第四章算法实现题

学号：2209060322　　姓名：梁桐　　班级：计算机 2203

4-1

题目：

4-1　会场安排问题。

问题描述：假设要在足够多的会场里安排一批活动，并希望使用尽可能少的会场。设计一个有效的贪心算法进行安排。（这个问题实际上是著名的图着色问题。若将每个活动作为图的一个顶点，不相容活动间用边相连。使相邻顶点着有不同颜色的最小着色数，相当于要找的最小会场数。）

算法设计：对于给定的 k 个待安排的活动，计算使用最少会场的时间表。

数据输入：由文件 input.txt 给出输入数据。第 1 行有 1 个正整数 k，表示有 k 个待安排的活动。接下来的 k 行中，每行有 2 个正整数，分别表示 k 个待安排的活动的开始时间和结束时间。时间以 0 点开始的分钟计。

结果输出：将计算的最少会场数输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
5	3
1 23	
12 28	
25 35	
27 80	
36 50	

解题思路：

这个算法的核心是一个贪心策略，目的是在最少数量的会场中安排一批活动。

输入解析：

首先，我们读取输入的数据，即活动的数量 k 和每个活动的开始和结束时间。

活动信息存储在一个结构体数组中，每个元素包含活动的开始时间、结束时间和一个标志位 flag，用来标记该活动是否已被安排。

贪心策略：

算法的主要思路是通过遍历活动数组，尽可能多地将活动安排到当前会场中，直到没有活动可以放入当前会场为止，然后开启一个新的会场继续安排剩下的活动。

room_avail 变量用来跟踪当前会场的空闲时间。初始时，它为 0，表示该会场从开始就可以使用。

具体步骤：

外部循环不断增加会场数，直到所有活动都被安排。

内部循环遍历每个活动。如果当前活动的开始时间不早于 room_avail，并且该活动还没有被安排（flag == false），则将该活动安排到当前会场，更新 room_avail 为该活动的结束时间，并标记该活动为已安排（flag = true），同时减少剩余未安排的活动数量 count。

一旦所有能安排到当前会场的活动都安排完了，将 room_avail 重置为 0，并在下一次外部循环中开始一个新的会场。

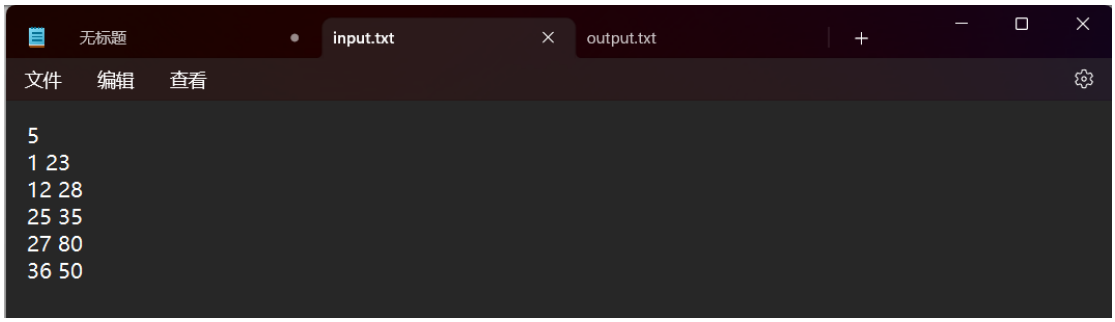
会场计数：

每次完整遍历一次活动列表并安排完一个会场后，room_num（会场计数）增加 1。
循环结束后，room_num 就是安排所有活动所需的最少会场数。
代码：

```
#include <iostream>
#include <fstream>
using namespace std;

struct ans {
    int begin, end;
    bool flag; // 标记活动是否已安排
};

int arrange(int k, ans* a) {
    int count = k, room_avail = 0, room_num = 0;
    while (count > 0) {
        for (int i = 0; i < k; i++) {
            if ((a[i].begin >= room_avail) && (!a[i].flag)) { // 检查活动是否未安排且不
冲突
                room_avail = a[i].end; // 更新会场的空闲时间
                a[i].flag = true;
                count--;
            }
        }
        room_avail = 0; // 重新初始化
        room_num++; // 增加会场计数
    }
    return room_num; // 返回使用的最少会场数
}

int main() {
    ifstream input("input.txt");
    ofstream output("output.txt");

    if (!input.is_open() || !output.is_open()) {
        cerr << "Error opening file!" << endl;
        return 1;
    }

    int k;
    input >> k;
    ans* a = new ans[k]; // 动态分配内存
    for (int i = 0; i < k; i++) {
        input >> a[i].begin >> a[i].end;
        a[i].flag = false;
```

```
        }

        int room_num = arrange(k, a);
        output << room_num << endl;

        delete[] a; // 释放动态内存
        input.close();
        output.close();
        return 0;
}
```

运行结果：



4-5

题目：

4-5 程序存储问题。

问题描述：设有 n 个程序$\{1, 2, \cdots, n\}$要存放在长度为 L 的磁带上。程序 i 存放在磁带上的长度是 l_i（$1 \leqslant i \leqslant n$）。程序存储问题要求确定这 n 个程序在磁带上的一个存储方案，使得能够在磁带上存储尽可能多的程序。

算法设计：对于给定的 n 个程序存放在磁带上的长度，计算磁带上最多可以存储的程序数。

数据输入：由文件 input.txt 给出输入数据。第 1 行是 2 个正整数，分别表示文件个数 n 和磁带的长度 L。接下来的 1 行中，有 n 个正整数，表示程序存放在磁带上的长度。

结果输出：将计算的最多可以存储的程序数输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
6 50	5
2 3 13 8 80 20	

解题思路：

文件输入输出设置：

使用 ifstream 和 ofstream 分别打开 input.txt 和 output.txt 文件，进行输入和输出操作。检查文件是否成功打开，如果未打开则输出错误信息并结束程序。

读取输入：

读取两个整数 n 和 maxLen，其中 n 表示程序的数量，maxLen 表示磁带的最大长度。创建一个数组 lengths，用来存储 n 个程序的长度。

数据读取：

使用 for 循环从文件中读取每个程序的长度并存入 lengths 数组。

排序：

使用 sort(lengths, lengths + n) 对 lengths 数组进行升序排序。这样确保我们从最短的程序开始考虑，以便在磁带中存储更多的程序，这是贪心策略的核心部分。

存储程序的逻辑：

初始化 count 为 0，用于记录已成功存储的程序数量。

初始化 totalLen 为 0，用于记录当前已存储的程序的总长度。

使用 for 循环遍历 lengths 数组：

如果 totalLen + lengths[i] 超过 maxLen，即将当前程序加入后超过磁带长度，则停止循环。

否则，将当前程序的长度加到 totalLen 中，并将 count 增加 1。

输出结果：

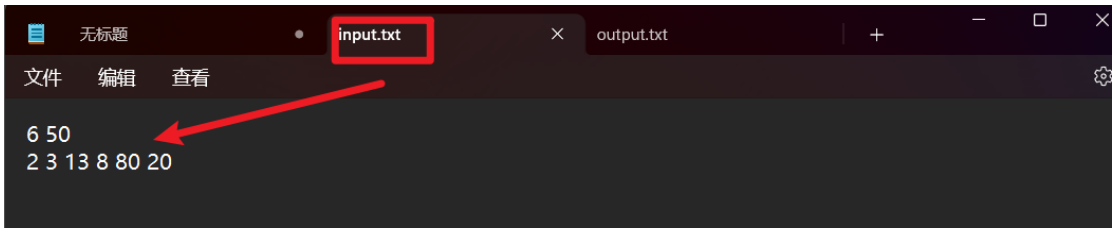
将能存储的最大程序数量 count 写入 output.txt 文件。

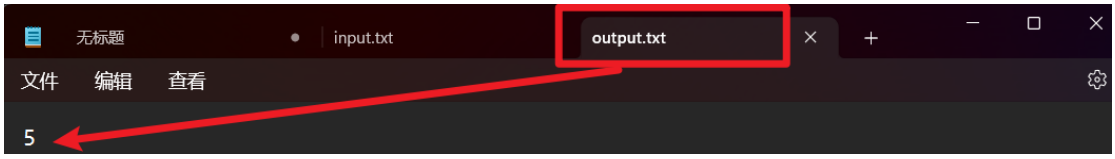
代码：

```
#include <iostream>
#include <fstream>
#include <algorithm>
using namespace std;

const int MAX_SIZE = 10000;

int main() {
```

```
    ifstream input("input.txt");
    ofstream output("output.txt");

    if (!input || !output) {
        cerr << "File error" << endl;
        return 1;
    }

    int n, maxLen;
    input >> n >> maxLen;
    int lengths[MAX_SIZE];

    for (int i = 0; i < n; ++i) {
        input >> lengths[i];
    }

    sort(lengths, lengths + n); // 默认升序排序

    int count = 0, totalLen = 0;

    for (int i = 0; i < n; ++i) {
        if (totalLen + lengths[i] > maxLen) {
            break;
        }
        totalLen += lengths[i];
        ++count;
    }

    output << count << endl;

    return 0;
}
```

运行结果：





4-6

题目：

4-7　多处最优服务次序问题。

问题描述：设有 n 个顾客同时等待一项服务。顾客 i 需要的服务时间为 t_i（$1 \leqslant i \leqslant n$），共有 s 处可以提供此项服务。应如何安排 n 个顾客的服务次序，才能使平均等待时间达到最小？平均等待时间是 n 个顾客等待服务时间的总和除以 n。

算法设计：对于给定的 n 个顾客需要的服务时间和 s 的值，计算最优服务次序。

数据输入：由文件 input.txt 给出输入数据。第 1 行有 2 个正整数 n 和 s，表示有 n 个顾客且有 s 处可以提供顾客需要的服务。接下来的 1 行中有 n 个正整数，表示 n 个顾客需要的服务时间。

结果输出：将计算的最小平均等待时间输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
10　2	336
56 12 1 99 1000 234 33 55 99 812	

思路：

我们需要安排 n 个顾客的服务，以使得平均等待时间最小。每个顾客都有一个独特的服务时间，并且有 s 个服务窗口可以同时为顾客提供服务。合理的分配顾客到服务窗口可以有效减少平均等待时间。1 文件输入输出设置

使用 ifstream 和 ofstream 来分别打开 input.txt 和 output.txt 文件，用于读取输入数据和写入输出结果。

检查文件是否成功打开，若打开失败则输出错误信息并结束程序。

读取输入数据

从文件中读取两个整数 n 和 s

n 顾客数量

s 可用的服务窗口数量

创建一个数组 a 用来存储 n 个顾客的服务时间。

使用 for 循环读取每个顾客的服务时间并存入 a 数组中。

排序服务时间

使用 sort(a, a + n) 对数组 a 进行升序排序。排序的目的是将服务时间短的顾客安排在前面，从而加快整体的服务速度，减少后续顾客的等待时间。

服务分配

使用两个数组 ser 和 sum 来记录服务窗口的当前等待时间和累计等待时间

ser[j] 第 j 个服务窗口的当前总等待时间

sum[j] 第 j 个服务窗口的所有顾客的累计等待时间

使用一个变量 j 来表示当前正在分配顾客的服务窗口，初始值为 0。

遍历每个顾客，依次将顾客分配到窗口

将顾客的服务时间添加到 ser[j] 表示该窗口当前的总等待时间

将当前窗口的总等待时间 ser[j] 加入到 sum[j] 中，表示该窗口的累计等待时间

更新服务窗口索引 j，当到达最后一个窗口时，重置为 0 实现轮流分配顾客。

计算最小平均等待时间

计算所有服务窗口的累计等待时间总和，并存储在 totalWaitTime 变量中。

将总等待时间除以顾客数量 n 来获得平均等待时间。

使用 output << fixed << setprecision(3) 将平均等待时间格式化为保留三位小数的形式，并写入输出文件。

贪心策略的思想

这个算法采用了贪心策略，主要体现在以下几个方面

优先服务时间短的顾客 通过将顾客按服务时间排序，确保最短的服务时间被最早安排，从而为后续顾客减少等待时间

轮流分配窗口 通过轮流将顾客分配到不同的服务窗口，尽量保持各个窗口的负载均衡，避免某个窗口服务时间过长，而其他窗口闲置。

代码：

```
#include <iostream>
#include <fstream>
#include <algorithm>
#include <cstring>
#include <iomanip>
using namespace std;

int main() {
    ifstream input("input.txt");
    ofstream output("output.txt");

    if (!input.is_open() || !output.is_open()) {
        cerr << "Error opening file!" << endl;
        return 1;
    }

    int n, s;
    input >> n >> s;
    int a[1000];
    int ser[100] = { 0 }; // 服务窗口的顾客等待时间
    int sum[100] = { 0 }; // 服务窗口顾客等待时间的总和

    for (int i = 0; i < n; i++) {
        input >> a[i];
    }

    sort(a, a + n); // 对服务时间排序

    int j = 0;
    for (int i = 0; i < n; i++) {
        ser[j] += a[i];
```

```
            sum[j] += ser[j];
            j = (j + 1) % s; // 轮流分配顾客到各服务窗口
        }

        double totalWaitTime = 0;
        for (int i = 0; i < s; i++) {
            totalWaitTime += sum[i];
        }
        totalWaitTime /= n; // 计算平均等待时间

        output << fixed << setprecision(3) << totalWaitTime << endl;

        input.close();
        output.close();
        return 0;
}
```

运行结果：