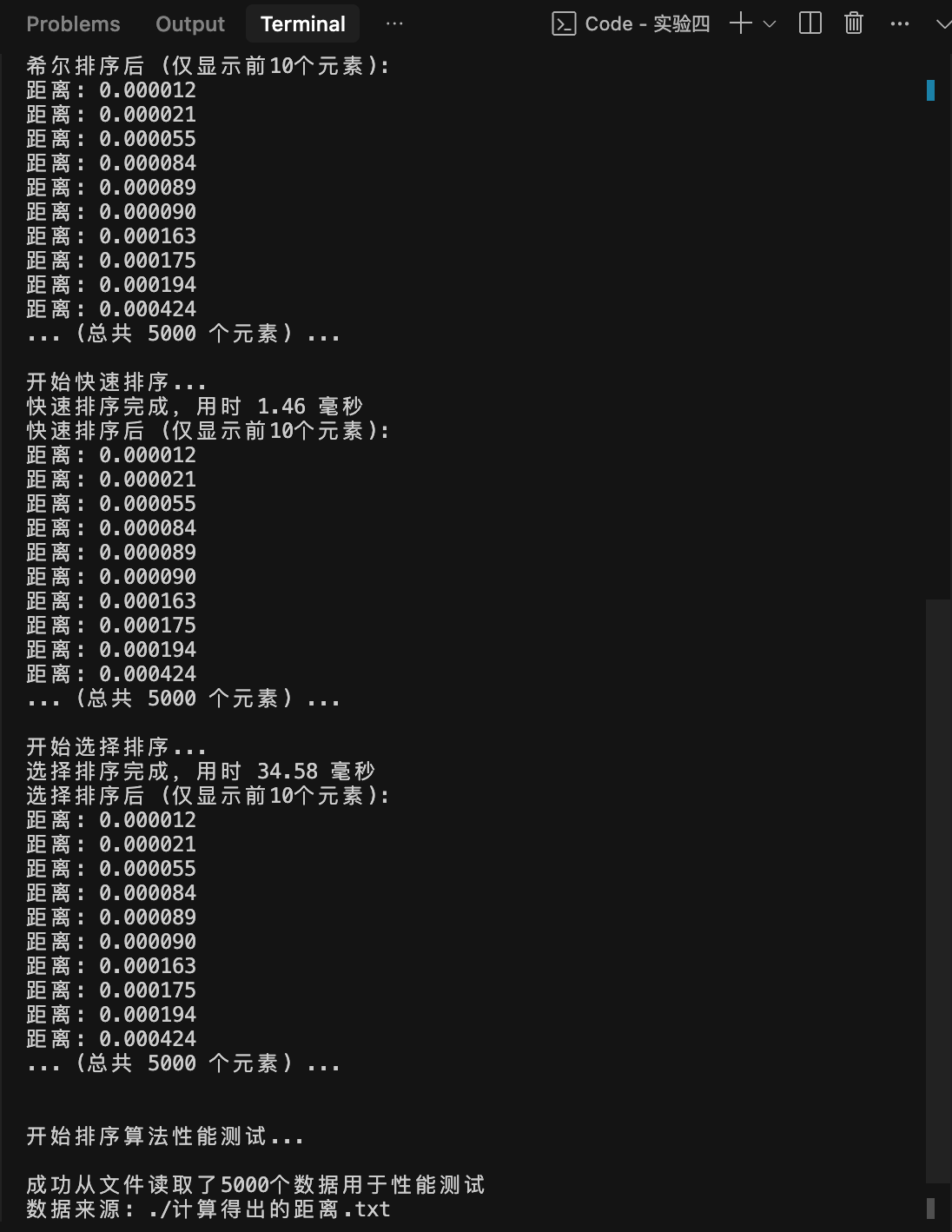
    testSortingPerformance();

    return 0;

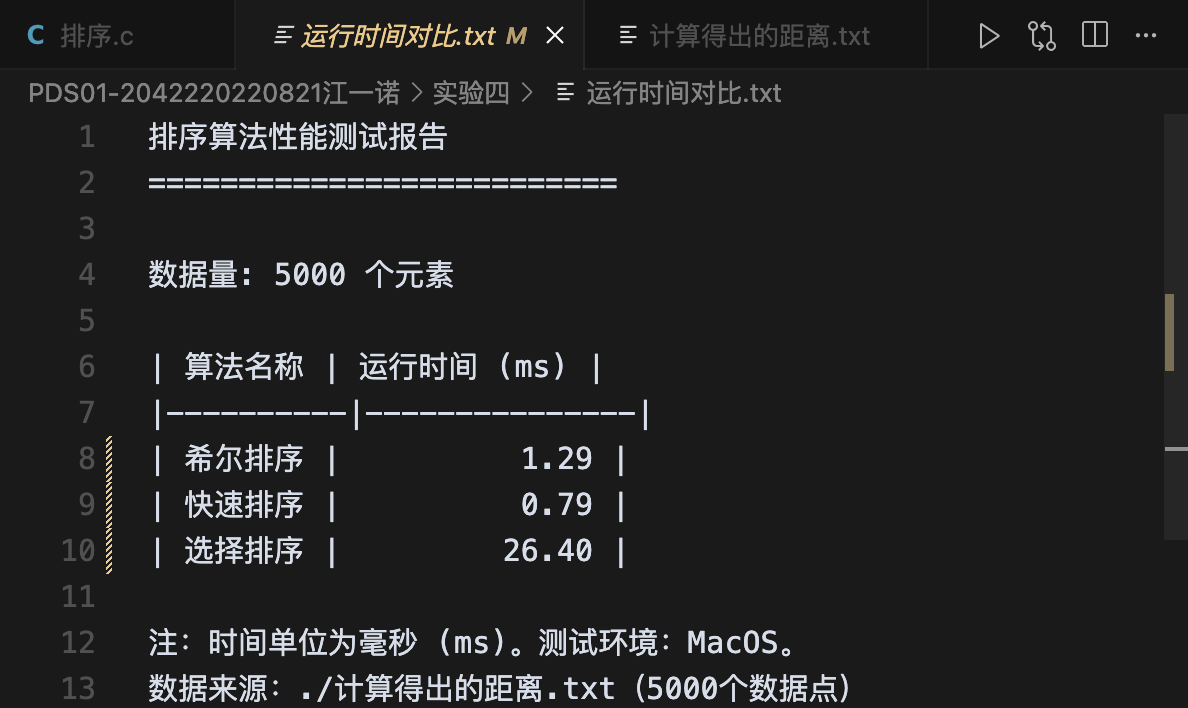
}

**5.3实验结果**

实验结果截图：



**性能测试——运行时间对比：**



**理论分析——时间复杂度：**



**理论上时间复杂度为：选择排序>快速排序>希尔排序**

***实验结果符合理论分析***

**5.4实验分析**

## 1. 插入排序(希尔排序)

希尔排序是插入排序的一种改进版本，也称为”缩小增量排序”。

### 优点

1. **克服了直接插入排序对基本有序的序列效率较高，对乱序序列效率较低的缺点**
2. **通过设置不同的增量序列，性能表现可以进一步优化**
3. **在小型数据集上表现良好**
4. **预排序后，最后一次的直接插入排序可以在几乎有序的情况下完成，大大提高了效率**
5. **比较和移动的操作都不是在相邻的元素之间进行的，因此希尔排序打破了严重时间复杂度的下限**
6. **空间复杂度为O(1)，是原地排序算法**

### 缺点

1. **增量序列的选择对排序性能影响较大，不同增量序列的选择可能导致不同的时间复杂度**
2. **对于大型数据集，其性能不如快速排序等高效算法**
3. **不稳定：因为相同关键字的记录可能会被分在不同的组中，导致相对位置发生变化**
4. **理论分析难度大，最优复杂度尚未被确定**

### 适用范围

1. **中小规模数据的排序（几百到几千个元素）**
2. **数据基本有序或接近有序的情况**
3. **内存空间有限、对稳定性要求不高的场景**
4. **作为高级排序算法（如快速排序）的补充，用于处理小规模子数组**

## 2. 交换排序(快速排序)

快速排序是一种分治的排序算法，基于比较和交换操作。

### 优点

1. **平均时间复杂度为O(nlogn)，是目前被认为最快的排序算法之一**
2. **内部排序算法中最好的选择**
3. **数据移动较少，一般只需要交换操作即可**
4. **可以进行原地排序，空间复杂度为O(logn)（主要用于递归调用的栈空间）**
5. **分治法的思想使其能够有效地利用缓存局部性**
6. **对于大数据集表现优异**

### 缺点

1. **最坏情况下的时间复杂度为O(n²)，当输入数据已经有序时性能会大幅下降**
2. **对于小数组，递归的开销可能会大于排序本身的开销**
3. **不稳定排序：相同键值的元素在排序后的相对位置可能发生变化**
4. **对于递归实现，可能会导致栈溢出（特别是对于非常大的数据集）**
5. **对于基准元素（枢轴）的选择敏感，选择不当会影响性能**

### 适用范围

1. **大规模数据排序，特别是随机分布的数据**
2. **内存限制不是特别严格的场景**
3. **对排序稳定性不敏感的应用**
4. **作为通用排序算法的首选，在各种编程语言的标准库中广泛采用**
5. **需要高性能排序的实时系统**

## 3. 选择排序

选择排序是一种简单直观的排序算法，基于选择操作。

### 优点

1. **实现简单，易于理解**
2. **数据移动操作少，每次交换都会将一个元素放到其最终位置**
3. **对于小规模数据，其简单性可能导致比其他复杂算法更快的实际运行时间**
4. **空间复杂度为O(1)，是原地排序算法**
5. **对输入数据的初始排列不敏感，无论原始数据如何分布，时间复杂度始终是O(n²)**

### 缺点

1. **时间复杂度固定为O(n²)，无论输入数据的分布如何**
2. **在大规模数据集上性能很差**
3. **不稳定排序：相同键值的元素在排序后的相对位置可能发生变化**
4. **没有利用数据的部分有序性**
5. **即使数组已经排序，仍然需要执行完整的排序过程**

### 适用范围

1. **小规模数据的排序（几十到几百个元素）**
2. **对排序稳定性不敏感的应用**
3. **内存空间严格受限的嵌入式系统**
4. **作为教学演示排序算法基本原理的实例**
5. **算法实现的简单性比效率更重要的场景**