第三章算法实现题

学号：2209060322　　姓名：梁桐　　班级：计算机 2203

3-1

题目：

3-1　独立任务最优调度问题。

问题描述：用 2 台处理机 A 和 B 处理 n 个作业。设第 i 个作业交给机器 A 处理时需要时间 a_i，若由机器 B 来处理，则需要时间 b_i。由于各作业的特点和机器的性能关系，很可能对于某些 i，有 $a_i \geqslant b_i$，而对于某些 j，$j \neq i$，有 $a_j < b_j$。既不能将一个作业分开由 2 台机器处理，也没有一台机器能同时处理 2 个作业。设计一个动态规划算法,使得这 2 台机器处理完这 n 个作业的时间最短（从任何一台机器开工到最后一台机器停工的总时间）。研究一个实例：$(a_1, a_2, a_3, a_4, a_5, a_6)=(2, 5, 7, 10, 5, 2)$，$(b_1, b_2, b_3, b_4, b_5, b_6)=(3, 8, 4, 11, 3, 4)$。

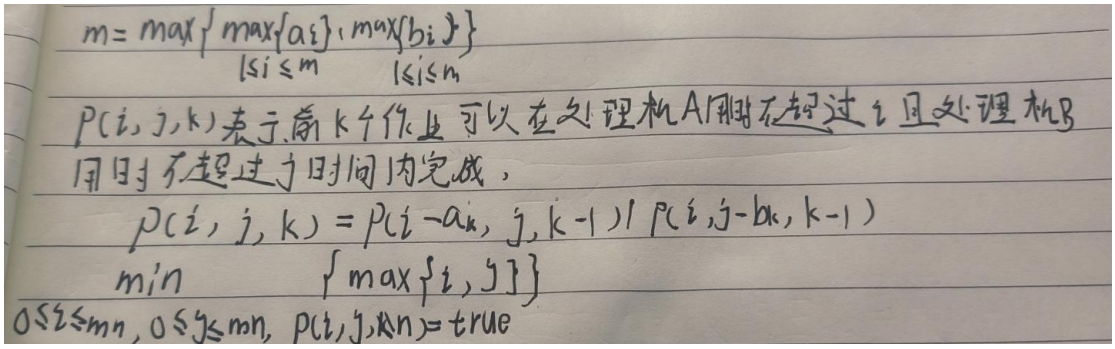
算法设计：对于给定的 2 台处理机 A 和 B 处理 n 个作业,找出一个最优调度方案，使 2 台机器处理完这 n 个作业的时间最短。

数据输入：由文件 input.txt 提供输入数据。文件的第 1 行是 1 个正整数 n，表示要处理 n 个作业。在接下来的 2 行中，每行有 n 个正整数,分别表示处理机 A 和 B 处理第 i 个作业需要的处理时间。

结果输出：将计算出的最短处理时间输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
6	15
2 5 7 10 5 2	
3 8 4 11 3 4	

算法逻辑：



1. 动态规划数组 p 的作用

p 数组的列坐标 k 记录 n 个作业，横坐标 t 表示机器 A 处理所需的时间。

p 数组元素值记录加上机器 B 后，在每次添加一个作业后，机器 B 的最优处理时间。

2. 更新 p 数组的过程

每次更新时有两种情况：

a. 新加入的作业 k 时，如果时间 t 小于 a[k]，则只能由机器 B 处理，更新 p[t][k]为 p[t][k-1] + b[k]。

b. 如果时间 t 大于等于 a[k]，则需要比较：

如果机器 A 不处理 k，p[t][k] = p[t][k-1] + b[k]。

如果机器 A 处理 k，p[t][k] = p[t-a[k]][k-1]。

3. 获取最短处理时间

更新完成后，遍历最后一列，计算每一行的最大值，记录所有最大值的最小值，即为最优解。

代码逻辑：

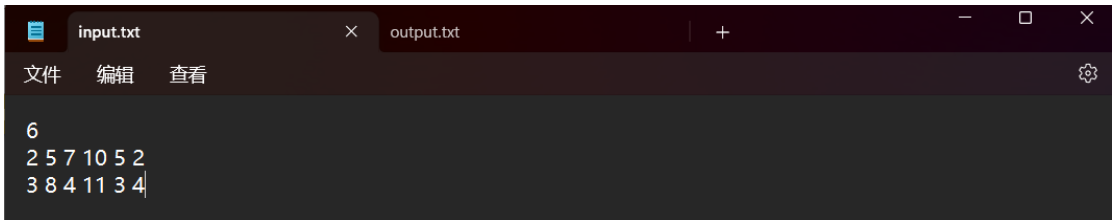
1. 读取输入数据并初始化动态分配的数组 a 和 b。
2. 调用动态规划函数 Dynamic，计算最短处理时间。
3. 将结果输出到文件，并释放动态分配的内存。

代码：

```
#include <iostream>
#include <fstream>
#include <algorithm>
using namespace std;
#define MAXTIME 500

int p[MAXTIME][MAXTIME];

int Dynamic(int a[], int b[], int n) {
    int aTime = 0;
    for (int k = 1; k <= n; k++) {
        aTime += a[k];
        for (int t = 0; t <= aTime; t++) {
            if (t < a[k]) {
                p[t][k] = p[t][k - 1] + b[k];
            }
            else {
                p[t][k] = max(p[t - a[k]][k - 1], p[t][k - 1] + b[k]);
            }
        }
    }
    int minTime = 999999;
    for (int t = 0; t < aTime; t++) {
        int maxt = max(t, p[t][n]);
        minTime = min(minTime, maxt);
    }
    return minTime;
}

int main() {
    ifstream infile("input.txt");
    ofstream outfile("output.txt");
    int n;
```

```
    infile >> n;

    int* a = new int[n + 1];
    int* b = new int[n + 1];

    for (int i = 1; i <= n; i++)
        infile >> a[i];
    for (int i = 1; i <= n; i++)
        infile >> b[i];

    int minTime = Dynamic(a, b, n);
    outfile << minTime;

    delete[] a;
    delete[] b;

    return 0;
}
```

运行结果:



3-2

3-2 最优批处理问题。

问题描述：在一台超级计算机上，编号为$1,2,\cdots,n$的n个作业等待批处理。批处理的任务就是将这n个作业分成若干批，每批包含相邻的若干作业。从时刻0开始，分批加工这些作业。在每批作业开始前，机器需要启动时间S，而完成这批作业所需的时间是单独完成批中各个作业需要时间的总和。单独完成第i个作业所需的时间是t_i，所需的费用是它的完成时刻乘以一个费用系数f_i。同一批作业将在同一时刻完成。例如，如果在时刻T开始一批作业$x,x+1,\cdots,x+k$，则这一批作业的完成时刻均为$T+S+(t_x+t_{x+1}+\cdots+t_{x+k})$。最优批处理问题就是要确定总费用最小的批处理方案。例如，假定有5个作业等待批处理，且

$$S=1,(t_1,t_2,t_3,t_4,t_5)=(1,3,4,2,1),(f_1,f_2,f_3,f_4,f_5)=(3,2,3,3,4)$$

如果采用批处理方案$\{1,2\},\{3\},\{4,5\}$，则各作业的完成时间分别为$(5,5,10,14,14)$，各作业的费用分别为$(15,10,30,42,56)$，因此，这个批处理方案总费用是153。

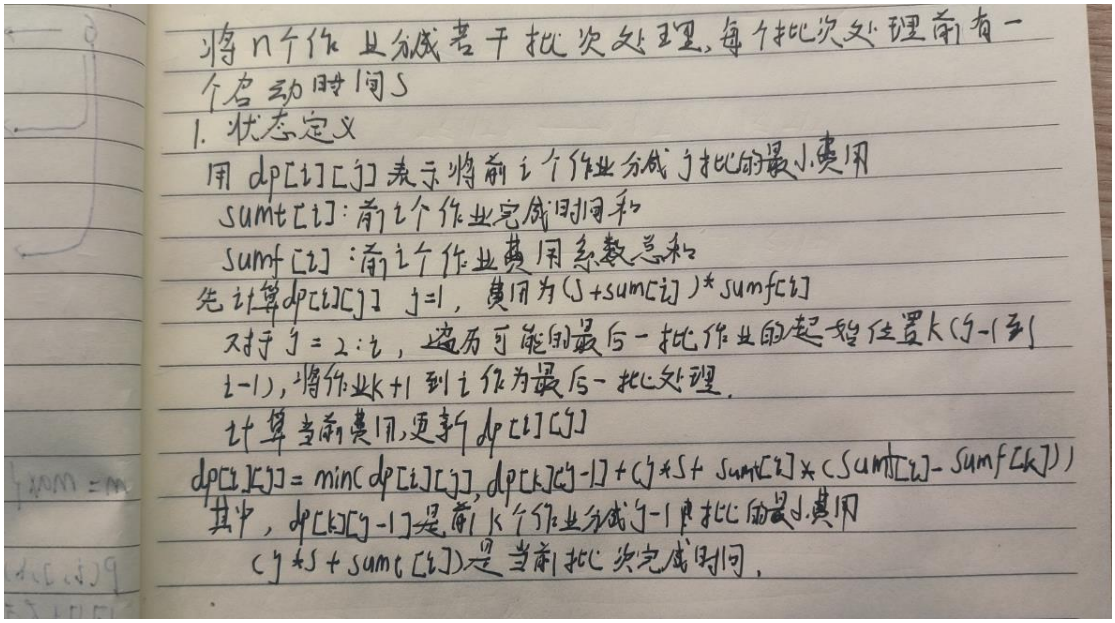
算法设计：对于给定的待批处理的n个作业，计算其总费用最小的批处理方案。

数据输入：由文件input.txt提供输入数据。文件的第1行是待批处理的作业数n，第2行是启动时间S。接下来每行有2个数，分别为单独完成第i个作业所需的时间是t_i和所需的费用系数f_i。

结果输出：将计算出的最小总费用输出到文件output.txt中。

输入文件示例	输出文件示例
input.txt	output.txt
5	153
1	
1 3	
3 2	
4 3	
2 3	
1 4	

算法思想:

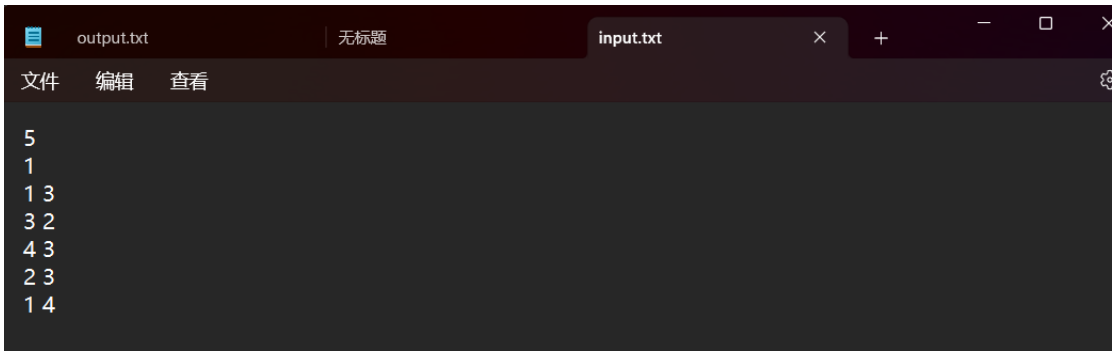


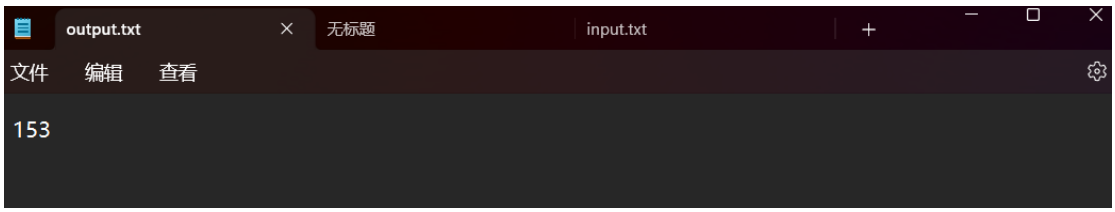
代码：

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

#define N 100
int n, S, ans = 99999;
int sumt[N], sumf[N], dp[N][N]; // dp[i][j]表示将前 i 个作业分成 j 批的最小费用
struct obj {
    int t;
    int f;
};

int main() {
    // 打开输入输出文件
    ifstream infile("input.txt");
    ofstream outfile("output.txt");

    infile >> n >> S;
    obj o[N]; // 使用数组存储作业
    for (int i = 1; i <= n; i++) {
        infile >> o[i].t >> o[i].f;
        sumt[i] = sumt[i - 1] + o[i].t;    // 引入前缀和记录完成前 i 个作业所需的总时间（不含启动）
        sumf[i] = sumf[i - 1] + o[i].f;     // 表示前 i 个作业的总费用系数
    }

    memset(dp, 0x3f, sizeof(dp));
    dp[1][1] = (S + o[1].t) * o[1].f;
    for (int i = 2; i <= n; i++) {
        dp[i][1] = (S + sumt[i]) * sumf[i];
        for (int j = 2; j <= i; j++) {
            // 遍历倒数第一批与前面分段的位置
            for (int k = j - 1; k < i; k++) {
                dp[i][j] = min(dp[i][j], dp[k][j - 1] + (j * S + sumt[i]) * (sumf[i] - sumf[k]));
            }
        }
    }

    for (int j = 1; j <= n; j++) ans = min(ans, dp[n][j]);
    outfile << ans; // 输出结果到文件
```

```
    infile.close();
    outfile.close();
    return 0;
}
```

运行结果：





3-8

3-8　最小 m 段和问题。

问题描述：给定 n 个整数组成的序列，现在要求将序列分割为 m 段，每段子序列中的数在原序列中连续排列。如何分割才能使这 m 段子序列的和的最大值达到最小？

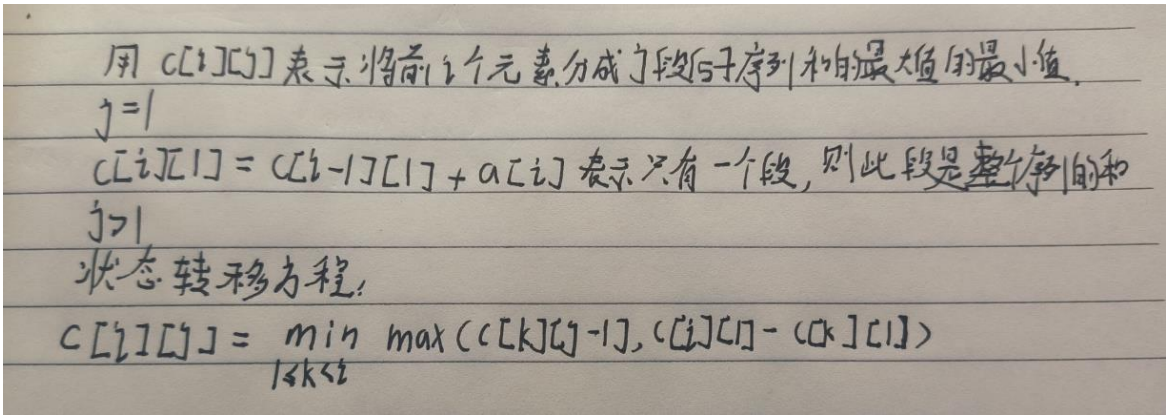
算法设计：给定 n 个整数组成的序列，计算该序列的最优 m 段分割，使 m 段子序列的和的最大值达到最小。

数据输入：由文件 input.txt 提供输入数据。文件的第 1 行中有 2 个正整数 n 和 m。正整数 n 是序列的长度；正整数 m 是分割的段数。接下来的一行中有 n 个整数。

结果输出：将计算结果输出到文件 output.txt。文件的第 1 行中的数是计算出的 m 段子序列的和的最大值的最小值。

输入文件示例	输出文件示例
input.txt	output.txt
1 1	10
10	

算法思想：



代码实现：

```
#include <iostream>
#include <fstream>
#include <algorithm> // for std::max and std::min
#include <climits>     // for INT_MAX
using namespace std;

int c[100][100]; //  存储 0~i 的 j 个分组的和的最大值的最小值

int M(int a[], int n, int m) {
    for (int i = 1; i <= n; i++) { // j=1 即只有 1 段，计算整个序列的和
        c[i][1] = c[i - 1][1] + a[i];
    }

    for (int j = 2; j <= m; j++) { //  分组数从 2 开始
        for (int i = 1; i <= n; i++) {
            int min_val = INT_MAX; //  使用 INT_MAX 替代 99999
            for (int k = 1; k < i; k++) { // k 表示分开的位置
                int t = max(c[k][j - 1], c[i][1] - c[k][1]);
                min_val = min(min_val, t);
            }
            c[i][j] = min_val;
        }
    }

    return c[n][m];
}

int main() {
    // 打开输入输出文件
    ifstream infile("input.txt");
```

```
    ofstream outfile("output.txt");

    int n, m;
    infile >> n >> m; // 从文件读取 n 和 m
    int a[100];
    for (int i = 1; i <= n; i++) {
        infile >> a[i]; // 从文件读取序列
    }

    M(a, n, m);
    outfile << c[n][m]; // 输出结果到文件

    infile.close();
    outfile.close();
    return 0;
}
```

运行结果：