第三、四章作业

学号：202211020213 姓名：赵燮

第三章

3-1 独立任务最优调度问题

问题描述

用两台处理机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ai，若由机器B来处理，则需要时间bi。由于各作业的特点和机器的性能关系，可能对于某些i，有ai≥bi，而对于某些j（j≠i），有aj<bj。既不能将一个作业分开由两台机器处理，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，使得这两台机器处理完这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：
（a1,a2,a3,a4,a5,a6)=（2,5,7,10,5,2），（b1,b2,b3,b4,b5,b6）=（3,8,4,11,3,4)。
算法设计：对于给定的两台处理机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。

问题分析

在本问题中，我们需要使用 **动态规划** 来寻找最优调度方案，使得两台机器处理完所有作业所需的最大时间最小化。具体而言，我们需要根据每个作业在两台机器上的处理时间，决定如何合理分配任务。

动态规划状态定义：

- `p(i, j, k)`：表示前 `k` 个作业在机器 A 上的总时间不超过 `i`，机器 B 上的总时间不超过 `j`，是否能够完成。

状态转移：

- 对于第 `k` 个作业，有两个选择：
  1. 将第 `k` 个作业分配给机器 A，这时机器 A 的处理时间增加 `a[k]`，机器 B 的处理时间不变。
  2. 将第 `k` 个作业分配给机器 B，这时机器 B 的处理时间增加 `b[k]`，机器 A 的处理时间不变。
  通过状态转移公式：
- `p(i, j, k) = p(i - a[k], j, k - 1)` 表示将第 `k` 个作业分配给机器 A。
- `p(i, j, k) = p(i, j - b[k], k - 1) | p(i, j, k)` 表示将第 `k` 个作业分配给机器 B。

计算过程：

- 对每一对 `(i, j)`，我们检查是否可以通过前 `k` 个作业来更新当前的最优方案，最终计算出最短的最大完成时间，即 `max(i, j)`。

代码

```
#include<bits/stdc++.h>
using namespace std;

const int N = 100;
int a[N];
int b[N];
bool p[N][N][N];

int main(){
    ifstream ifs("input.txt");
    ofstream ofs("output.txt");
    int n;
    ifs >> n;
    int a_sum = 0;
    int b_sum = 0;
    int ans = 9999999;

    for(int i=1;i<=n;i++){   //输入a[]
        ifs >> a[i];
        a_sum += a[i];
    }
    for(int i=1;i<=n;i++){   //输入b[]
        ifs >> b[i];
        b_sum += b[i];
    }

    for(int i=0;i<=a_sum;i++){ //初始化p[i][j][k],当k=0时，均为true，其余均为fal
        for(int j=0;j<=b_sum;j++){
            p[i][j][0]=true;
            for(int k=1;k<=n;k++){
                p[i][j][k]=false;
            }
        }
    }
    for(int i=0;i<=a_sum;i++){
        for(int j=0;j<=b_sum;j++){
            for(int k=1;k<=n;k++){
                if(i>=a[k]) p[i][j][k] = p[i-a[k]][j][k-1];  //先判断i>=a[k]
                if(j>=b[k]) p[i][j][k] = p[i][j-b[k]][k-1]|p[i][j][k]; //这里
                if(k==n && p[i][j][k]){  //k==n，处理到最后一个，p[i][j][k]==tr
                    if(ans>max(i,j)){    //比较ans与max(i,j)来判断是否需要更新答
```

```
                        ans = min(ans,max(i,j));
                    }
                }
            }
        }
    }
    ofs << ans;

    ifs.close();
    ofs.close();
}
```

input

```
6
2 5 7 10 5 2
3 8 4 11 3 4
```

output

```
15
```

3-7 汽车加油行驶问题

问题描述

给定一个NxN的方形网格，设其左上角为起点O，坐标为（1，1），X轴向右为正，Y轴向下为正，每个方格边长为1。一辆汽车从起点○出发驶向右下角终点，其坐标为(N，M)。在若干网格交叉点处，设置了油库，可供汽车在行驶途中加油。汽车在行驶过程中应遵守如下规则：
（1）汽车只能沿网格边行驶，装满油后能行驶K条网格边。出发时汽车已装满油，在起点与终点处不设油库。
（2）当汽车行驶经过一条网格边时，若其X坐标或Y坐标减小，则应付费用B，否则免付费用。
（3）汽车在行驶过程中遇油库则应加满油并付加油费用A。
（4）在需要时可在网格点处增设油库，并付增设油库费用C（不含加油费用A）.
（5）（1）~（4）中的各数N、K、A、B、C均为正整数
求汽车从起点出发到达终点的一条所付费用最少的行驶路线。

问题分析

- 使用一个二维数组 `minCost[x][y]` 来记录到达每个坐标 `(x, y)` 时的最小花费。
  - 另外使用一个二维数组 `remainingOil[x][y]` 来记录到达某个位置时的剩余油量，以便后续的判断和更新。

- **BFS遍历**：
  - 从起点 `(1,1)` 开始，依次探索每个可能的状态。对每个状态的邻居进行更新：
    - 如果当前油量足够，继续沿着网格边走；
    - 如果到达油库，重新加油；
    - 如果油量耗尽，则可能需要增加油库来加油。
  - 每次更新时，检查是否可以通过当前节点到达一个新的状态，并且是否需要更新最小花费和剩余油量。
- **剪枝策略**：
  - 如果当前状态的花费已经超过了已知的最小花费，则不再探索该状态。
  - 如果当前状态的花费等于已知的最小花费，但剩余油量更多，则也有可能继续探索该路径，防止未来由于油量不足而导致额外的花费。
- **终止条件**：
  - 当遍历完所有状态后，最终的最小花费将存储在 `minCost[n][n]` 中，即从起点 `(1, 1)` 到终点 `(n, n)` 的最小花费。

代码

```
#include <bits/stdc++.h>
using namespace std;

// 辅助函数：返回两个数中的最小值
int mymin(int x, int y) {
    return (x < y) ? x : y;
}

// 定义四个方向的移动：上、左、下、右
const int dx[4] = { -1, 0, 1, 0 };
const int dy[4] = { 0, -1, 0, 1 };

// 最大网格大小
const int MAX_SIZE = 110;

// 定义状态结构体，用于在BFS中表示当前节点的状态
struct Node {
    int x, y;  // 当前坐标
    int cost;  // 累计的花费
    int oil;   // 剩余油量
} queueList[8110010];  // BFS队列

// 全局变量
int n, k, a, b, c;                // 网格大小、初始油量、加油费用A、费用B、增设油库
int head, tail;                    // BFS队列的头尾指针
int Map[MAX_SIZE][MAX_SIZE];       // 地图，标记油库位置（1表示有油库，0表示没有油库
int minCost[MAX_SIZE][MAX_SIZE];   // 到达每个位置的最小花费
int remainingOil[MAX_SIZE][MAX_SIZE]; // 达到每个位置时的剩余油量
```

```
int main() {
    // 文件输入输出流
    ifstream ifs("input.txt");
    ofstream ofs("output.txt");

    // 读取输入参数：网格大小n、初始油量k、费用A（加油费用）、B（方向费用）、C（增设油库
    ifs >> n >> k >> a >> b >> c;

    // 读取地图信息
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++) {
            ifs >> Map[i][j];  // 读取每个位置是否有油库
        }
    }
    ifs.close();  // 关闭输入文件

    // 初始化最小花费数组为一个很大的值（表示未访问）
    memset(minCost, 63, sizeof(minCost));
    minCost[1][1] = 0;           // 起点的花费为0
    remainingOil[1][1] = k;      // 起点时油量为满油

    // 初始化BFS队列，起点入队
    queueList[1].x = 1;
    queueList[1].y = 1;
    queueList[1].cost = 0;
    queueList[1].oil = k;
    head = 1;
    tail = 2;

    // 使用广度优先搜索（BFS）遍历所有可能的路径
    while (head != tail) {
        // 遍历四个方向（上、左、下、右）
        for (int t = 0; t < 4; t++) {
            Node current = queueList[head];  // 获取当前节点
            // 计算新的坐标
            int newX = current.x + dx[t];
            int newY = current.y + dy[t];

            // 检查新坐标是否在网格内
            if (newX >= 1 && newX <= n && newY >= 1 && newY <= n) {
                // 创建新的节点
                Node nextNode;
                nextNode.x = newX;
                nextNode.y = newY;
                nextNode.cost = current.cost;       // 初始花费为当前节点的花费
                nextNode.oil = current.oil - 1;    // 移动时油量减少1

                // 如果到达终点
                if (newX == n && newY == n) {
                    // 更新终点的最小花费
```

```
                minCost[n][n] = mymin(minCost[n][n], nextNode.cost);
                continue; // 到达终点后无需继续搜索
            }

            // 如果当前位置有油库，支付加油费用A，并将油量加满
            if (Map[newX][newY] == 1) {
                nextNode.cost += a;  // 支付加油费用A
                nextNode.oil = k;    // 油量加满
            }
            // 如果油量耗尽，需要增加油库并支付费用C+A
            else if (nextNode.oil == 0) {
                nextNode.cost += (c + a);  // 增设油库并支付增设油库费用C和加
                nextNode.oil = k;          // 油量加满
            }

            // 如果移动方向是上或左，需要支付费用B
            if (t <= 1) {  // t=0 (上)，t=1 (左)
                nextNode.cost += b; // 支付方向费用B
            }

            // 剪枝：如果当前路径的花费已经超过已知的最小花费，跳过
            if (nextNode.cost > minCost[n][n]) {
                continue;
            }

            // 如果找到更小的花费到达新位置，更新并入队
            if (nextNode.cost < minCost[nextNode.x][nextNode.y]) {
                minCost[nextNode.x][nextNode.y] = nextNode.cost;
                remainingOil[nextNode.x][nextNode.y] = nextNode.oil;
                queueList[tail++] = nextNode;  // 将新节点入队
            }
            // 如果花费相同但剩余油量更多，考虑将新节点入队
            else if (nextNode.oil > remainingOil[nextNode.x][nextNode.y]
                     nextNode.cost < minCost[nextNode.x][nextNode.y] + a
                queueList[tail++] = nextNode;
            }

            // 处理循环队列：当队列尾部超过最大大小时重置尾指针
            if (tail == 8110010) {
                tail = 1;
            }
        }
    }

    // 移动队列头指针，准备处理下一个节点
    if (++head == 8110010) {
        head = 1;
    }
}
```

```
    // 输出到达终点的最小花费
    ofs << minCost[n][n] << endl;

    ofs.close();  // 关闭输出文件
    return 0;
}
```

input

```
9 3 2 3 6
0 0 0 0 1 0 0 0 0
0 0 0 1 0 1 1 0 0
1 0 1 0 0 0 0 1 0
0 0 0 0 0 1 0 0 1
1 0 0 1 0 0 1 0 0
0 1 0 0 0 0 0 1 0
0 0 0 0 1 0 0 0 1
1 0 0 1 0 0 0 1 0
0 1 0 0 0 0 0 0 0
```

output

```
12
```

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？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。

问题分析

为了找到 `m` 段的最优分割方案，我们需要将序列划分成 `m` 段，使得每一段的和尽可能小。我们可以通过 **动态规划**（DP）来求解。

状态定义

定义 `dp[i][j]` 表示将前 `i` 个数分割成 `j` 段的最小可能最大值。具体来说，`dp[i][j]` 代表将前 `i` 个元素分割成 `j` 段后，所得到的 `m` 段中的最大和的最小值。

状态转移

1. **初始状态**：
   - `dp[i][1]` 表示将前 `i` 个元素分割为 1 段的最大和，即 `dp[i][1] = sum(1, i)`，即前 `i` 个元素的和。
2. **转移公式**：
   对于任意 `i` 和 `j`（`i >= j`），可以通过以下公式计算 `dp[i][j]`：
   - `dp[i][j] = min{ max(dp[k][j-1], sum(k+1, i)) }`，其中 `0 < k < i`，即对于每个位置 `k`，我们尝试把前 `k` 个数分成 `j-1` 段，把后面的从 `k+1` 到 `i` 个数作为第 `j` 段，选择使得 `dp[i][j]` 最小的 `k`。
   - `dp[i][1]` 是前 `i` 个数的和，可以通过前缀和数组来快速计算。
3. **时间复杂度分析**：
   - 对于每个 `dp[i][j]`，我们需要遍历前 `i` 个元素来计算最优的 `k`，因此总时间复杂度是 `O(n^2 * m)`，其中 `n` 是序列的长度，`m` 是分段的数量。

代码

```
#include <bits/stdc++.h>
using namespace std;

int dp[500][500];
int num[500];
int n, m,temp, maxt;

int main()
{
    ifstream ifs("input.txt");
    ofstream ofs("output.txt");
    ifs >> n >> m;
    if (n < m || n == 0){
        ofs << 0 << endl;
        return 0;
    }
    for (int i = 1; i <= n; i++) {
        ifs >> num[i];
        dp[i][1] = dp[i - 1][1] + num[i];  //初始化dp[][]数组，这里采用前缀和的方
    }
    for (int j = 2; j <= m; j++){
        for (int i = j; i <= n; i++){
            temp = INT_MAX;
            for (int k = 1; k < i; k++){   //对于任意一个dp[i][j],取k=[1,i)遍历
                maxt = max(dp[i][1] - dp[k][1], dp[k][j - 1]);
                temp = min(temp,maxt);
            }
            dp[i][j] = temp;
        }
    }
    ofs << dp[n][m] << endl;                //dp[n][m]：前n个数字分为m段的最小最
```

```
    ifs.close();
    ofs.close();
}
```

input

```
5 2
4 2 1 4 5
```

output

```
9
```

3-15 收集样本问题

问题描述

机器人Rob在一个有nxn个方格的方形区域F中收集样本。(i,j)方格中样本 的价值为v(i,j)，如图3-6所示。Rob从方形区域F的左上角A点出发，向下或向右行走，直到右下角的B点，在走过的路上，收集方格中的样本。 Rob从A点到B点共走2次，试找出Rob的2条行走路径 使其取得的样本总价值最大。 给定方形区域F中的样本分布，计算Rob 的2条行走路径，使其取得的样本总价值最大。

问题分析

- 假设 `h[x1][y1][x2][y2]` 代表两个机器人的状态：一个机器人从起点 A 到 (x1, y1)，另一个机器人从起点 A 到 (x2, y2)，这两个机器人从 (1, 1) 到 (n, n) 的路径上的最大样本总价值。
- 我们的目标是通过动态规划计算出 `h[n][n][n][n]`，即最终两个机器人都到达终点 B 后能获得的最大样本价值。

状态转移：

- 在某一时刻，我们知道机器人在位置 (x1,y1)(x1, y1)(x1,y1) 和 (x2,y2)(x2, y2)(x2,y2)，我们需要计算出两个机器人从当前状态走到下一个状态后可能获得的最大价值。
- 由于机器人每次只能向下或向右走，因此可以有 4 种状态转移：
    1. (x1+1,y1)(x1+1, y1)(x1+1,y1) 和 (x2+1,y2)(x2+1, y2)(x2+1,y2)：两个机器人都向下走。
    2. (x1+1,y1)(x1+1, y1)(x1+1,y1) 和 (x2,y2+1)(x2, y2+1)(x2,y2+1)：第一个机器人向下，第二个机器人向右。
    3. (x1,y1+1)(x1, y1+1)(x1,y1+1) 和 (x2+1,y2)(x2+1, y2)(x2+1,y2)：第一个机器人向右，第二个机器人向下。

4. (x1,y1+1)(x1, y1+1)(x1,y1+1) 和 (x2,y2+1)(x2, y2+1)(x2,y2+1)：两个机器人都向右走。

- 在每次更新状态时，我们需要保证：
  - 目标是最大化两个机器人路径上获得的样本总价值。
  - 当两个机器人在同一个位置时，要考虑是否重复收集该位置的样本。

代码

```
#include<bits/stdc++.h>
using namespace std;

int n;
int h[100][100][100][100];
int g[100][100];

void val(int x1, int y1, int x2, int y2, int v) {
    if (x1 == x2 && y1 == y2) {
        h[x1][y1][x2][y2] = max(h[x1][y1][x2][y2], v + g[x1][y1]);
    } else {
        h[x1][y1][x2][y2] = max(h[x1][y1][x2][y2], v + g[x1][y1] + g[x2][y2]
    }
}

void func() {
    // 初始化h数组为负无穷
    for (int i = 0; i <= n; i++) {
        for (int j = 0; j <= n; j++) {
            for (int k = 0; k <= n; k++) {
                for (int l = 0; l <= n; l++) {
                    h[i][j][k][l] = -1e9; // 设置为一个很小的数，表示无效路径
                }
            }
        }
    }

    // 初始位置
    h[1][1][1][1] = g[1][1];

    // 状态转移
    for (int s = 2; s <= n + n - 1; s++) { // s表示走的步数
        for (int x1 = 1; x1 <= min(s, n); x1++) {
            int y1 = s - x1;
            if (y1 < 1 || y1 > n) continue; // 检查坐标是否合法

            for (int x2 = 1; x2 <= min(s, n); x2++) {
                int y2 = s - x2;
                if (y2 < 1 || y2 > n) continue; // 检查坐标是否合法
```

```
                // 计算当前状态
                if (h[x1][y1][x2][y2] != -1e9) {
                    int v = h[x1][y1][x2][y2];
                    if (x1 + 1 <= n && x2 + 1 <= n) {
                        val(x1 + 1, y1, x2 + 1, y2, v); // 1. x1, y1 向下，x2
                    }
                    if (x1 + 1 <= n && y2 + 1 <= n) {
                        val(x1 + 1, y1, x2, y2 + 1, v); // 2. x1, y1 向下，x2
                    }
                    if (y1 + 1 <= n && x2 + 1 <= n) {
                        val(x1, y1 + 1, x2 + 1, y2, v); // 3. x1, y1 向右，x2
                    }
                    if (y1 + 1 <= n && y2 + 1 <= n) {
                        val(x1, y1 + 1, x2, y2 + 1, v); // 4. x1, y1 向右，x2
                    }
                }
            }
        }
    }
}

int main() {
    ifstream ifs("input.txt");
    ofstream ofs("output.txt");
    ifs >> n;

    int x, y, c;
    while (1) {
        ifs >> x >> y >> c;
        if (x == 0 && y == 0 && c == 0) break;
        g[x][y] = c;
    }

    func();
    ofs << h[n][n][n][n] << endl;

    ifs.close();
    ofs.close();
}
```

input

```
8
2 3 13
2 6 6
3 5 7
4 4 14
```

```
5 2 21
5 6 4
6 3 15
7 2 14
0 0 0
```

output

```
67
```

3-17 字符串比较问题

问题描述

对于长度相同的两个字符串A和B，其距离定义为相应位置字符距离之和。两个非空格字符的距离是它们的ASCII编码之差的绝对值。空格与空格的距离为0，空格与其他字符的距离为一定值k。
在一般情况下，字符串A和B的长度不一定相同。字符串A的扩展是在A中插入若干空格字符所产生的字符串。在字符串A和B的所有长度相同的扩展中，有一对距离最小的扩展，该距离称为字符串A和B的扩展距离。对于给定的字符串A和B，试设计一个算法，计算其扩展距离。

问题分析

对于给定的两个字符串A,B，我们可以通过在A、B中添加一些空格，来使得在ASCII码上相近的字符尽量在相对应的位置，来使得两个字符串的相对距离较小。
于是，对于字符串`A[0:i]`和`B[0:j]`现只分析最后一位，我们可以：

1. 不添加空格，让`A[i]`和`B[j]`直接配对，于是`dp[i][j]=dp[i-1][j-1]+dist(A[i],B[j])`
2. 在A后添加空格与`B[j]`匹配，于是`dp[i][j]=dp[i][j-1]+blank`
3. 在B后添加空格与`A[i]`匹配，于是`dp[i][j]=dp[i-1][j]+blank`

于是状态转移方程为：
`dp(i,j)=min{dp(i-1,j)+blank,dp(i,j-1)+blank,dp(i-1,j-1)+dist(A[i],B[j]))`

代码

```
#include<bits/stdc++.h>
using namespace std;

int dp[500][500]; //dp[i][j]记录A[1:i]和B[1:j]的最小扩展距离

int main(){
    string a,b;
```

```
    int blank,sum=0;
    ifstream ifs("input.txt");
    ofstream ofs("output.txt");
    ifs >> a >> b >> blank;
    int a_len = a.length();
    int b_len = b.length();
    dp[0][0]=0;
    int tmp;
    for (int i = 0; i <= a_len; i++) {
        for (int j = 0; j <= b_len; j++) {
            if (i + j) { //排除dp[0][0]，已在前面定义
                dp[i][j] = INT_MAX;
                if ((i * j) && (tmp = dp[i-1][j-1] + abs(a[i-1]-b[j-1])) < d
                    dp[i][j] = tmp;
                if (i && (tmp = dp[i-1][j] + blank) < dp[i][j])    //i不为0，防
                    dp[i][j] = tmp;
                if (j && (tmp = dp[i][j-1] + blank) < dp[i][j])    //j不为0，防
                    dp[i][j] = tmp;
            }
        }
    }
    ofs<<dp[a_len][b_len];
    ifs.close();
    ofs.close();
}
```

input

```
xbcd
bcdr
2
```

output

```
4
```

3-25 m处理器问题

问题描述

问题描述：在网络通信系统中，要将 n 个数据包依次分配给 m 个处理器进行数据处理，并要求处理器负载尽可能均衡。设给定的数据包序列为 $\{\sigma_0, \sigma_1, \cdots, \sigma_{n-1}\}$。$m$ 处理器问题要求的是 $r_0=0 \leqslant r_1 \leqslant \cdots \leqslant r_{m-1} \leqslant n=r_m$，将数据包序列划分为 m 段：$\{\sigma_0, \cdots, \sigma_{r_1-1}\}$，$\{\sigma_{r_1}, \cdots, \sigma_{r_2-1}\}$，$\cdots$，

$\{\sigma_{r_{m-1}},\cdots,\sigma_{n-1}\}$，使 $\max_{i=0}^{m-1}\{f(r_i,r_{i+1})\}$ 达到最小。式中，$f(i,j)=\sqrt{\sigma_i^2+\cdots+\sigma_j^2}$ 是序列 $\{\sigma_i,\cdots,\sigma_j\}$ 的负载量。

$\max_{i=0}^{m-1}\{f(r_i,r_{i+1})\}$ 的最小值称为数据包序列 $\{\sigma_0,\sigma_1,\cdots,\sigma_{n-1}\}$ 的均衡负载量。

算法设计：对于给定的数据包序列 $\{\sigma_0,\sigma_1,\cdots,\sigma_{n-1}\}$，计算 m 个处理器的均衡负载量。

问题分析

使用动态规划函数 `g(i, k)` 来表示将从 `σ_i` 到 `σ_{n-1}` 的数据包序列划分为 `k` 段的最小最大负载量。

递推关系如下：

- **当 2<=k<n-i 时：**

$$g(i,k)=\min_{i\le j\le n-k}(\max\{f(i,j),g(j+1,k-1)\})$$

- **当 n-i<k<=m 时：**

$$g(i,k)=g(i,n-i)$$

边界条件：

- **当 (k = 1) 时：**

$$g(i,1)=f(i,n-1)$$

- **当 (k = n-i) 时：**

$$g(i,n-i)=\max_{i\le j\le n}f(j,j)$$

代码

```
#include <bits/stdc++.h>
using namespace std;

// 定义最大值数组和数据包权重数组
double g[500][500];
double t[500];
int n, m;

// 计算从数据包i到j的负载量
double f(int a, int b) {
    double ret = 0;
```

```
    for (int i = a; i <= b; i++) {
        ret += t[i] * t[i]; // 累加权重的平方
    }
    return sqrt(ret); // 返回负载量的平方根
}

// 动态规划求解最优负载均衡
double solve() {
    // 初始化 g[i][1] = f(i, n-1) 为边界条件
    for (int i = 0; i < n; i++) {
        g[i][1] = f(i, n - 1); // 单个区间的负载
    }

    // 动态规划，计算 g[i][k]，其中 k 从 2 到 m
    for (int k = 2; k <= m; k++) { // 从2段开始，因为1段是初始化的
        for (int i = 0; i <= n - k; i++) {
            double tmp = DBL_MAX; // 使用更精确的最大值表示
            for (int j = i; j <= n - k; j++) { // 寻找最优分割点
                double maxt = max(f(i, j), g[j + 1][k - 1]); // 计算当前分割的
                tmp = min(tmp, maxt); // 更新最小最大负载
            }
            g[i][k] = tmp; // 存储结果
        }
    }
    return g[0][m]; // 返回最优解
}

int main() {
    cin >> n >> m; // 读取数据包数量和处理器数量
    for (int i = 0; i < n; i++) {
        cin >> t[i]; // 读取每个数据包的权重
    }

    cout << fixed << solve() << endl; // 输出最优负载均衡结果
    return 0; // 程序结束
}
```

input

```
6 3
2 2 12 3 6 11
```

output

```
12.32
```

第四章

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。）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。

问题分析

对于将要举行的会议，可以先按照会议的开始时间进行一个排序，早开始的在前，后开始的在后；用一个vector数组来存储此时会场的是否空闲，之后从第一个会议开始，遍历已存在的所有会场，若有会场空闲（**通过该会场上一场会议的结束世界和该会议的开始时间**），则直接进入该会场，否则为该会议创建一个会场。

代码

```
#include<bits/stdc++.h>
using namespace std;

struct Meet{
    int begin;
    int end;
};
Meet meet[500];
bool Bool(Meet m1,Meet m2){  //判断函数，用以对会议按照开始时间排序
    return m1.begin<m2.begin;
}
vector<int> v;
int main(){
    ifstream ifs("input.txt");
    ofstream ofs("output.txt");
    int n;
    ifs >> n;
    for(int i=0;i<n;i++){
        ifs >> meet[i].begin>>meet[i].end;
    }
    sort(meet,meet+n,Bool);

    int sum = 1;
    v.push_back(0);

    for(int i=0;i<n;i++){
        int judge = 0;
```

```
        for(int j = 0;j < v.size();j++){
            if(meet[i].begin >= v[j]){    //存在空闲的会场
                v[j] = meet[i].end;       //使用该会场，并把该会场的空闲时间调整为
                break;
            }
            if(meet[i].begin < v[j] && j == v.size()-1){
                judge = 1;
            }
        }
        if(judge == 1){         //无空闲会场，则创建一个
            v.push_back(meet[i].end);
            sum ++;
        }
    }
    ofs << sum;
}
```

input

```
5
1 23
12 28
25 35
27 80
36 50
```

output

```
3
```

4-2 最优合并问题

问题描述

给定k个排好序的序列S1，S2，……，Sk，用2路合并算法将这k个序列合并成个序列。假设采用的2路合并算法合并2个长度分别为m和n的序列需要m+n-1次比较。
试设计一个算法确定合并这个序列的最优合并顺序，使所需的总比较次数最少。为了进行比较，还需要确定合并这个序列的最差合并顺序，使所需的总比较次数最多。对于给定的k个待合并序列，计算最多比较次数和最少比较次数合并方案。

问题分析

由于每次合并两个长度为m,n的数列，需要m+n-1次比较，因此先合并的数列在后面合并时还会被重复比较，于是应先将长度比较短数列进行合并，这样便可最小化比较次数；同样的，若

每次都将长度最大的两个数列进行合并，则会得到最多次的比较

代码

```
#include<bits/stdc++.h>
using namespace std;

vector<int> best;  //用以最少次数的计算
vector<int> worst; //用以最多次数的计算
bool B1(int a,int b){
    return a<b;
}
bool B2(int a,int b){
    return a>b;
}
int main(){
    ifstream ifs("input.txt");
    ofstream ofs("output.txt");
    int n,tmp;
    ifs >> n;
    if(n == 1){
        ofs<<0;
        return 0;
    }
    for(int i=0;i<n;i++){
        ifs >> tmp;
        best.push_back(tmp);
        worst.push_back(tmp);
    }
    int miny=0;
    int maxy=0;
    for(int i=0;i<n-1;i++){
        sort(best.begin(),best.end(),B1);       //先将best数组按照从小到大的顺序排
        tmp = best[0]+best[1];                  //取最小的两个合并
        miny += tmp - 1;
        sort(best.begin(),best.end(),B2);       //再按从大到小的顺序排列，便以清除
        best.pop_back();
        best.pop_back();
        best.push_back(tmp);
    }
    for(int i=0;i<n-1;i++){
        sort(worst.begin(),worst.end(),B2);     //先将worst数组按照从大到小的顺序
        tmp = worst[0]+worst[1];                //取最大的两个合并
        maxy += tmp - 1;
        sort(worst.begin(),worst.end(),B1);     //再按从小到大的顺序排列，便以清除
        worst.pop_back();
        worst.pop_back();
        worst.push_back(tmp);
    }
```

```
    ofs << miny<<" "<<maxy<<endl;
    ifs.close();
    ofs.close();
}
```

input

```
4
5 12 11 2
```

output

```
52 78
```

4-9 虚拟汽车加油问题

问题描述

一辆虚拟汽车加满油后可行驶nkm。旅途中有若干加油站。设计一个有效算法，指出应在哪些加油站停靠加油，使沿途加油次数最少。并证明算法能产生一个最优解。
对于给定的n和k个加油站位置，计算最少加油次数。

问题分析

要想要最小化加油次数，，则应该最大化汽油利用率，即`若此时剩余的油量足够走到下一加油站则在此加油站不加油`；另外如果距离下一加油站的距离大于满油里程，则无法到达目的地，输出`No Solution`

代码

```
#include<bits/stdc++.h>
using namespace std;

int rount[1000];
int main(){
    ifstream ifs("input.txt");
    ofstream ofs("output.txt");
    int n,m;
    ifs >> n >> m;
    for(int i=0;i<=n;i++){
        ifs >> rount[i];
    }
    int sum = 0;
    int curr = m;
    for(int i=0;i<=n;i++){
```

```
        if(curr >= rount[i]) curr -= rount[i];          //目前剩余的里程数足以到
        else if(curr < rount[i]){
            if(m < rount[i]){                           //下一加油站距离大于加满
                ofs<<"No Solution"<<endl;
                return 0;
            }
            curr = m - rount[i];                        //否则加油
            sum++;
        }
    }
    ofs << sum;
    ifs.close();
    ofs.close();
}
```

input

```
7 7
1 2 3 4 5 1 6 6
```

output

```
4
```