**操作系统实验**

**实验报告**

题目：[实验六 磁盘调度算法]

**专业：[软件工程]**

**年级：[2017级]**

**姓名：[陈思翔]**

**学号：[1725121003]**

目录

[一、需求分析 4](#_Toc27927000)

[1、程序设计的任务和目的 4](#_Toc27927001)

[2、输入的形式和输入值的范围 4](#_Toc27927002)

[3、输出的形式 4](#_Toc27927003)

[4、程序所能达到的功能 4](#_Toc27927004)

[5、测试数据，包括正确的输入及其输出结果和含有错误的输入及其输出结果 4](#_Toc27927005)

[正确用例 4](#_Toc27927006)

[错误用例（使用始终使用同一磁道） 6](#_Toc27927007)

[二、概要设计 6](#_Toc27927008)

[1、抽象数据类型的定义 6](#_Toc27927009)

[2、主程序的流程 7](#_Toc27927010)

[3、各程序模块之间的层次(调用)关系 7](#_Toc27927011)

[三、详细设计 8](#_Toc27927012)

[1、先来先服务FCFS算法 8](#_Toc27927013)

[2、最短磁盘调度时间优先SSTF算法 9](#_Toc27927014)

[3、SCAN算法 10](#_Toc27927015)

[4、循环SCAN算法 11](#_Toc27927016)

[四、调试分析 12](#_Toc27927017)

[调试过程中遇到的问题以及解决方法，设计与实现的回顾讨论和分析 12](#_Toc27927018)

[算法的性能分析(包括基本操作和其它算法的时间复杂度和空间复杂度的分析)及其改进设想 12](#_Toc27927019)

[性能分析 12](#_Toc27927020)

[改进设想 12](#_Toc27927021)

[五、用户使用说明 12](#_Toc27927022)

[使用说明 12](#_Toc27927023)

[六、测试结果 13](#_Toc27927024)

[测试结果，包括输入和输出 13](#_Toc27927025)

[七、附录 17](#_Toc27927026)

[带注释的源程序 17](#_Toc27927027)

# 一、需求分析

## 1、程序设计的任务和目的

通过这次实验，加深对磁盘调度算法的理解，进一步掌握先来先服务FCFS、最短磁盘调度时间优先SSTF、SCAN和循环SCAN算法的实现方法。

## 2、输入的形式和输入值的范围

磁道个数n和磁道访问序列，开始磁道号m和磁头移动方向（对SCAN和循环SCAN算法有效），算法选择1-FCFS，2-SSTF，3-SCAN，4-循环SCAN。

## 3、输出的形式

每种算法的平均磁盘调度长度。

## 4、程序所能达到的功能

设计程序模拟先来先服务FCFS、最短磁盘调度时间优先SSTF、SCAN和循环SCAN算法的工作过程。假设有n个磁道号所组成的磁道访问序列，给定开始磁道号m和磁头移动的方向（正向或者反向），分别利用不同的磁盘调度算法访问磁道序列，给出每一次访问的磁头移动距离，计算每种算法的平均磁盘调度长度。

## 5、测试数据，包括正确的输入及其输出结果和含有错误的输入及其输出结果

### 正确用例

#### 输入

请输入磁道个数TrackNum：9  
请输入开始磁盘号StartTrack：100  
请输入磁盘第1个访问磁道TrackOrder[1]：55  
请输入磁盘第2个访问磁道TrackOrder[2]：58  
请输入磁盘第3个访问磁道TrackOrder[3]：39  
请输入磁盘第4个访问磁道TrackOrder[4]：18  
请输入磁盘第5个访问磁道TrackOrder[5]：90  
请输入磁盘第6个访问磁道TrackOrder[6]：160  
请输入磁盘第7个访问磁道TrackOrder[7]：150  
请输入磁盘第8个访问磁道TrackOrder[8]：38  
请输入磁盘第9个访问磁道TrackOrder[9]：184  
  
请选择想要先使用的算法（ 1-FCFS，2-SSTF，3-SCAN，4-循环SCAN )：4  
请输入磁头移动方向direction（!0为向外，0为向内）：1  
您选择的磁头移动方向direction为：向外  
您选择的是4-循环SCAN算法

#### 输出

输出磁盘调度算法模拟过程如下：  
当前磁道为：100，下一磁道为：150，磁盘调度方向：向外，磁盘调度距离为： 50，至此平均磁盘调度距离为：50.00  
当前磁道为：150，下一磁道为：160，磁盘调度方向：向外，磁盘调度距离为： 10，至此平均磁盘调度距离为：30.00  
当前磁道为：160，下一磁道为：184，磁盘调度方向：向外，磁盘调度距离为： 24，至此平均磁盘调度距离为：28.00  
当前磁道为：184，下一磁道为： 18，磁盘调度方向：向外，磁盘调度距离为：166，至此平均磁盘调度距离为：62.50  
当前磁道为： 18，下一磁道为： 38，磁盘调度方向：向外，磁盘调度距离为： 20，至此平均磁盘调度距离为：54.00  
当前磁道为： 38，下一磁道为： 39，磁盘调度方向：向外，磁盘调度距离为： 1，至此平均磁盘调度距离为：45.17  
当前磁道为： 39，下一磁道为： 55，磁盘调度方向：向外，磁盘调度距离为： 16，至此平均磁盘调度距离为：41.00  
当前磁道为： 55，下一磁道为： 58，磁盘调度方向：向外，磁盘调度距离为： 3，至此平均磁盘调度距离为：36.25  
当前磁道为： 58，下一磁道为： 90，磁盘调度方向：向外，磁盘调度距离为： 32，至此平均磁盘调度距离为：35.78  
  
输出磁盘调度算法模拟表格如下：  
  
 Simulate 1 Simulate 2 Simulate 3 Simulate 4 Simulate 5 Simulate 6 Simulate 7 Simulate 8 Simulate 9  
TrackIndex 100 150 160 184 18 38 39 55 58  
TrackNext 150 160 184 18 38 39 55 58 90  
Direction Outward Outward Outward Outward Outward Outward Outward Outward Outward  
MoveDistance 50 10 24 166 20 1 16 3 32  
AverageDistance 50.00 30.00 28.00 62.50 54.00 45.17 41.00 36.25 35.78  
算法平均磁盘调度长度为：35.78

### 错误用例（使用始终使用同一磁道）

#### 输入

请输入磁道个数TrackNum：3  
请输入开始磁盘号StartTrack：100  
请输入磁盘第1个访问磁道TrackOrder[1]：100  
请输入磁盘第2个访问磁道TrackOrder[2]：100  
请输入磁盘第3个访问磁道TrackOrder[3]：100  
  
请选择想要先使用的算法（ 1-FCFS，2-SSTF，3-SCAN，4-循环SCAN )：4  
您选择的是2-最短磁盘调度时间优先SSTF算法

#### 输出

输出磁盘调度算法模拟过程如下：  
当前磁道为：100，下一磁道为：100，磁盘调度方向：向内，磁盘调度距离为： 0，至此平均磁盘调度距离为：0.00  
当前磁道为：100，下一磁道为：100，磁盘调度方向：向内，磁盘调度距离为：2147483647，至此平均磁盘调度距离为：1073741823.50  
当前磁道为：100，下一磁道为：2147483647，磁盘调度方向：向外，磁盘调度距离为：2147483547，至此平均磁盘调度距离为：1431655731.33  
  
输出磁盘调度算法模拟表格如下：  
  
 Simulate 1 Simulate 2 Simulate 3  
TrackIndex 100 100 100  
TrackNext 100 100 2147483647  
Direction Inward Inward Outward  
MoveDistance 0 2147483647 2147483547  
AverageDistance 0.00 1073741823.50 1431655731.33  
算法平均磁盘调度长度为：1431655731.33

# 二、概要设计

## 1、抽象数据类型的定义

int TrackNum;//磁道数  
int TrackOrder[MaxNumber];//磁盘访问序列  
bool direction;//磁头移动方向  
int isAlgorithm;//算法选择  
int StartTrack;//开始磁盘号  
  
//定义模拟过程的数据结构  
typedef struct {  
 int TrackIndex;//当前磁道  
 int TrackNext;//下一个磁道  
 bool Direction;//磁头移动方向  
 int MoveDistance;//磁头移动距离  
 double AverageDistance;//平均磁盘调度长度  
} Simulate;  
  
//定义移动距离数据结构  
typedef struct {  
 int TrackIndex;//磁道  
 int MoveDistance;//磁头移动距离  
} Distance;

## 2、主程序的流程

int main() {  
 DiskSchedulingAlgorithm diskSchedulingAlgorithm{};  
  
 diskSchedulingAlgorithm.Input();  
  
 return 0;  
}

## 3、各程序模块之间的层次(调用)关系

int main() {  
 DiskSchedulingAlgorithm diskSchedulingAlgorithm{};  
  
 diskSchedulingAlgorithm.Input();//主函数中调用输入函数  
  
 return 0;  
}  
  
//输入函数调用InputAlgorithm函数选择输入函数  
void Input() {  
 ···  
 InputAlgorithm();  
}  
  
//调用IsAlgorithm函数进行算法存储确认  
void InputAlgorithm() {  
 ···  
 IsAlgorithm();  
}  
  
//算法确认后按照确认结果调用不同算法函数，若确认失败，要求重新输入  
void IsAlgorithm() {  
 ···  
 switch (isAlgorithm) {  
 case 1:  
 AlgorithmFCFS();  
 case 2:  
 AlgorithmSSTF();  
 case 3:  
 AlgorithmSCAN();  
 case 4:  
 AlgorithmCycleSCAN();  
 default:  
 InputAlgorithm();  
 }  
}  
  
//算法函数仅以FCFS为例，在完成算法处理后调用Print函数输出本算法运算结果，并调用NextAlgorithm函数询问下一算法使用  
void AlgorithmFCFS() {  
 ···  
  
 Print();  
  
 NextAlgorithm();  
}  
  
//询问后续，若有则重新调用算法确认函数循环流程，若触发终止条件结束进程  
void NextAlgorithm() {  
 ···  
 IsAlgorithm();  
}

# 三、详细设计

**实现程序模块的具体算法**

## 1、先来先服务FCFS算法

//调用利用先来先服务FCFS算法进行调度计算  
void AlgorithmFCFS() {  
  
 for (int i = 1; i <= TrackNum; i++) {  
 if (i == 1) {  
 //当为第一个磁道磁盘调度时给模拟列表simulate[i]数据结构的各值赋值  
 ···  
 continue;  
 }  
 //当为非第一个磁道磁盘调度时给模拟列表simulate[i]数据结构的各值赋值  
 ···  
 }  
  
 Print();  
  
 NextAlgorithm();  
}

## 2、最短磁盘调度时间优先SSTF算法

//调用最短磁盘调度时间优先SSTF算法进行调度计算  
void AlgorithmSSTF() {  
 //复制空闲分区  
 int TrackOrderCopy[MaxNumber];//复制TrackOrder  
 for (int i = 1; i <= TrackNum; i++) {  
 TrackOrderCopy[i] = TrackOrder[i];  
 }  
  
 for (int i = 1; i <= TrackNum; i++) {  
  
 //计算当前磁道到其它磁道的移动距离  
 for (int j = 1; j <= TrackNum; j++) {  
 distance[j].TrackIndex = TrackOrderCopy[j];  
 if (i == 1) {  
 //计算磁道1到其余磁道的移动距离  
 ···  
 } else if (simulate[i - 1].TrackNext == distance[j].TrackIndex) {  
 //磁道到本身的移动距离计算情况  
 ···  
 } else {  
 //其余移动距离计算情况  
 ···  
 }  
 }  
  
 //按MoveDistance从小到大冒泡排序Distance序列  
 for (int j = 1; j <= TrackNum; j++) {  
 for (int k = 1; k <= TrackNum - j; k++) {  
 ···  
 }  
 }  
 }  
  
 //xie写入simulate模拟数组  
 if (i == 1) {  
 //当为第一个磁道磁盘调度时给模拟列表simulate[i]数据结构的各值赋值  
 ···  
 continue;  
 }  
 //当为非第一个磁道磁盘调度时给模拟列表simulate[i]数据结构的各值赋值  
 ···  
 }  
  
 Print();  
  
 NextAlgorithm();  
}

## 3、SCAN算法

//调用SCAN算法进行调度计算  
void AlgorithmSCAN() {  
 //复制空闲分区  
 int TrackOrderCopy[MaxNumber];//复制TrackOrder  
 for (int i = 1; i <= TrackNum; i++) {  
 TrackOrderCopy[i] = TrackOrder[i];  
 }  
  
 for (int i = 1; i <= TrackNum; i++) {  
  
 //计算当前磁道到其它磁道的移动距离  
 for (int j = 1; j <= TrackNum; j++) {  
 //与上个算法相同不多写了  
 ···  
 }  
  
 //按MoveDistance从小到大冒泡排序Distance序列  
 for (int j = 1; j <= TrackNum; j++) {  
 for (int k = 1; k <= TrackNum - j; k++) {  
 ···  
 }  
 }  
  
 //写入simulate模拟数组  
 for (int j = 1; j <= TrackNum; j++) {  
 //常规情况写入模拟数组，与上一算法相同，不详述  
 ···  
  
 //到达两端最后一个磁道转向，并写入模拟列表  
 if (j == TrackNum) {  
 ···  
 }  
 }  
  
 }  
  
 Print();  
  
 NextAlgorithm();  
}

## 4、循环SCAN算法

//调用循环SCAN算法进行调度计算  
void AlgorithmCycleSCAN() {  
 //复制空闲分区  
 int TrackOrderCopy[MaxNumber];//复制TrackOrder  
 for (int i = 1; i <= TrackNum; i++) {  
 TrackOrderCopy[i] = TrackOrder[i];  
 }  
  
 for (int i = 1; i <= TrackNum; i++) {  
  
 //计算当前磁道到其它磁道的移动距离  
 for (int j = 1; j <= TrackNum; j++) {  
 //与上个算法相同不多写了  
 ···  
 }  
  
 //按MoveDistance从小到大冒泡排序Distance序列  
 for (int j = 1; j <= TrackNum; j++) {  
 for (int k = 1; k <= TrackNum - j; k++) {  
 ···  
 }  
 }  
 }  
   
 //写入simulate模拟数组  
 for (int j = 1; j <= TrackNum; j++) {  
 //常规情况写入模拟数组，与上一算法相同，不详述  
 ···  
  
 //越界，循环跳转  
 if (j == TrackNum) {  
 for (int k = TrackNum; k >= 1; k--) {  
 if (distance[k].TrackIndex != INT\_MAX) {  
 ···  
 break;  
 }  
 }  
 }  
 }  
 }  
  
 Print();  
  
 NextAlgorithm();  
}

# 四、调试分析

## 调试过程中遇到的问题以及解决方法，设计与实现的回顾讨论和分析

|  |  |  |
| --- | --- | --- |
| 问题 | 描述 | 解决方法 |
| bool型输入问题 | 原认为bool型输入，0为false，其余为true，但是存在错误 | 后改用int型输入，使用条件语句给bool型变量赋值 |
| 数组标记问题 | 不存在标记时，SCAN及循环SCAN算法需要多次计算优先级，会造成死循环 | 将已调度磁道标记 |

## 算法的性能分析(包括基本操作和其它算法的时间复杂度和空间复杂度的分析)及其改进设想

### 性能分析

|  |  |  |
| --- | --- | --- |
| 算法 | 时间复杂度 | 空间复杂度 |
| 先来先服务FCFS算法 | T(n) = O(n) | S(n) = O(n) |
| 最短寻道时间优先SSTF算法 | T(n) = O(n3) | S(n) = O(n) |
| SCAN算法 | T(n) = O(n3) | S(n) = O(n) |
| 循环SCAN算法 | T(n) = O(n3) | S(n) = O(n) |

### 改进设想

本实验为6个实验中唯一出现T(n) = O(n3)的时间复杂度的算法，考虑改进设计降低时间复杂度

# 五、用户使用说明

## 使用说明

* 控制台会提示要求用户进行输入，按提示输入内容即可
* 不可输入相同磁道数
* 选择算法输入完成后将会输出磁盘调度过程及模拟列表
* 可根据需要选择是否使用其他算法进行磁盘调度或退出

# 六、测试结果

## 测试结果，包括输入和输出

请输入磁道个数TrackNum：9  
请输入开始磁盘号StartTrack：100  
请输入磁盘第1个访问磁道TrackOrder[1]：55  
请输入磁盘第2个访问磁道TrackOrder[2]：58  
请输入磁盘第3个访问磁道TrackOrder[3]：39  
请输入磁盘第4个访问磁道TrackOrder[4]：18  
请输入磁盘第5个访问磁道TrackOrder[5]：90  
请输入磁盘第6个访问磁道TrackOrder[6]：160  
请输入磁盘第7个访问磁道TrackOrder[7]：150  
请输入磁盘第8个访问磁道TrackOrder[8]：38  
请输入磁盘第9个访问磁道TrackOrder[9]：184  
  
请选择想要先使用的算法（ 1-FCFS，2-SSTF，3-SCAN，4-循环SCAN )：1  
  
您选择的是1-先来先服务FCFS算法  
  
输出磁盘调度算法模拟过程如下：  
当前磁道为：100，下一磁道为： 55，寻道方向：向内，寻道距离为： 45，至此平均寻道距离为：45.00  
当前磁道为： 55，下一磁道为： 58，寻道方向：向外，寻道距离为： 3，至此平均寻道距离为：24.00  
当前磁道为： 58，下一磁道为： 39，寻道方向：向内，寻道距离为： 19，至此平均寻道距离为：22.33  
当前磁道为： 39，下一磁道为： 18，寻道方向：向内，寻道距离为： 21，至此平均寻道距离为：22.00  
当前磁道为： 18，下一磁道为： 90，寻道方向：向外，寻道距离为： 72，至此平均寻道距离为：32.00  
当前磁道为： 90，下一磁道为：160，寻道方向：向外，寻道距离为： 70，至此平均寻道距离为：38.33  
当前磁道为：160，下一磁道为：150，寻道方向：向内，寻道距离为： 10，至此平均寻道距离为：34.29  
当前磁道为：150，下一磁道为： 38，寻道方向：向内，寻道距离为：112，至此平均寻道距离为：44.00  
当前磁道为： 38，下一磁道为：184，寻道方向：向外，寻道距离为：146，至此平均寻道距离为：55.33  
  
输出磁盘调度算法模拟表格如下：  
  
 Simulate 1 Simulate 2 Simulate 3 Simulate 4 Simulate 5 Simulate 6 Simulate 7 Simulate 8 Simulate 9  
TrackIndex 100 55 58 39 18 90 160 150 38  
TrackNext 55 58 39 18 90 160 150 38 184  
Direction Inward Outward Inward Inward Outward Outward Inward Inward Outward  
MoveDistance 45 3 19 21 72 70 10 112 146  
AverageDistance 45.00 24.00 22.33 22.00 32.00 38.33 34.29 44.00 55.33  
算法平均寻道长度为：55.33  
  
请问是否还要进行其余算法，若是，请输入（1-4值）；若否，请输入任意字符（ 1-FCFS，2-SSTF，3-SCAN，4-循环SCAN ）：2  
  
您选择的是2-最短寻道时间优先SSTF算法  
  
输出磁盘调度算法模拟过程如下：  
当前磁道为：100，下一磁道为： 90，寻道方向：向内，寻道距离为： 10，至此平均寻道距离为：10.00  
当前磁道为： 90，下一磁道为： 58，寻道方向：向内，寻道距离为： 32，至此平均寻道距离为：21.00  
当前磁道为： 58，下一磁道为： 55，寻道方向：向内，寻道距离为： 3，至此平均寻道距离为：15.00  
当前磁道为： 55，下一磁道为： 39，寻道方向：向内，寻道距离为： 16，至此平均寻道距离为：15.25  
当前磁道为： 39，下一磁道为： 38，寻道方向：向内，寻道距离为： 1，至此平均寻道距离为：12.40  
当前磁道为： 38，下一磁道为： 18，寻道方向：向内，寻道距离为： 20，至此平均寻道距离为：13.67  
当前磁道为： 18，下一磁道为：150，寻道方向：向外，寻道距离为：132，至此平均寻道距离为：30.57  
当前磁道为：150，下一磁道为：160，寻道方向：向外，寻道距离为： 10，至此平均寻道距离为：28.00  
当前磁道为：160，下一磁道为：184，寻道方向：向外，寻道距离为： 24，至此平均寻道距离为：27.56  
  
输出磁盘调度算法模拟表格如下：  
  
 Simulate 1 Simulate 2 Simulate 3 Simulate 4 Simulate 5 Simulate 6 Simulate 7 Simulate 8 Simulate 9  
TrackIndex 100 90 58 55 39 38 18 150 160  
TrackNext 90 58 55 39 38 18 150 160 184  
Direction Inward Inward Inward Inward Inward Inward Outward Outward Outward  
MoveDistance 10 32 3 16 1 20 132 10 24  
AverageDistance 10.00 21.00 15.00 15.25 12.40 13.67 30.57 28.00 27.56  
算法平均寻道长度为：27.56  
  
请问是否还要进行其余算法，若是，请输入（1-4值）；若否，请输入任意字符（ 1-FCFS，2-SSTF，3-SCAN，4-循环SCAN ）：3  
请输入磁头移动方向direction（!0为向外，0为向内）：1  
您选择的磁头移动方向direction为：向外  
您选择的是3-SCAN算法  
  
输出磁盘调度算法模拟过程如下：  
当前磁道为：100，下一磁道为：150，寻道方向：向外，寻道距离为： 50，至此平均寻道距离为：50.00  
当前磁道为：150，下一磁道为：160，寻道方向：向外，寻道距离为： 10，至此平均寻道距离为：30.00  
当前磁道为：160，下一磁道为：184，寻道方向：向外，寻道距离为： 24，至此平均寻道距离为：28.00  
当前磁道为：184，下一磁道为： 90，寻道方向：向内，寻道距离为： 94，至此平均寻道距离为：44.50  
当前磁道为： 90，下一磁道为： 58，寻道方向：向内，寻道距离为： 32，至此平均寻道距离为：42.00  
当前磁道为： 58，下一磁道为： 55，寻道方向：向内，寻道距离为： 3，至此平均寻道距离为：35.50  
当前磁道为： 55，下一磁道为： 39，寻道方向：向内，寻道距离为： 16，至此平均寻道距离为：32.71  
当前磁道为： 39，下一磁道为： 38，寻道方向：向内，寻道距离为： 1，至此平均寻道距离为：28.75  
当前磁道为： 38，下一磁道为： 18，寻道方向：向内，寻道距离为： 20，至此平均寻道距离为：27.78  
  
输出磁盘调度算法模拟表格如下：  
  
 Simulate 1 Simulate 2 Simulate 3 Simulate 4 Simulate 5 Simulate 6 Simulate 7 Simulate 8 Simulate 9  
TrackIndex 100 150 160 184 90 58 55 39 38  
TrackNext 150 160 184 90 58 55 39 38 18  
Direction Outward Outward Outward Inward Inward Inward Inward Inward Inward  
MoveDistance 50 10 24 94 32 3 16 1 20  
AverageDistance 50.00 30.00 28.00 44.50 42.00 35.50 32.71 28.75 27.78  
算法平均寻道长度为：27.78  
  
请问是否还要进行其余算法，若是，请输入（1-4值）；若否，请输入任意字符（ 1-FCFS，2-SSTF，3-SCAN，4-循环SCAN ）：4  
请输入磁头移动方向direction（!0为向外，0为向内）：1  
您选择的磁头移动方向direction为：向外  
您选择的是4-循环SCAN算法  
  
输出磁盘调度算法模拟过程如下：  
当前磁道为：100，下一磁道为：150，寻道方向：向外，寻道距离为： 50，至此平均寻道距离为：50.00  
当前磁道为：150，下一磁道为：160，寻道方向：向外，寻道距离为： 10，至此平均寻道距离为：30.00  
当前磁道为：160，下一磁道为：184，寻道方向：向外，寻道距离为： 24，至此平均寻道距离为：28.00  
当前磁道为：184，下一磁道为： 18，寻道方向：向外，寻道距离为：166，至此平均寻道距离为：62.50  
当前磁道为： 18，下一磁道为： 38，寻道方向：向外，寻道距离为： 20，至此平均寻道距离为：54.00  
当前磁道为： 38，下一磁道为： 39，寻道方向：向外，寻道距离为： 1，至此平均寻道距离为：45.17  
当前磁道为： 39，下一磁道为： 55，寻道方向：向外，寻道距离为： 16，至此平均寻道距离为：41.00  
当前磁道为： 55，下一磁道为： 58，寻道方向：向外，寻道距离为： 3，至此平均寻道距离为：36.25  
当前磁道为： 58，下一磁道为： 90，寻道方向：向外，寻道距离为： 32，至此平均寻道距离为：35.78  
  
输出磁盘调度算法模拟表格如下：  
  
 Simulate 1 Simulate 2 Simulate 3 Simulate 4 Simulate 5 Simulate 6 Simulate 7 Simulate 8 Simulate 9  
TrackIndex 100 150 160 184 18 38 39 55 58  
TrackNext 150 160 184 18 38 39 55 58 90  
Direction Outward Outward Outward Outward Outward Outward Outward Outward Outward  
MoveDistance 50 10 24 166 20 1 16 3 32  
AverageDistance 50.00 30.00 28.00 62.50 54.00 45.17 41.00 36.25 35.78  
算法平均寻道长度为：35.78  
  
请问是否还要进行其余算法，若是，请输入（1-4值）；若否，请输入任意字符（ 1-FCFS，2-SSTF，3-SCAN，4-循环SCAN ）：0  
  
Process finished with exit code 0

# 七、附录

## 带注释的源程序

#include <iostream>  
#include <iomanip>  
#include <climits>  
  
using namespace std;  
#define MaxNumber 100  
  
class DiskSchedulingAlgorithm {  
public:  
  
 int TrackNum;//磁道数  
 int TrackOrder[MaxNumber];//磁盘访问序列  
// int MoveDistance[MaxNumber];//磁头移动距离  
// double AverageDistance;//平均寻道长度  
 bool direction;//磁头移动方向  
 int isAlgorithm;//算法选择  
 int StartTrack;//开始磁盘号  
  
 //定义模拟过程的数据结构  
 typedef struct {  
 int TrackIndex;//当前磁道  
 int TrackNext;//下一个磁道  
 bool Direction;//磁头移动方向  
 int MoveDistance;//磁头移动距离  
 double AverageDistance;//平均寻道长度  
 } Simulate;  
  
 Simulate simulate[MaxNumber];  
  
 //定义移动距离数据结构  
 typedef struct {  
 int TrackIndex;//磁道  
 int MoveDistance;//磁头移动距离  
 } Distance;  
  
 Distance distance[MaxNumber];  
  
 //输入空闲分区数、空闲的分区大小、进程数、进程需要的分区大小  
 void Input() {  
 cout << "请输入磁道个数TrackNum：";  
 cin >> TrackNum;  
  
 cout << "请输入开始磁盘号StartTrack：";  
 cin >> StartTrack;  
  
 for (int i = 1; i <= TrackNum; i++) {  
 cout << "请输入磁盘第" << i << "个访问磁道" << "TrackOrder[" << i << "]：";  
 cin >> TrackOrder[i];  
 }  
  
 InputAlgorithm();  
 }  
  
 //获取算法选择输入  
 void InputAlgorithm() {  
 cout << endl  
 << "请选择想要先使用的算法（ 1-FCFS，2-SSTF，3-SCAN，4-循环SCAN )：";  
 cin >> isAlgorithm;  
  
 IsAlgorithm();  
 }  
  
 //算法存储确认  
 void IsAlgorithm() {  
  
 if (isAlgorithm == 3 || isAlgorithm == 4) {  
 cout << "请输入磁头移动方向direction（!0为向外，0为向内）：";  
 int in = 0;  
 cin >> in;  
 direction = in != 0;  
 cout << "您选择的磁头移动方向direction为：" << (direction ? "向外" : "向内");  
 }  
  
 switch (isAlgorithm) {  
 case 1:  
 cout << endl << "您选择的是1-先来先服务FCFS算法" << endl;  
 AlgorithmFCFS();  
 break;  
 case 2:  
 cout << endl << "您选择的是2-最短寻道时间优先SSTF算法" << endl;  
 AlgorithmSSTF();  
 break;  
 case 3:  
 cout << endl << "您选择的是3-SCAN算法" << endl;  
 AlgorithmSCAN();  
 break;  
 case 4:  
 cout << endl << "您选择的是4-循环SCAN算法" << endl;  
 AlgorithmCycleSCAN();  
 break;  
 default:  
 cout << "算法值：" << isAlgorithm  
 << "有误,请重新输入正确的算法类型（ 1-FCFS，2-SSTF，3-SCAN，4-循环SCAN ）"  
 << endl;  
 InputAlgorithm();  
 }  
 }  
  
 //询问是否还要进行其余算法  
 void NextAlgorithm() {  
 cout << endl  
 << "请问是否还要进行其余算法，若是，请输入（1-4值）；若否，请输入任意字符（ 1-FCFS，2-SSTF，3-SCAN，4-循环SCAN ）：";  
 cin >> isAlgorithm;  
 if (isAlgorithm != 1 && isAlgorithm != 2 && isAlgorithm != 3 && isAlgorithm != 4) {  
 return;  
 }  
  
 IsAlgorithm();  
 }  
  
 //输出页面置换算法模拟过程及缺页次数与缺页率  
 void Print() {  
  
 cout << endl << "输出磁盘调度算法模拟过程如下：" << endl;  
  
 for (int i = 1; i <= TrackNum; i++) {  
 cout << "当前磁道为：" << right << setw(3) << simulate[i].TrackIndex << "，下一磁道为：" << right << setw(3)  
 << simulate[i].TrackNext << "，寻道方向：" << (simulate[i].Direction ? "向外" : "向内") << "，寻道距离为："  
 << setw(3) << simulate[i].MoveDistance << "，至此平均寻道距离为：" << fixed << setprecision(2)  
 << simulate[i].AverageDistance << endl;  
 }  
  
 cout << endl << "输出磁盘调度算法模拟表格如下：" << endl;  
 //模拟过程  
 cout << endl << left << setw(15) << "";  
 for (int i = 1; i <= TrackNum; i++) {  
 cout << right << setw(12) << "Simulate" << setw(3) << i;  
 }  
  
 cout << endl << left << setw(15) << "TrackIndex";  
 for (int i = 1; i <= TrackNum; i++) {  
 cout << right << setw(15) << simulate[i].TrackIndex;  
 }  
  
 cout << endl << left << setw(15) << "TrackNext";  
 for (int i = 1; i <= TrackNum; i++) {  
 cout << right << setw(15) << simulate[i].TrackNext;  
 }  
  
 cout << endl << left << setw(15) << "Direction";  
 for (int i = 1; i <= TrackNum; i++) {  
 cout << right << setw(15) << (simulate[i].Direction ? "Outward" : "Inward");  
 }  
  
 cout << endl << left << setw(15) << "MoveDistance";  
 for (int i = 1; i <= TrackNum; i++) {  
 cout << right << setw(15) << simulate[i].MoveDistance;  
 }  
  
 cout << endl << left << setw(15) << "AverageDistance";  
 for (int i = 1; i <= TrackNum; i++) {  
 cout << right << setw(15) << setprecision(2) << simulate[i].AverageDistance;  
 }  
  
 cout << endl << "算法平均寻道长度为：" << setprecision(2) << simulate[TrackNum].AverageDistance << endl;  
  
 }  
  
 //调用利用先来先服务FCFS算法进行调度计算  
 void AlgorithmFCFS() {  
  
 for (int i = 1; i <= TrackNum; i++) {  
 if (i == 1) {  
 simulate[i].TrackIndex = StartTrack;  
 simulate[i].Direction = (simulate[i].TrackIndex < TrackOrder[i]);  
 simulate[i].MoveDistance =  
 simulate[i].TrackIndex > TrackOrder[i] ? simulate[i].TrackIndex - TrackOrder[i] :  
 TrackOrder[i] - simulate[i].TrackIndex;  
 simulate[i].AverageDistance = simulate[i].MoveDistance;  
 simulate[i].TrackNext = TrackOrder[i];  
 continue;  
 }  
 simulate[i].TrackIndex = TrackOrder[i - 1];  
 simulate[i].Direction = (simulate[i].TrackIndex < TrackOrder[i]);  
 simulate[i].MoveDistance =  
 simulate[i].TrackIndex > TrackOrder[i] ? simulate[i].TrackIndex - TrackOrder[i] : TrackOrder[i] -  
 simulate[i].TrackIndex;  
 simulate[i].AverageDistance = ((simulate[i - 1].AverageDistance \* (double) (i - 1) / (double) i) +  
 ((double) simulate[i].MoveDistance / (double) i));  
 simulate[i].TrackNext = TrackOrder[i];  
 }  
  
 Print();  
  
 NextAlgorithm();  
 }  
  
 //调用最短寻道时间优先SSTF算法进行调度计算  
 void AlgorithmSSTF() {  
 //复制空闲分区  
 int TrackOrderCopy[MaxNumber];//复制TrackOrder  
 for (int i = 1; i <= TrackNum; i++) {  
 TrackOrderCopy[i] = TrackOrder[i];  
 }  
  
 for (int i = 1; i <= TrackNum; i++) {  
  
 //计算当前磁道到其它磁道的移动距离  
 for (int j = 1; j <= TrackNum; j++) {  
 distance[j].TrackIndex = TrackOrderCopy[j];  
 if (i == 1) {  
 distance[j].MoveDistance =  
 StartTrack > TrackOrderCopy[j] ? StartTrack - TrackOrderCopy[j] : TrackOrderCopy[j] -  
 StartTrack;  
 } else if (simulate[i - 1].TrackNext == distance[j].TrackIndex) {  
 distance[j].MoveDistance = INT\_MAX;  
 TrackOrderCopy[j] = INT\_MAX;  
 } else {  
 distance[j].MoveDistance =  
 simulate[i - 1].TrackNext > TrackOrderCopy[j] ? simulate[i - 1].TrackNext -  
 TrackOrderCopy[j] :  
 TrackOrderCopy[j] - simulate[i - 1].TrackNext;  
 }  
 }  
  
 //按MoveDistance从小到大冒泡排序Distance序列  
 for (int j = 1; j <= TrackNum; j++) {  
 for (int k = 1; k <= TrackNum - j; k++) {  
 if (distance[k].MoveDistance > distance[k + 1].MoveDistance) {  
 Distance temp = distance[k];  
 distance[k] = distance[k + 1];  
 distance[k + 1] = temp;  
 }  
 }  
 }  
  
 //xie写入simulate模拟数组  
 if (i == 1) {  
 simulate[i].TrackIndex = StartTrack;  
 simulate[i].Direction = (simulate[i].TrackIndex < distance[1].TrackIndex);  
 simulate[i].MoveDistance = distance[1].MoveDistance;  
 simulate[i].AverageDistance = simulate[i].MoveDistance;  
 simulate[i].TrackNext = distance[1].TrackIndex;  
 continue;  
 }  
 simulate[i].TrackIndex = simulate[i - 1].TrackNext;  
 simulate[i].Direction = (simulate[i].TrackIndex < distance[1].TrackIndex);  
 simulate[i].MoveDistance = distance[1].MoveDistance;  
 simulate[i].AverageDistance = ((simulate[i - 1].AverageDistance \* (double) (i - 1) / (double) i) +  
 ((double) simulate[i].MoveDistance / (double) i));  
 simulate[i].TrackNext = distance[1].TrackIndex;  
  
 }  
  
 Print();  
  
 NextAlgorithm();  
 }  
  
 //调用SCAN算法进行调度计算  
 void AlgorithmSCAN() {  
 //复制空闲分区  
 int TrackOrderCopy[MaxNumber];//复制TrackOrder  
 for (int i = 1; i <= TrackNum; i++) {  
 TrackOrderCopy[i] = TrackOrder[i];  
 }  
  
 for (int i = 1; i <= TrackNum; i++) {  
  
 //计算当前磁道到其它磁道的移动距离  
// cout << "第" << i << "次计算" << endl;  
 for (int j = 1; j <= TrackNum; j++) {  
 distance[j].TrackIndex = TrackOrderCopy[j];  
 if (i == 1) {  
 distance[j].MoveDistance =  
 StartTrack > TrackOrderCopy[j] ? StartTrack - TrackOrderCopy[j] : TrackOrderCopy[j] -  
 StartTrack;  
 } else if (simulate[i - 1].TrackNext == distance[j].TrackIndex) {  
 distance[j].MoveDistance = INT\_MAX;  
 distance[j].TrackIndex = INT\_MAX;  
 TrackOrderCopy[j] = INT\_MAX;  
 } else {  
 distance[j].MoveDistance =  
 simulate[i - 1].TrackNext > TrackOrderCopy[j] ? simulate[i - 1].TrackNext -  
 TrackOrderCopy[j] :  
 TrackOrderCopy[j] - simulate[i - 1].TrackNext;  
 }  
// cout << distance[j].TrackIndex << " " << distance[j].MoveDistance << " " << endl;  
 }  
  
 //按MoveDistance从小到大冒泡排序Distance序列  
 for (int j = 1; j <= TrackNum; j++) {  
 for (int k = 1; k <= TrackNum - j; k++) {  
 if (distance[k].MoveDistance > distance[k + 1].MoveDistance) {  
 Distance temp = distance[k];  
 distance[k] = distance[k + 1];  
 distance[k + 1] = temp;  
 }  
 }  
 }  
  
/\* cout << "第" << i << "次冒泡" << endl;  
 for (int j = 1; j <= TrackNum; j++) {  
 cout << distance[j].TrackIndex << " " << distance[j].MoveDistance << endl;  
 }\*/  
  
 //写入simulate模拟数组  
 for (int j = 1; j <= TrackNum; j++) {  
 if (i == 1) {  
 if (direction == (StartTrack < distance[j].TrackIndex)) {  
 simulate[i].TrackIndex = StartTrack;  
 simulate[i].Direction = (simulate[i].TrackIndex < distance[j].TrackIndex);  
 simulate[i].MoveDistance = distance[j].MoveDistance;  
 simulate[i].AverageDistance = simulate[i].MoveDistance;  
 simulate[i].TrackNext = distance[j].TrackIndex;  
 break;  
 } else {  
 continue;  
 }  
 }  
  
 if (simulate[i - 1].Direction == (distance[j].TrackIndex > simulate[i - 1].TrackNext) &&  
 distance[j].TrackIndex != INT\_MAX) {  
 simulate[i].TrackIndex = simulate[i - 1].TrackNext;  
 simulate[i].Direction = (simulate[i].TrackIndex < distance[j].TrackIndex);  
 simulate[i].MoveDistance = distance[j].MoveDistance;  
 simulate[i].AverageDistance = ((simulate[i - 1].AverageDistance \* (double) (i - 1) / (double) i) +  
 ((double) simulate[i].MoveDistance / (double) i));  
 simulate[i].TrackNext = distance[j].TrackIndex;  
 break;  
 }  
  
 if (j == TrackNum) {  
 simulate[i].TrackIndex = simulate[i - 1].TrackNext;  
 simulate[i].Direction = (simulate[i].TrackIndex < distance[1].TrackIndex);  
 simulate[i].MoveDistance = distance[1].MoveDistance;  
 simulate[i].AverageDistance = ((simulate[i - 1].AverageDistance \* (double) (i - 1) / (double) i) +  
 ((double) simulate[i].MoveDistance / (double) i));  
 simulate[i].TrackNext = distance[1].TrackIndex;  
 }  
 }  
  
 }  
  
 Print();  
  
 NextAlgorithm();  
 }  
  
 //调用循环SCAN算法进行调度计算  
 void AlgorithmCycleSCAN() {  
 //复制空闲分区  
 int TrackOrderCopy[MaxNumber];//复制TrackOrder  
 for (int i = 1; i <= TrackNum; i++) {  
 TrackOrderCopy[i] = TrackOrder[i];  
 }  
  
 for (int i = 1; i <= TrackNum; i++) {  
  
 //计算当前磁道到其它磁道的移动距离  
// cout << "第" << i << "次计算" << endl;  
 for (int j = 1; j <= TrackNum; j++) {  
 distance[j].TrackIndex = TrackOrderCopy[j];  
 if (i == 1) {  
 distance[j].MoveDistance =  
 StartTrack > TrackOrderCopy[j] ? StartTrack - TrackOrderCopy[j] : TrackOrderCopy[j] -  
 StartTrack;  
 } else if (simulate[i - 1].TrackNext == distance[j].TrackIndex) {  
 distance[j].MoveDistance = INT\_MAX;  
 distance[j].TrackIndex = INT\_MAX;  
 TrackOrderCopy[j] = INT\_MAX;  
 } else {  
 distance[j].MoveDistance =  
 simulate[i - 1].TrackNext > TrackOrderCopy[j] ? simulate[i - 1].TrackNext -  
 TrackOrderCopy[j] :  
 TrackOrderCopy[j] - simulate[i - 1].TrackNext;  
 }  
// cout << distance[j].TrackIndex << " " << distance[j].MoveDistance << " " << endl;  
 }  
  
 //按MoveDistance从小到大冒泡排序Distance序列  
 for (int j = 1; j <= TrackNum; j++) {  
 for (int k = 1; k <= TrackNum - j; k++) {  
 if (distance[k].MoveDistance > distance[k + 1].MoveDistance) {  
 Distance temp = distance[k];  
 distance[k] = distance[k + 1];  
 distance[k + 1] = temp;  
 }  
 }  
 }  
  
/\* cout << "第" << i << "次冒泡" << endl;  
 for (int j = 1; j <= TrackNum; j++) {  
 cout << distance[j].TrackIndex << " " << distance[j].MoveDistance << endl;  
 }\*/  
  
 //写入simulate模拟数组  
 for (int j = 1; j <= TrackNum; j++) {  
 if (i == 1) {  
 if (direction == (StartTrack < distance[j].TrackIndex)) {  
 simulate[i].TrackIndex = StartTrack;  
 simulate[i].Direction = (simulate[i].TrackIndex < distance[j].TrackIndex);  
 simulate[i].MoveDistance = distance[j].MoveDistance;  
 simulate[i].AverageDistance = simulate[i].MoveDistance;  
 simulate[i].TrackNext = distance[j].TrackIndex;  
 break;  
 } else {  
 continue;  
 }  
 }  
  
 if (simulate[i - 1].Direction == (distance[j].TrackIndex > simulate[i - 1].TrackNext) &&  
 distance[j].TrackIndex != INT\_MAX) {  
 simulate[i].TrackIndex = simulate[i - 1].TrackNext;  
 simulate[i].Direction = (simulate[i].TrackIndex < distance[j].TrackIndex);  
 simulate[i].MoveDistance = distance[j].MoveDistance;  
 simulate[i].AverageDistance = ((simulate[i - 1].AverageDistance \* (double) (i - 1) / (double) i) +  
 ((double) simulate[i].MoveDistance / (double) i));  
 simulate[i].TrackNext = distance[j].TrackIndex;  
 break;  
 }  
  
 if (j == TrackNum) {  
 for (int k = TrackNum; k >= 1; k--) {  
 if (distance[k].TrackIndex != INT\_MAX) {  
 simulate[i].TrackIndex = simulate[i - 1].TrackNext;  
 simulate[i].Direction = simulate[i].TrackIndex > distance[k].TrackIndex;  
 simulate[i].MoveDistance = distance[k].MoveDistance;  
 simulate[i].AverageDistance = (  
 (simulate[i - 1].AverageDistance \* (double) (i - 1) / (double) i) +  
 ((double) simulate[i].MoveDistance / (double) i));  
 simulate[i].TrackNext = distance[k].TrackIndex;  
// cout << endl << i << " " << simulate[i].TrackIndex << endl;  
 break;  
 }  
 }  
 }  
 }  
  
 }  
  
 Print();  
  
 NextAlgorithm();  
 }  
};  
  
int main() {  
 DiskSchedulingAlgorithm diskSchedulingAlgorithm{};  
  
 diskSchedulingAlgorithm.Input();  
  
 return 0;  
}