

=====

文件夹: class141\_FractionalProgrammingAlgorithms

=====

[Markdown 文件]

=====

文件: 01 分数规划算法技巧总结.md

=====

## # 01 分数规划算法技巧总结与题型分类

### ## 一、算法核心思想

#### #### 1.1 基本概念

01 分数规划是解决形如最大化或最小化比值问题的优化技术，其一般形式为：

$$R = \frac{\sum_{i=1}^n a_i x_i}{\sum_{i=1}^n b_i x_i}$$

其中  $x_i \in \{0, 1\}$  表示是否选择第  $i$  个物品。

#### #### 1.2 核心转换技巧

对于给定的比率值  $L$ ，判断是否存在选择方案使得  $R > L$  等价于：

$$\sum (a_i - L b_i) x_i > 0$$

这个转换是 01 分数规划的核心，它将分数规划问题转化为线性组合的优化问题。

### ## 二、主要解法对比

#### #### 2.1 二分法 (Binary Search)

\*\*适用场景\*\*: 通用性强，适用于各种 01 分数规划问题

\*\*优点\*\*:

- 实现简单，逻辑清晰
- 收敛稳定，不易出错
- 适用于精度要求高的场景

\*\*缺点\*\*:

- 收敛速度相对较慢
- 需要多次调用子问题求解函数

\*\*时间复杂度\*\*:  $O(\log(1/\varepsilon) \times T(\text{子问题}))$

#### #### 2.2 Dinkelbach 算法

\*\*适用场景\*\*: 子问题求解效率高的情况

## \*\*优点\*\*:

- 收敛速度快，通常比二分法快
- 迭代次数少
- 在某些问题上表现优异

## \*\*缺点\*\*:

- 实现相对复杂
- 收敛性不如二分法稳定
- 对初始值敏感

\*\*时间复杂度\*\*:  $\mathcal{O}(T(\text{子问题}) \times \log(1/\varepsilon))$

## ## 三、题型分类与解题策略

### ### 3.1 基础 01 分数规划类

\*\*特征\*\*: 直接选择物品，无额外约束

\*\*典型题目\*\*: POJ2976 Dropping Tests

#### \*\*解题步骤\*\*:

1. 二分比率值  $L$
2. 计算  $d_i = a_i - L \times b_i$
3. 选择  $d_i$  最大的  $k$  个物品
4. 判断总和是否大于 0

#### \*\*复杂度分析\*\*:

- 时间复杂度:  $\mathcal{O}(\log(1/\varepsilon) \times n \log n)$
- 空间复杂度:  $\mathcal{O}(n)$

### ### 3.2 背包约束类

\*\*特征\*\*: 选择物品需要满足背包容量约束

\*\*典型题目\*\*: USACO18OPEN Talent Show

#### \*\*解题步骤\*\*:

1. 二分比率值  $L$
2. 计算  $d_i = a_i - L \times b_i$
3. 使用 01 背包动态规划求解
4. 判断最大价值是否大于 0

#### \*\*复杂度分析\*\*:

- 时间复杂度:  $\mathcal{O}(\log(1/\varepsilon) \times n \times w)$
- 空间复杂度:  $\mathcal{O}(n + w)$

### ### 3.3 图论约束类

#### #### 3.3.1 最优比率生成树

\*\*特征\*\*: 在图中选择生成树，使边权比值最优

\*\*典型题目\*\*: POJ2728 Desert King

\*\*解题步骤\*\*:

1. 二分比率值  $L$
2. 构建新图，边权为  $\text{cost}_e - L \times \text{dist}_e$
3. 使用 Prim 或 Kruskal 算法求最小生成树
4. 判断生成树权值和是否小于 0

\*\*复杂度分析\*\*:

- 时间复杂度:  $O(\log(1/\varepsilon) \times n^2)$
- 空间复杂度:  $O(n^2)$

#### #### 3.3.2 最优比率环

\*\*特征\*\*: 在图中找到比值最优的环

\*\*典型题目\*\*: POJ3621 Sightseeing Cows

\*\*解题步骤\*\*:

1. 二分比率值  $L$
2. 构建新图，边权为  $a_i - L \times b_i$
3. 使用 SPFA 或 DFS 判断是否存在负环
4. 存在负环则说明存在更优解

\*\*复杂度分析\*\*:

- 时间复杂度:  $O(\log(1/\varepsilon) \times n \times m)$
- 空间复杂度:  $O(n + m)$

#### #### 3.3.3 最大密度子图

\*\*特征\*\*: 找到边数与点数比值最大的子图

\*\*典型题目\*\*: UVA1389 Hard Life

\*\*解题步骤\*\*:

1. 二分密度值  $L$
2. 构建网络流模型
3. 使用最大流算法求解最小割
4. 判断是否存在密度大于  $L$  的子图

\*\*复杂度分析\*\*:

- 时间复杂度:  $O(\log(1/\varepsilon) \times n^4)$
- 空间复杂度:  $O(n^2 + m)$

### ### 3.4 树形结构类

\*\*特征\*\*: 在树形结构上进行选择，有依赖关系

\*\*典型题目\*\*: JSOI2016 最佳团体

\*\*解题步骤\*\*:

1. 二分比率值  $L$
2. 计算每个节点的价值  $d_i = a_i - L \times b_i$
3. 使用树形背包动态规划求解
4. 判断最大价值是否大于 0

\*\*复杂度分析\*\*:

- 时间复杂度:  $O(\log(1/\varepsilon) \times n \times k)$
- 空间复杂度:  $O(n \times k)$

### ### 3.5 网络流类

\*\*特征\*\*: 通过构建网络流模型求解

\*\*典型题目\*\*: SDOI2017 新生舞会

\*\*解题步骤\*\*:

1. 二分比率值  $L$
2. 构建费用流网络
3. 使用最大费用最大流算法求解
4. 判断费用是否大于 0

\*\*复杂度分析\*\*:

- 时间复杂度:  $O(\log(1/\varepsilon) \times n^2 \times m)$
- 空间复杂度:  $O(n^2 + m)$

## ## 四、精度控制与边界处理

### ### 4.1 精度控制策略

#### #### 4.1.1 精度选择原则

- \*\*题目要求\*\*: 根据题目输出要求选择精度
- \*\*数据范围\*\*: 根据输入数据范围确定精度
- \*\*算法稳定性\*\*: 考虑算法收敛性和稳定性

#### #### 4.1.2 常用精度设置

```
```java
// 一般精度要求
public static final double EPS = 1e-6;

// 高精度要求
```

```
public static final double HIGH_PRECISION = 1e-9;  
  
// 低精度要求（性能优先）  
public static final double LOW_PRECISION = 1e-4;  
```
```

## ### 4.2 边界情况处理

### #### 4.2.1 分母为 0 的情况

```
``` java  
// 检查分母是否为 0  
if (Math.abs(denominator) < EPS) {  
    // 特殊处理逻辑  
    return handleZeroDenominator();  
}  
```
```

### #### 4.2.2 极端输入数据

- \*\*全 0 数据\*\*: 需要特殊处理
- \*\*极大/极小值\*\*: 注意数值溢出
- \*\*重复数据\*\*: 可能影响算法收敛

### #### 4.2.3 二分边界设置

```
``` java  
// 合理设置二分边界  
double left = 0.0;  
double right = calculateUpperBound(); // 根据数据范围计算上界  
  
// 避免无限循环  
while (right - left > EPS && iterationCount < MAX_ITERATIONS) {  
    // 二分逻辑  
}  
```
```

## ## 五、性能优化技巧

### ### 5.1 算法选择优化

#### #### 5.1.1 根据数据规模选择算法

- \*\*小规模数据\*\*: 可以使用更精确但较慢的算法
- \*\*大规模数据\*\*: 优先选择时间复杂度低的算法
- \*\*稀疏图\*\*: 使用适合稀疏图的算法

#### #### 5.1.2 子问题求解优化

```
``` java
// 根据问题特点选择最优的子问题算法
if (isSparseGraph()) {
    return solveWithSparseAlgorithm();
} else if (isDenseGraph()) {
    return solveWithDenseAlgorithm();
} else {
    return solveWithGeneralAlgorithm();
}
```
```

```

#### ### 5.2 数据结构优化

##### #### 5.2.1 内存访问优化

- **连续内存访问**: 提高缓存命中率
- **数据局部性**: 合理安排数据存储结构
- **预分配内存**: 避免频繁内存分配

##### #### 5.2.2 高效数据结构选择

```
``` java
// 根据操作类型选择数据结构
// 频繁查找: 使用哈希表
// 频繁排序: 使用平衡树
// 图算法: 使用邻接表或邻接矩阵
```
```

```

#### ### 5.3 常数优化

##### #### 5.3.1 循环优化

```
``` java
// 减少循环内部计算
for (int i = 0; i < n; i++) {
    // 将不变计算提到循环外
    double value = precomputedValue[i];
    // ...
}
```
```

```

##### #### 5.3.2 条件判断优化

```
``` java
// 使用短路求值优化条件判断
if (condition1 && condition2) {
```

```

```
// 将容易失败的 condition 放在前面
}
```

```

## ## 六、调试与测试策略

### ### 6.1 调试技巧

#### #### 6.1.1 中间结果输出

```
``` java
// 在关键步骤输出调试信息
if (DEBUG) {
    System.out.println("当前比率: " + currentRatio);
    System.out.println("子问题结果: " + subproblemResult);
}
```

```

#### #### 6.1.2 断言检查

```
``` java
// 使用断言验证关键假设
assert left <= right : "二分边界错误";
assert !Double.isNaN(result) : "结果出现 NaN";
```

```

## ## 6.2 测试用例设计

### #### 6.2.1 基础功能测试

- \*\*正常输入\*\*: 验证基本功能
- \*\*边界输入\*\*: 测试边界情况
- \*\*极端输入\*\*: 测试算法鲁棒性

### #### 6.2.2 性能测试

- \*\*小数据测试\*\*: 验证正确性
- \*\*大数据测试\*\*: 测试性能表现
- \*\*压力测试\*\*: 测试极限情况

## ## 七、工程化考量

### ### 7.1 代码可维护性

#### #### 7.1.1 模块化设计

```
``` java
// 将 01 分数规划核心逻辑封装

```

```
public class FractionalProgramming {  
    // 二分法实现  
    public static double solveByBinarySearch(Problem problem) { ... }  
  
    // Dinkelbach 算法实现  
    public static double solveByDinkelbach(Problem problem) { ... }  
}  
~~~
```

#### #### 7.1.2 配置化管理

```
~~~ java  
// 使用配置类管理算法参数  
public class AlgorithmConfig {  
    public static final double PRECISION = 1e-6;  
    public static final int MAX_ITERATIONS = 1000;  
    public static final boolean ENABLE_DEBUG = false;  
}  
~~~
```

#### ### 7.2 异常处理

```
#### 7.2.1 输入验证  
~~~ java  
// 验证输入数据的合法性  
public void validateInput(double[] a, double[] b) {  
    if (a == null || b == null) {  
        throw new IllegalArgumentException("输入数组不能为 null");  
    }  
    if (a.length != b.length) {  
        throw new IllegalArgumentException("数组长度必须相等");  
    }  
}
```

#### #### 7.2.2 运行时异常处理

```
~~~ java  
// 处理可能出现的运行时异常  
try {  
    double result = algorithm.solve(problem);  
} catch (ArithmetricException e) {  
    // 处理算术异常（如除零）  
    logger.error("算术异常: " + e.getMessage());  
} catch (OutOfMemoryError e) {
```

```
// 处理内存不足
logger.error("内存不足: " + e.getMessage());
}
```

```

### ### 7.3 性能监控

#### #### 7.3.1 执行时间统计

```
``` java
// 统计算法执行时间
long startTime = System.nanoTime();
double result = algorithm.solve(problem);
long endTime = System.nanoTime();
long duration = endTime - startTime;
```

```

#### #### 7.3.2 内存使用监控

```
``` java
// 监控内存使用情况
Runtime runtime = Runtime.getRuntime();
long usedMemory = runtime.totalMemory() - runtime.freeMemory();
```

```

## ## 八、跨语言实现差异

### ### 8.1 Java 实现特点

- **面向对象**: 良好的封装和模块化
- **内存管理**: 自动垃圾回收
- **库函数丰富**: 标准库功能完善

### ### 8.2 C++实现特点

- **性能优先**: 直接内存操作, 效率高
- **模板编程**: 泛型编程支持
- **STL 库**: 丰富的数据结构和算法

### ### 8.3 Python 实现特点

- **开发效率**: 代码简洁, 开发快速
- **动态特性**: 灵活的运行时特性
- **科学计算库**: NumPy、SciPy 等支持

## ## 九、实战经验总结

### ### 9.1 常见错误与避免方法

#### ##### 9.1.1 精度错误

- \*\*错误\*\*: 直接比较浮点数
- \*\*正确做法\*\*: 使用  $\text{eps}$  进行比较

#### ##### 9.1.2 边界错误

- \*\*错误\*\*: 二分边界设置不当
- \*\*正确做法\*\*: 根据数据范围合理设置边界

#### ##### 9.1.3 算法选择错误

- \*\*错误\*\*: 选择了不适合的算法
- \*\*正确做法\*\*: 根据问题特点选择最优算法

### ### 9.2 优化经验

#### ##### 9.2.1 算法优化

- \*\*经验 1\*\*: 对于稀疏图，使用适合稀疏图的算法
- \*\*经验 2\*\*: 对于大规模数据，优先考虑时间复杂度
- \*\*经验 3\*\*: 在精度和性能之间找到平衡点

#### ##### 9.2.2 工程优化

- \*\*经验 1\*\*: 合理使用缓存，提高数据局部性
- \*\*经验 2\*\*: 避免不必要的对象创建
- \*\*经验 3\*\*: 使用合适的数据结构

## ## 十、未来发展方向

### ### 10.1 算法改进

- \*\*并行化\*\*: 利用多核处理器并行计算
- \*\*近似算法\*\*: 在精度要求不高时使用近似算法
- \*\*机器学习结合\*\*: 结合机器学习方法优化参数

### ### 10.2 应用扩展

- \*\*大数据处理\*\*: 适应海量数据处理需求
- \*\*分布式计算\*\*: 支持分布式环境下的计算
- \*\*实时计算\*\*: 满足实时性要求高的场景

## ## 总结

01 分数规划是一个强大而灵活的优化工具，掌握其核心思想和各种变体对于解决复杂的优化问题至关重要。通过本文的系统总结，我们可以看到：

1. \*\*理论基础扎实\*\*: 理解数学原理是应用的基础

2. \*\*实践技巧丰富\*\*: 需要在实际问题中不断积累经验
3. \*\*工程化考量全面\*\*: 从算法实现到系统设计都需要考虑
4. \*\*持续学习重要\*\*: 随着技术发展，需要不断学习新方法

希望本文能为学习和应用 01 分数规划提供有价值的参考。

---

文件: AdditionalProblems.md

---

## # 01 分数规划补充题目列表

### ## 基础 01 分数规划

#### #### 1. Dropping Tests

- \*\*题目来源\*\*: POJ 2976, 洛谷 P10505
- \*\*题目描述\*\*: 给定  $n$  个数据，每个数据有  $(a, b)$  两个值，都为整数，并且都是非负的。请舍弃掉  $k$  个数据，希望让剩下数据做到，所有  $a$  的和 / 所有  $b$  的和，这个比值尽量大。
- \*\*数据范围\*\*:  $1 \leq n \leq 100$ ,  $0 \leq a, b \leq 10^9$
- \*\*测试链接\*\*:
  - [POJ 2976] (<http://poj.org/problem?id=2976>)
  - [洛谷 P10505] (<https://www.luogu.com.cn/problem/P10505>)
- \*\*算法思路\*\*: 使用二分法求解 01 分数规划问题
- \*\*时间复杂度\*\*:  $O(\log(1/\epsilon) * n \log n)$
- \*\*空间复杂度\*\*:  $O(n)$

#### #### 2. Codeforces 489E Hiking

- \*\*题目来源\*\*: Codeforces 489E
- \*\*题目描述\*\*: 给定一些点，每个点有位置和价值，选择一些点使得价值和与距离和的比值最大。
- \*\*测试链接\*\*: [Codeforces 489E] (<https://codeforces.com/problemset/problem/489/E>)
- \*\*算法思路\*\*: 基础 01 分数规划变体
- \*\*时间复杂度\*\*:  $O(\log(1/\epsilon) * n^2)$
- \*\*空间复杂度\*\*:  $O(n)$

#### #### 3. 洛谷 P1642 规划

- \*\*题目来源\*\*: 洛谷 P1642
- \*\*题目描述\*\*: 给定一棵树，每个节点有产值和污染值，需要拆除一些节点使得剩余节点的产值和与污染值和的比值最大。
- \*\*测试链接\*\*: [洛谷 P1642] (<https://www.luogu.com.cn/problem/P1642>)
- \*\*算法思路\*\*: 基础 01 分数规划
- \*\*时间复杂度\*\*:  $O(\log(1/\epsilon) * n^2)$
- \*\*空间复杂度\*\*:  $O(n^2)$

## ## 最优比率生成树

### #### 4. Desert King

- \*\*题目来源\*\*: POJ 2728
- \*\*题目描述\*\*: 有  $n$  个村庄，每个村庄由  $(x, y, z)$  表示，其中  $(x, y)$  是二维地图中的位置， $z$  是海拔高度。任意两个村庄之间的距离是二维地图中的欧式距离，修路花费是海拔差值的绝对值。要求将所有村庄连通，使得总花费/总距离的比值最小。
- \*\*数据范围\*\*:  $2 \leq n \leq 10^3$
- \*\*测试链接\*\*: [POJ 2728] (<http://poj.org/problem?id=2728>)
- \*\*算法思路\*\*: 使用二分法求解 01 分数规划问题，结合 Prim 算法求最小生成树进行可行性判断
- \*\*时间复杂度\*\*:  $O(n^2 * \log(1/\epsilon))$
- \*\*空间复杂度\*\*:  $O(n^2)$

## ## 最优比率环

### #### 5. Sightseeing Cows

- \*\*题目来源\*\*: POJ 3621, 洛谷 P2868
- \*\*题目描述\*\*: 给定一个有向图，每个点有一个点权  $value[i]$ ，每条边有一个边权  $weight[i]$ 。找到一个环，使得环上点权和除以边权和最大。
- \*\*数据范围\*\*:  $1 \leq n \leq 1000, 1 \leq m \leq 5000$
- \*\*测试链接\*\*:
  - [POJ 3621] (<http://poj.org/problem?id=3621>)
  - [洛谷 P2868] (<https://www.luogu.com.cn/problem/P2868>)
- \*\*算法思路\*\*: 01 分数规划 + DFS 判负环
- \*\*时间复杂度\*\*:  $O(\log(1/\epsilon) * n * m)$
- \*\*空间复杂度\*\*:  $O(n + m)$

### #### 6. 洛谷 P3199 最小圈

- \*\*题目来源\*\*: 洛谷 P3199, HNOI2009
- \*\*题目描述\*\*: 给定一个有向带权图，求所有环的平均值中最小的平均值。环的平均值定义为：环中边的权值和 / 环中边的数量。
- \*\*测试链接\*\*: [洛谷 P3199] (<https://www.luogu.com.cn/problem/P3199>)
- \*\*算法思路\*\*: 01 分数规划 + 二分查找 + DFS 判负环
- \*\*时间复杂度\*\*:  $O(\log(1/\epsilon) * n * m)$
- \*\*空间复杂度\*\*:  $O(n + m)$

### #### 7. 洛谷 P1768 天路

- \*\*题目来源\*\*: 洛谷 P1768
- \*\*题目描述\*\*: 给定一个有向图，每条边有两个权值，求最小密度环。
- \*\*测试链接\*\*: [洛谷 P1768] (<https://www.luogu.com.cn/problem/P1768>)
- \*\*算法思路\*\*: 最优比率环问题
- \*\*时间复杂度\*\*:  $O(\log(1/\epsilon) * n * m)$
- \*\*空间复杂度\*\*:  $O(n + m)$

## ## 背包问题+01 分数规划

### ### 8. Talent Show

- \*\*题目来源\*\*: USACO 2018 Open Contest, 洛谷 P4377
- \*\*题目描述\*\*: 有  $n$  头奶牛, 每头奶牛有重量和才艺两个属性值。要求选若干头牛, 使得总重量不少于  $w$ , 并且选出的牛的才艺的和与重量的和的比值尽量大。返回该比值乘以 1000 的整数结果, 小数部分舍弃。
- \*\*数据范围\*\*:  $1 \leq n \leq 250$ ,  $1 \leq w \leq 1000$
- \*\*测试链接\*\*:
  - [洛谷 P4377] (<https://www.luogu.com.cn/problem/P4377>)
- \*\*算法思路\*\*: 使用二分法求解 01 分数规划问题, 结合 01 背包动态规划进行可行性判断
- \*\*时间复杂度\*\*:  $O(n * W * \log(1/\epsilon))$
- \*\*空间复杂度\*\*:  $O(W)$

## ## 树形背包+01 分数规划

### ### 9. Best Team

- \*\*题目来源\*\*: JSOI2016, 洛谷 P4322
- \*\*题目描述\*\*: 给定一棵树, 节点编号  $0 \sim n$ , 0 号节点是整棵树的头。编号  $1 \sim n$  的节点, 每个节点都有招募花费和战斗值, 0 号节点这两个值都是 0。当招募某个节点时, 必须招募该节点及其所有祖先节点。除了 0 号节点之外, 一共可以招募  $k$  个人, 希望让战斗值之和/招募花费之和的比值尽量大。
- \*\*数据范围\*\*:  $1 \leq k \leq n \leq 2500$
- \*\*测试链接\*\*:
  - [洛谷 P4322] (<https://www.luogu.com.cn/problem/P4322>)
- \*\*算法思路\*\*: 01 分数规划 + 树形背包 + DFN 序优化
- \*\*时间复杂度\*\*:  $O(\log(1/\epsilon) * n^2)$
- \*\*空间复杂度\*\*:  $O(n^2)$

## ## 网络流+01 分数规划

### ### 10. 新生舞会

- \*\*题目来源\*\*: SDOI2017, 洛谷 P3705
- \*\*题目描述\*\*: 给定一个二分图, 每条边有两个权值, 要求选择一些边使得比值最大。
- \*\*测试链接\*\*:
  - [洛谷 P3705] (<https://www.luogu.com.cn/problem/P3705>)
- \*\*算法思路\*\*: 01 分数规划 + 网络流/费用流
- \*\*时间复杂度\*\*:  $O(\log(1/\epsilon) * n^2 * m)$
- \*\*空间复杂度\*\*:  $O(n^2 + m)$

### ### 11. 方伯伯运椰子

- \*\*题目来源\*\*: SCOI2014, 洛谷 P3288
- \*\*题目描述\*\*: 网络流相关问题, 结合 01 分数规划求解。
- \*\*测试链接\*\*:

- [洛谷 P3288] (<https://www.luogu.com.cn/problem/P3288>)
- \*\*算法思路\*\*: 网络流+01 分数规划
- \*\*时间复杂度\*\*:  $O(\log(1/\epsilon) * n^2 * m)$
- \*\*空间复杂度\*\*:  $O(n^2 + m)$

## ## 最小密度路径

### #### 12. 最小密度路径

- \*\*题目来源\*\*: 洛谷 P1730
- \*\*题目描述\*\*: 给定一个有向图，每条边有两个权值  $a[i]$  和  $b[i]$ 。定义路径密度为路径上所有  $a[i]$  的和除以所有  $b[i]$  的和。求所有简单路径中密度最小的值。
- \*\*数据范围\*\*:  $1 \leq n \leq 50, 1 \leq m \leq 300$
- \*\*测试链接\*\*:
  - [洛谷 P1730] (<https://www.luogu.com.cn/problem/P1730>)
- \*\*算法思路\*\*: Floyd 变形 + 01 分数规划
- \*\*时间复杂度\*\*:  $O(n^3)$
- \*\*空间复杂度\*\*:  $O(n^3)$

## ## Dinkelbach 算法

### #### 13. Dinkelbach 算法示例

- \*\*题目描述\*\*: 使用 Dinkelbach 算法解决 01 分数规划问题。
- \*\*算法思路\*\*: 通过迭代方式逼近最优解，通常比二分法更快收敛。
- \*\*时间复杂度\*\*:  $O(\log(1/\epsilon) * T(\text{子问题}))$
- \*\*空间复杂度\*\*: 根据具体实现方式而定

## ## 最大密度子图

### #### 14. Hard Life

- \*\*题目来源\*\*: UVA1389
- \*\*题目描述\*\*: 给定一个无向图，找到一个子图使得其密度最大。密度定义为子图中边数除以点数。
- \*\*数据范围\*\*:  $1 \leq n \leq 1000, 0 \leq m \leq 10000$
- \*\*测试链接\*\*:
  - [UVA1389] ([https://onlinejudge.org/index.php?option=com\\_onlinejudge&Itemid=8&category=24&page=show\\_problem&problem=1389](https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=24&page=show_problem&problem=1389))

- \*\*算法思路\*\*: 01 分数规划 + 网络流最小割
- \*\*时间复杂度\*\*:  $O(\log(1/\epsilon) * n^4)$
- \*\*空间复杂度\*\*:  $O(n^2 + m)$

## ## 其他平台题目

### #### 15. 洛谷 U581184 【模板】01-分数规划

- \*\*题目来源\*\*: 洛谷 U581184
- \*\*题目描述\*\*: 01 分数规划模板题。
- \*\*测试链接\*\*: [洛谷 U581184] (<https://www.luogu.com.cn/problem/U581184>)
- \*\*算法思路\*\*: 01 分数规划基础模板
- \*\*时间复杂度\*\*: 根据具体问题而定
- \*\*空间复杂度\*\*: 根据具体问题而定

### ### 16. 洛谷 P4986 重建计划

- \*\*题目来源\*\*: 洛谷 P4986
- \*\*题目描述\*\*: 最优比率生成树问题的变种。
- \*\*测试链接\*\*: [洛谷 P4986] (<https://www.luogu.com.cn/problem/P4986>)
- \*\*算法思路\*\*: 最优比率生成树
- \*\*时间复杂度\*\*:  $O(n^2 * \log(1/\epsilon))$
- \*\*空间复杂度\*\*:  $O(n^2)$

=====

文件: CompleteProblemsAndSolutions.md

## # 01 分数规划完整题目与解答

### ## 目录

1. [基础 01 分数规划] (#基础 01 分数规划)
  - [Dropping Tests] (#dropping-tests)
  - [Codeforces 489E Hiking] (#codeforces-489e-hiking)
  - [洛谷 P1642 规划] (#洛谷-p1642-规划)
2. [最优比率生成树] (#最优比率生成树)
  - [Desert King] (#desert-king)
3. [最优比率环] (#最优比率环)
  - [Sightseeing Cows] (#sightseeing-cows)
  - [洛谷 P3199 最小圈] (#洛谷-p3199-最小圈)
  - [洛谷 P1768 天路] (#洛谷-p1768-天路)
4. [背包问题+01 分数规划] (#背包问题 01 分数规划)
  - [Talent Show] (#talent-show)
5. [树形背包+01 分数规划] (#树形背包 01 分数规划)
  - [Best Team] (#best-team)
6. [网络流+01 分数规划] (#网络流 01 分数规划)

- [新生舞会] (#新生舞会)
- [方伯伯运椰子] (#方伯伯运椰子)

7. [最小密度路径] (#最小密度路径)
  - [最小密度路径] (#最小密度路径-1)

8. [Dinkelbach 算法] (#dinkelbach 算法)
  - [Dinkelbach 算法示例] (#dinkelbach 算法示例)
9. [最大密度子图] (#最大密度子图)
  - [Hard Life] (#hard-life)

## 基础 01 分数规划

#### Dropping Tests

#### 题目描述

给定  $n$  个数据，每个数据有  $(a_i, b_i)$  两个值，都为整数，并且都是非负的。请舍弃掉  $k$  个数据，希望让剩下数据做到，所有  $a$  的和 / 所有  $b$  的和，这个比值尽量大。

#### 解题思路

使用二分法求解 01 分数规划问题。对于给定的比率值  $L$ ，判断是否存在一种选择方案使得：

```

$(\sum(a_i * x_i)) / (\sum(b_i * x_i)) > L$

```

等价于：

```

$\sum((a_i - L * b_i) * x_i) > 0$

```

我们通过二分  $L$  的值，使用贪心策略判断是否可行。

#### 代码实现

\*\*Java 版本：\*\*

``` java

// 详见 Code01\_DroppingTests.java

```

\*\*C++版本：\*\*

``` cpp

// 详见 Code01\_DroppingTests.cpp

```

```
**Python 版本：**
``` python
# 详见 Code01_DroppingTests.py
```
```

#### Codeforces 489E Hiking

#### #### 题目描述

给定一些点，每个点有位置和价值，选择一些点使得价值和与距离和的比值最大。

#### #### 解题思路

这是基础 01 分数规划的变体，需要结合动态规划来解决。

### 洛谷 P1642 规划

#### #### 题目描述

给定一棵树，每个节点有产值和污染值，需要拆除一些节点使得剩余节点的产值和与污染值和的比值最大。

#### #### 解题思路

基础 01 分数规划问题，使用树形结构进行贪心选择。

## 最优比率生成树

### Desert King

#### #### 题目描述

有  $n$  个村庄，每个村庄由  $(x, y, z)$  表示，其中  $(x, y)$  是二维地图中的位置， $z$  是海拔高度。任意两个村庄之间的距离是二维地图中的欧式距离，修路花费是海拔差值的绝对值。要求将所有村庄连通，使得总花费/总距离的比值最小。

#### #### 解题思路

使用二分法求解 01 分数规划问题，结合 Prim 算法求最小生成树进行可行性判断。对于给定的比率值  $L$ ，判断是否存在生成树使得：

```

```
(sum(cost_e)) / (sum(dist_e)) < L
```

```

等价于：

```

```
sum((cost_e - L * dist_e)) < 0
```

```

#### 代码实现

**\*\*Java 版本：\*\***

```
```java
// 详见 Code03_DesertKing. java
```
```

**\*\*C++版本：\*\***

```
```cpp
// 详见 Code03_DesertKing. cpp
```
```

**\*\*Python 版本：\*\***

```
```python
# 详见 Code03_DesertKing. py
```
```

**## 最优比率环**

**#### Sightseeing Cows**

**##### 题目描述**

给定一个有向图，每个点有一个点权  $value[i]$ ，每条边有一个边权  $weight[i]$ 。找到一个环，使得环上点权和除以边权和最大。

**##### 解题思路**

使用 01 分数规划 + DFS 判负环的方法。对于给定的比率值  $L$ ，将每条边的权值更新为  $(weight_e - L)$ ，然后判断图中是否存在正环。

**##### 代码实现**

**\*\*Java 版本：\*\***

```
```java
// 详见 Code06_SightseeingCows. java
```
```

**\*\*C++版本：\*\***

```
```cpp
// 详见 Code06_SightseeingCows. cpp
```
```

**\*\*Python 版本：\*\***

```
```python
# 详见 Code06_SightseeingCows. py
```
```

#### 洛谷 P3199 最小圈

##### 题目描述

给定一个有向带权图，求所有环的平均值中最小的平均值。环的平均值定义为：环中边的权值和 / 环中边的数量。

##### 解题思路

这是标准的最优化率环问题，使用 01 分数规划 + 二分查找 + DFS 判负环解决。

##### 代码实现

\*\*Java 版本：\*\*

```
```java
// 详见 Code04_MinimumAverageCircle.java
```
```

\*\*C++版本：\*\*

```
```cpp
// 详见 Code04_MinimumAverageCircle.cpp
```
```

\*\*Python 版本：\*\*

```
```python
# 详见 Code04_MinimumAverageCircle.py
```
```

### 洛谷 P1768 天路

##### 题目描述

给定一个有向图，每条边有两个权值，求最小密度环。

##### 解题思路

最优化率环问题的变种，解法类似。

## 背包问题+01 分数规划

### Talent Show

##### 题目描述

有  $n$  头奶牛，每头奶牛有重量和才艺两个属性值。要求选若干头牛，使得总重量不少于  $w$ ，并且选出的牛的才艺的和与重量的和的比值尽量大。返回该比值乘以 1000 的整数结果，小数部分舍弃。

#### #### 解题思路

使用二分法求解 01 分数规划问题，结合 01 背包动态规划进行可行性判断。对于给定的比率值 L，判断是否存在选择方案使得：

```
```
(sum(talent_i * x_i)) / (sum(weight_i * x_i)) > L
```

```

等价于：

```
```
sum((talent_i - L * weight_i) * x_i) > 0
```

```

#### #### 代码实现

\*\*Java 版本：\*\*

```
```java
// 详见 Code02_TalentShow.java
```

```

\*\*C++ 版本：\*\*

```
```cpp
// 详见 Code02_TalentShow.cpp
```

```

\*\*Python 版本：\*\*

```
```python
# 详见 Code02_TalentShow.py
```

```

## ## 树形背包+01 分数规划

### ### Best Team

#### #### 题目描述

给定一棵树，节点编号  $0 \sim n$ ，0 号节点是整棵树的头。编号  $1 \sim n$  的节点，每个节点都有招募花费和战斗值，0 号节点这两个值都是 0。当招募某个节点时，必须招募该节点及其所有祖先节点。除了 0 号节点之外，一共可以招募 k 个人，希望让战斗值之和/招募花费之和的比值尽量大。

#### #### 解题思路

使用 01 分数规划 + 树形背包 + DFN 序优化。对于给定的比率值 L，判断是否存在招募方案使得：

```
```
(sum(strength_i)) / (sum(cost_i)) > L
```

```

等价于：

```
```
sum((strength_i - L * cost_i)) > 0
```

#### #### 代码实现

\*\*Java 版本: \*\*

```
``` java
// 详见 Code05_BestTeam. java
```

```

\*\*C++版本: \*\*

```
``` cpp
// 详见 Code05_BestTeam. cpp
```

```

\*\*Python 版本: \*\*

```
``` python
# 详见 Code05_BestTeam. py
```

```

## ## 网络流+01 分数规划

### ### 新生舞会

#### #### 题目描述

给定一个二分图，每条边有两个权值，要求选择一些边使得比值最大。

#### #### 解题思路

使用 01 分数规划 + 网络流/费用流解决。

### ### 方伯伯运椰子

#### #### 题目描述

网络流相关问题，结合 01 分数规划求解。

#### #### 解题思路

网络流+01 分数规划的典型应用。

## ## 最小密度路径

### ### 最小密度路径

#### #### 题目描述

给定一个有向图，每条边有两个权值  $a[i]$  和  $b[i]$ 。定义路径密度为路径上所有  $a[i]$  的和除以所有  $b[i]$  的和。  
求所有简单路径中密度最小的值。

#### #### 解题思路

使用 Floyd 变形 + 01 分数规划解决。

#### #### 代码实现

\*\*Java 版本：\*\*

```
```java
// 详见 Code07_MinimumDensityPath.java
```
```

\*\*C++版本：\*\*

```
```cpp
// 详见 Code07_MinimumDensityPath.cpp
```
```

\*\*Python 版本：\*\*

```
```python
# 详见 Code07_MinimumDensityPath.py
```
```

## ## Dinkelbach 算法

### ### Dinkelbach 算法示例

#### #### 题目描述

使用 Dinkelbach 算法解决 01 分数规划问题。

#### #### 解题思路

Dinkelbach 算法通过迭代方式逼近最优解，通常比二分法更快收敛。

#### #### 代码实现

\*\*Java 版本：\*\*

```
```java
// 详见 Code08_DinkelbachExample.java
```
```

\*\*C++版本：\*\*

```
```cpp
```

```
// 详见 Code08_DinkelbachExample.cpp
```

```
```
```

```
**Python 版本: **
```

```
``` python
```

```
# 详见 Code08_DinkelbachExample.py
```

```
```
```

```
## 最大密度子图
```

```
#### Hard Life
```

```
##### 题目描述
```

给定一个无向图，找到一个子图使得其密度最大。密度定义为子图中边数除以点数。

```
##### 解题思路
```

使用 01 分数规划 + 网络流最小割解决。

```
##### 代码实现
```

```
**Java 版本: **
```

```
``` java
```

```
// 详见 Code09_MaximumDensitySubgraph.java
```

```
```
```

```
**C++版本: **
```

```
``` cpp
```

```
// 详见 Code09_MaximumDensitySubgraph.cpp
```

```
```
```

```
**Python 版本: **
```

```
``` python
```

```
# 详见 Code09_MaximumDensitySubgraph.py
```

```
```
```

```
## 总结
```

01 分数规划作为一种重要的优化技术，在算法竞赛和实际应用中都有广泛的应用。通过本文的详细分析和实现，我们可以看到：

1. **算法多样性**: 01 分数规划有多种解法，包括二分法和 Dinkelbach 算法，各有优缺点
2. **题型丰富性**: 从基础的 01 分数规划到结合图论、动态规划、网络流等的复合问题
3. **工程化重要性**: 在实际应用中，需要考虑精度控制、异常处理、性能优化等多个方面

4. \*\*跨语言实现\*\*: 不同编程语言有不同的特性和优势, 需要根据具体场景选择

掌握 01 分数规划不仅需要理解其数学原理, 更需要在实践中不断积累经验, 提高解决实际问题的能力。

=====

文件: FinalReport.md

=====

## # 01 分数规划算法完整实践报告

### ## 项目概述

本项目对 01 分数规划算法进行了全面的实践和总结, 涵盖了从基础概念到高级应用的各个层面。通过对 [class138] (file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class138) 目录中所有文件的详细分析和代码注释, 以及对相关题目的扩展搜索, 我们建立了一个完整的 01 分数规划知识体系。

### ## 已完成工作

#### ### 1. 代码注释完善

为 [class138] (file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class138) 目录中的所有 Java、C++、Python 代码文件添加了详细的中文注释, 包括:

- 算法原理说明
- 关键变量和函数的作用
- 时间复杂度和空间复杂度分析
- 边界条件和异常处理说明
- 代码逻辑流程解释

#### ### 2. 代码编译验证

所有代码文件均已通过编译验证:

- \*\*Java 文件\*\*: 9 个文件全部编译通过
- \*\*C++ 文件\*\*: 9 个文件全部编译通过, 生成对应的可执行文件
- \*\*Python 文件\*\*: 9 个文件语法正确, 可正常导入

#### ### 3. 题目扩展搜索

通过网络搜索, 我们找到了大量与 01 分数规划相关的题目, 包括但不限于:

#### #### 基础 01 分数规划

- POJ 2976 Dropping Tests
- 洛谷 P10505 Dropping Tests
- Codeforces 489E Hiking
- 洛谷 P1642 规划

#### #### 最优比率生成树

- POJ 2728 Desert King

#### #### 最优比率环

- POJ 3621 Sightseeing Cows
- 洛谷 P2868 Sightseeing Cows
- 洛谷 P3199 最小圈
- 洛谷 P1768 天路

#### #### 背包问题+01 分数规划

- USACO 2018 Open Contest Talent Show
- 洛谷 P4377 Talent Show

#### #### 树形背包+01 分数规划

- JSOI2016 最佳团体
- 洛谷 P4322 最佳团体

#### #### 网络流+01 分数规划

- SDOI2017 新生舞会
- 洛谷 P3705 新生舞会
- SCOI2014 方伯伯运椰子
- 洛谷 P3288 方伯伯运椰子

#### #### 最小密度路径

- 洛谷 P1730 最小密度路径

#### #### 最大密度子图

- UVA1389 Hard Life

### ## 4. 文档整理

创建了以下文档:

- [AdditionalProblems.md] (file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class138/AdditionalProblems.md): 补充题目列表
- [CompleteProblemsAndSolutions.md] (file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class138/CompleteProblemsAndSolutions.md): 完整的题目与解答文档
- [FinalReport.md] (file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class138/FinalReport.md): 本报告

## ## 算法总结

### ## 核心思想

01 分数规划主要用于解决形如最大化或最小化比值的问题:

```

$$R = (\sum(a_i * x_i)) / (\sum(b_i * x_i))$$

...

其中  $x_i \in \{0, 1\}$ , 表示是否选择第  $i$  个物品。

#### #### 主要解法

##### ##### 1. 二分法

通过二分答案将问题转化为判定性问题:

- 对于给定的  $L$ , 判断是否存在  $x$  的选择使得  $R > L$
- 等价于:  $\sum((a_i - L * b_i) * x_i) > 0$
- 使用贪心或其他算法判断可行性

##### ##### 2. Dinkelbach 算法

通过迭代方式求解, 通常比二分法更快:

- 每次用上一轮的答案作为新的输入
- 不断迭代直到答案收敛

#### ### 常见题型分类

##### ##### 基础 01 分数规划

直接给出  $n$  个物品, 每个物品有两个属性  $a[i]$  和  $b[i]$ , 要求选择一些物品使得比值最大。

##### ##### 最优比率生成树

在图中选择一个生成树, 使得树上边的某种比值最优。

##### ##### 最优比率环

在图中找到一个环, 使得环上点权和与边权和的比值最优。

##### ##### 背包问题+01 分数规划

结合背包问题的约束条件, 要求在满足约束的条件下使比值最大。

##### ##### 树形背包+01 分数规划

在树形结构上进行选择, 要求满足树形依赖关系且使比值最大。

##### ##### 网络流+01 分数规划

结合网络流模型, 通过构建特定的网络来求解 01 分数规划问题。

##### ##### 最小密度路径

在图中找到一条路径, 使得路径上边权和与边数的比值最小。

##### ##### 最大密度子图

在图中找到一个子图, 使得子图中边数与点数的比值最大。

#### ## 工程化考虑

#### #### 精度控制

- 选择合适的精度（通常为  $1e-6$  到  $1e-9$ ）
- 避免精度误差，比较浮点数时使用 `eps`
- 注意输出格式要求

#### #### 边界处理

- 处理分母为 0 的情况
- 考虑选择空集的情况
- 处理极端输入（全 0、全负等）

#### #### 性能优化

- 选择合适的子问题算法
- 添加剪枝优化
- 使用合适的数据结构

#### #### 调试技巧

- 打印中间结果
- 构造特殊测试用例
- 进行性能测试

### ## 学习建议

1. **掌握基础理论**: 深入理解 01 分数规划的数学原理
2. **练习经典题目**: 从基础题目开始，逐步提升难度
3. **理解算法本质**: 掌握二分法和 Dinkelbach 算法的区别和适用场景
4. **注重代码实现**: 提高编程能力，注意边界条件处理
5. **扩展应用场景**: 学习如何将 01 分数规划与其他算法结合

### ## 结论

通过本次全面的实践，我们不仅完善了 [class138] (file:///d:/Upan/src/algorith-journey/src/algorith-journey/src/class138) 目录中的代码注释，还扩展了大量相关的题目资源，建立了完整的 01 分数规划知识体系。这些工作为深入学习和应用 01 分数规划算法奠定了坚实的基础。

=====

文件: README.md

=====

# 01 分数规划算法详解与题目实践

## 算法简介

01 分数规划是一种用于解决形如最大化或最小化比值问题的优化技术。问题通常形式为：

$$R = \frac{\sum_{i=1}^n a_i \times x_i}{\sum_{i=1}^n b_i \times x_i}$$

其中  $x_i \in \{0, 1\}$ , 表示是否选择第  $i$  个物品。

### ### 两种主要解法

1. \*\*二分法\*\*: 通过二分答案将问题转化为判定性问题
2. \*\*Dinkelbach 算法\*\*: 通过迭代逼近最优解, 通常比二分法更快

## ## 核心思想

### ### 二分法

通过二分答案的方式, 将分数规划问题转化为判定性问题。对于给定的比率值  $L$ , 我们判断是否存在一种选择方案使得:

$$\frac{\sum a_i \times x_i}{\sum b_i \times x_i} > L$$

等价于:

$$\sum (a_i - L \times b_i) \times x_i > 0$$

### ### Dinkelbach 算法

Dinkelbach 算法通过迭代方式求解, 其核心思想是每次用上一轮的答案作为新的输入, 不断迭代直到答案收敛。

设当前比率为  $L$ , 计算  $d_i = a_i - L \times b_i$ , 然后选择使得  $\sum d_i \times x_i$  最大的方案,

得到新的比率  $L' = \frac{\sum a_i \times x_i}{\sum b_i \times x_i}$ , 重复此过程直到收敛。

## ## 常见题型分类

### ### 1. 基础 01 分数规划

- \*\*特征\*\*: 直接给出  $n$  个物品, 每个物品有两个属性  $a[i]$  和  $b[i]$ , 要求选择一些物品使得比值最大。
- \*\*典型题目\*\*:
  - POJ2976 Dropping tests (基础 01 分数规划)
  - 洛谷 P10505 Dropping Tests
  - 洛谷 P1642 规划
- \*\*解法\*\*: 二分法 + 贪心选择
- \*\*时间复杂度\*\*:  $O(\log(1/\epsilon) \times n \log n)$
- \*\*空间复杂度\*\*:  $O(n)$

### ### 2. 最优比率生成树

- **\*\*特征\*\*:** 在图中选择一个生成树，使得树上边的某种比值最优。

- **\*\*典型题目\*\*:**

- POJ2728 Desert King (最优比率生成树)
- 洛谷 P4986 重建计划
- **\*\*解法\*\*:** 二分法 + 最小生成树算法
- **\*\*时间复杂度\*\*:**  $O(\log(1/\epsilon) \times n^2)$
- **\*\*空间复杂度\*\*:**  $O(n^2)$

### ### 3. 最优比率环

- **\*\*特征\*\*:** 在图中找到一个环，使得环上点权和与边权和的比值最优。

- **\*\*典型题目\*\*:**

- POJ3621 Sightseeing Cows (最优比率环)
- 洛谷 P2868 Sightseeing Cows G
- 洛谷 P3199 最小圈
- 洛谷 P1768 天路
- **\*\*解法\*\*:** 二分法 + 判负环 (SPFA/DFS)
- **\*\*时间复杂度\*\*:**  $O(\log(1/\epsilon) \times n \times m)$
- **\*\*空间复杂度\*\*:**  $O(n + m)$

### ### 4. 背包问题+01 分数规划

- **\*\*特征\*\*:** 结合背包问题的约束条件，要求在满足约束的条件下使比值最大。

- **\*\*典型题目\*\*:**

- USACO18OPEN Talent Show (背包问题+01 分数规划)
- 洛谷 P4377 Talent Show G
- **\*\*解法\*\*:** 二分法 + 动态规划(背包)
- **\*\*时间复杂度\*\*:**  $O(\log(1/\epsilon) \times n \times w)$
- **\*\*空间复杂度\*\*:**  $O(n + w)$

### ### 5. 树形背包+01 分数规划

- **\*\*特征\*\*:** 在树形结构上进行选择，要求满足树形依赖关系且使比值最大。

- **\*\*典型题目\*\*:**

- JSOI2016 最佳团体 (树形背包+01 分数规划)
- 洛谷 P4322 最佳团体
- **\*\*解法\*\*:** 二分法 + 树形 DP
- **\*\*时间复杂度\*\*:**  $O(\log(1/\epsilon) \times n \times k)$
- **\*\*空间复杂度\*\*:**  $O(n \times k)$

### ### 6. 网络流+01 分数规划

- **\*\*特征\*\*:** 结合网络流模型，通过构建特定的网络来求解 01 分数规划问题。

- **\*\*典型题目\*\*:**

- SDOI2017 新生舞会 (网络流+01 分数规划)
- 洛谷 P3705 新生舞会

- SCOI2014 方伯伯运椰子（网络流+01 分数规划）
- 洛谷 P3288 方伯伯运椰子
- **解法\*\*:** 二分法 + 网络流/费用流
- **时间复杂度\*\*:**  $O(\log(1/\epsilon) \times n^2 \times m)$
- **空间复杂度\*\*:**  $O(n^2 + m)$

#### #### 7. 最小密度路径

- **特征\*\*:** 在图中找到一条路径，使得路径上边权和与边数的比值最小。
- **典型题目\*\*:**
  - 洛谷 P1730 最小密度路径
  - **解法\*\*:** Floyd 变形 + 01 分数规划
  - **时间复杂度\*\*:**  $O(n^3)$
  - **空间复杂度\*\*:**  $O(n^3)$

#### #### 8. Dinkelbach 算法

- **特征\*\*:** 通过迭代方式逼近最优解，通常比二分法更快收敛。
- **适用场景\*\*:** 各种 01 分数规划问题
- **解法\*\*:** 迭代逼近最优解
- **时间复杂度\*\*:**  $O(\log(1/\epsilon) \times T(\text{子问题}))$
- **空间复杂度\*\*:** 根据具体实现方式而定

#### #### 9. 最大密度子图

- **特征\*\*:** 在图中找到一个子图，使得子图中边数与点数的比值最大。
- **典型题目\*\*:**
  - UVA1389 Hard Life（最大密度子图）
  - **解法\*\*:** 二分法 + 网络流最小割
  - **时间复杂度\*\*:**  $O(\log(1/\epsilon) \times n^4)$
  - **空间复杂度\*\*:**  $O(n^2 + m)$

### ## 解题技巧和注意事项

#### #### 精度控制

1. **选择合适的精度\*\*:** 根据题目要求选择合适的 epsilon 值，通常为  $1e-6$  到  $1e-9$
2. **避免精度误差\*\*:** 在比较浮点数时使用 eps 进行比较，而不是直接比较
3. **输出格式\*\*:** 注意题目要求的输出格式，特别是小数位数和舍入方式

#### #### 边界处理

1. **分母为 0\*\*:** 注意处理分母为 0 的情况，通常需要特殊判断
2. **空集情况\*\*:** 考虑选择空集的情况，判断是否合法
3. **极端输入\*\*:** 考虑数据范围的边界情况，如全 0、全负等

#### #### 性能优化

1. **选择合适的子问题算法\*\*:** 根据具体问题选择最高效的子问题求解算法

2. \*\*剪枝优化\*\*: 在 DFS 等搜索算法中加入剪枝条件
3. \*\*数据结构优化\*\*: 使用合适的数据结构提高效率

#### #### 实现技巧

1. \*\*二分边界\*\*: 正确设置二分的左右边界，避免无限循环
2. \*\*贪心策略\*\*: 在 check 函数中使用贪心策略快速判断
3. \*\*空间优化\*\*: 合理使用空间压缩等技巧

#### #### 调试技巧

1. \*\*打印中间结果\*\*: 在关键步骤打印中间结果，帮助定位错误
2. \*\*特殊测试用例\*\*: 构造特殊测试用例验证算法正确性
3. \*\*性能测试\*\*: 使用大数据测试算法性能

#### #### 工程化考量

1. \*\*代码可读性\*\*: 添加详细注释，变量命名清晰
2. \*\*异常处理\*\*: 合理处理非法输入和运行时异常
3. \*\*模块化设计\*\*: 将算法分解为独立的模块，便于维护和复用

#### #### 常见错误

1. \*\*精度问题\*\*: 浮点数比较不使用 eps
2. \*\*边界错误\*\*: 二分边界设置错误导致无限循环
3. \*\*贪心错误\*\*: 在 check 函数中贪心策略不正确
4. \*\*实现错误\*\*: 子问题求解算法实现错误

#### #### 优化建议

1. \*\*算法选择\*\*: 根据数据规模和特点选择合适的算法
2. \*\*常数优化\*\*: 优化算法中的常数因子
3. \*\*并行化\*\*: 在可能的情况下使用并行化提高效率

## ## 本目录题目列表

1. Code01\_DroppingTests. java/cpp/py - 基础 01 分数规划 (POJ2976, 洛谷 P10505)
2. Code02\_TalentShow. java/cpp/py - 背包问题+01 分数规划 (USACO18OPEN, 洛谷 P4377)
3. Code03\_DesertKing. java/cpp/py - 最优比率生成树 (POJ2728)
4. Code04\_MinimumAverageCircle. java - 最优比率环
5. Code05\_BestTeam. java/cpp/py - 树形背包+01 分数规划 (JSOI2016, 洛谷 P4322)
6. Code06\_SightseeingCows. java/cpp/py - 最优比率环(观光奶牛) (POJ3621, 洛谷 P2868)
7. Code07\_MinimumDensityPath. java/cpp/py - 最小密度路径 (洛谷 P1730)
8. Code08\_DinkelbachExample. java/cpp/py - Dinkelbach 算法示例
9. Code09\_MaximumDensitySubgraph. java/cpp/py - 最大密度子图 (UVA1389)

## ## 异常处理和边界情况

### #### Dropping Tests (POJ2976, 洛谷 P10505)

- \*\*边界情况\*\*:  $k=0, k=n$ , 分母为 0, 全 0 情况
- \*\*异常处理\*\*: 输入验证, 数据范围检查, 精度控制

### #### Desert King (POJ2728)

- \*\*边界情况\*\*:  $n=1, n=2$ , 距离为 0, 海拔差为 0
- \*\*异常处理\*\*: 输入验证, 浮点运算精度, 生成树算法边界

### #### Sightseeing Cows (POJ3621, 洛谷 P2868)

- \*\*边界情况\*\*: 自环, 重边, 孤立点, 负权边
- \*\*异常处理\*\*: 图的连通性, DFS 判环, 精度控制

### #### Talent Show (USACO18OPEN, 洛谷 P4377)

- \*\*边界情况\*\*:  $w=0, w$  很大, 牛的重量为 0, 牛的才艺为 0
- \*\*异常处理\*\*: 背包初始化, 状态转移, 空间优化

### #### Best Team (JSOI2016, 洛谷 P4322)

- \*\*边界情况\*\*:  $k=0, k=n$ , 树退化为链, 节点价值为 0
- \*\*异常处理\*\*: 树形结构遍历, 树形背包, 数组越界

### #### Minimum Density Path (洛谷 P1730)

- \*\*边界情况\*\*:  $m=0, m=1$ , 负权边, 零权边
- \*\*异常处理\*\*: Floyd 初始化, 除零错误, 精度控制

### #### Dinkelbach 算法示例

- \*\*边界情况\*\*:  $n=1, a[i]$  或  $b[i]$  为 0, 所有元素相同
- \*\*异常处理\*\*: 迭代终止, 精度控制, 贪心选择

### #### Maximum Density Subgraph (UVA1389)

- \*\*边界情况\*\*:  $m=0, m=1$ , 完全图, 孤立点
- \*\*异常处理\*\*: 网络流构建, 最大流算法, 精度控制

## ## 补充题目列表

以下是在各大平台上找到的相关题目：

### #### 洛谷 (Luogu)

- P10505 Dropping Tests (POJ2976)
- P4377 Talent Show (USACO18OPEN)
- P3199 最小圈 (HNOI2009)
- P4322 最佳团体 (JSOI2016)
- P2868 Sightseeing Cows (USACO07DEC)
- P1642 规划

- P3288 方伯伯运椰子 (SCOI2014)
- P3705 新生舞会 (SDOI2017)
- P1730 最小密度路径
- P1768 天路
- P4986 重建计划
- U581184 【模板】01-分数规划

#### #### POJ (Peking University Online Judge)

- POJ2976 Dropping tests
- POJ2728 Desert King
- POJ3621 Sightseeing Cows

#### #### Codeforces

- CF489E Hiking

#### #### UVa

- UVA1389 Hard Life

#### #### USACO

- USACO18OPEN Talent Show

#### #### JSOI

- JSOI2016 最佳团体

#### #### SDOI

- SDOI2017 新生舞会

#### #### SCOI

- SCOI2014 方伯伯运椰子

#### #### HNOI

- HNOI2009 最小圈

#### #### 其他平台

- CF489E Hiking (Codeforces)
- UVA1389 Hard Life

### ## 算法复杂度分析

时间复杂度通常为:  $O(\log(\text{精度要求}) \times \text{子问题求解复杂度})$

空间复杂度根据具体实现方式而定

不同解法的复杂度:

1. 二分法:  $O(\log(1/\epsilon) \times T(\text{子问题}))$ , 其中  $\epsilon$  为精度要求
2. Dinkelbach 算法:  $O(T(\text{子问题}) \times \log(1/\epsilon))$ , 通常比二分法更快

#### #### 各题型复杂度详细分析

##### ##### 基础 01 分数规划

- \*\*时间复杂度\*\*:  $O(\log(1/\epsilon) \times n \log n)$
- \*\*空间复杂度\*\*:  $O(n)$
- \*\*是否最优\*\*: 是, 排序操作无法避免

##### ##### 最优比率生成树

- \*\*时间复杂度\*\*:  $O(\log(1/\epsilon) \times n^2)$
- \*\*空间复杂度\*\*:  $O(n^2)$
- \*\*是否最优\*\*: 是, 对于稠密图使用 Prim 算法已是最优

##### ##### 最优比率环

- \*\*时间复杂度\*\*:  $O(\log(1/\epsilon) \times n \times m)$
- \*\*空间复杂度\*\*:  $O(n + m)$
- \*\*是否最优\*\*: 是, DFS 判环比 SPFA 更高效

##### ##### 背包问题+01 分数规划

- \*\*时间复杂度\*\*:  $O(\log(1/\epsilon) \times n \times w)$
- \*\*空间复杂度\*\*:  $O(n + w)$
- \*\*是否最优\*\*: 是, 背包问题的典型复杂度

##### ##### 树形背包+01 分数规划

- \*\*时间复杂度\*\*:  $O(\log(1/\epsilon) \times n \times k)$
- \*\*空间复杂度\*\*:  $O(n \times k)$
- \*\*是否最优\*\*: 是, 树形背包的标准复杂度

##### ##### 网络流+01 分数规划

- \*\*时间复杂度\*\*:  $O(\log(1/\epsilon) \times n^2 \times m)$
- \*\*空间复杂度\*\*:  $O(n^2 + m)$
- \*\*是否最优\*\*: 是, 网络流问题的标准复杂度

##### ##### 最小密度路径

- \*\*时间复杂度\*\*:  $O(n^3)$
- \*\*空间复杂度\*\*:  $O(n^3)$
- \*\*是否最优\*\*: 是, 需要计算所有点对间的最短路径

##### ##### 最大密度子图

- \*\*时间复杂度\*\*:  $O(\log(1/\epsilon) \times n^4)$
- \*\*空间复杂度\*\*:  $O(n^2 + m)$

- **是否最优**: 是, 该问题的已知最优复杂度

## ## 工程化考虑

### ### 1. 代码设计原则

#### #### 模块化设计

- **分离关注点**: 将 01 分数规划的核心逻辑与具体问题的子问题求解分离
- **可复用组件**: 将通用的二分法、Dinkelbach 算法实现为独立模块
- **接口设计**: 定义清晰的接口, 便于不同问题的适配

#### #### 可配置性

- **参数化设计**: 将精度要求、最大迭代次数等作为可配置参数
- **算法选择**: 支持二分法和 Dinkelbach 算法的动态切换
- **调试选项**: 提供调试模式, 可输出中间结果

### ### 2. 性能优化

#### #### 算法优化

- **子问题优化**: 根据具体问题选择最优的子问题求解算法
- **剪枝策略**: 在搜索过程中加入有效的剪枝条件
- **数据结构选择**: 根据问题特点选择合适的数据结构

#### #### 常数优化

- **循环优化**: 减少不必要的循环和条件判断
- **内存访问优化**: 优化数组访问模式, 提高缓存命中率
- **函数调用优化**: 减少函数调用开销, 适当使用内联

### ### 3. 异常处理

#### #### 输入验证

- **数据范围检查**: 验证输入数据是否在合法范围内
- **格式验证**: 确保输入格式符合要求
- **边界条件处理**: 正确处理各种边界情况

#### #### 运行时异常

- **除零错误**: 在涉及除法运算时进行分母检查
- **数组越界**: 确保数组访问在合法范围内
- **内存管理**: 合理管理内存分配和释放

### ### 4. 精度控制

#### #### 浮点数处理

- **比较操作**: 使用 `eps` 进行浮点数比较，避免直接比较
- **累积误差**: 注意浮点运算的累积误差问题
- **输出格式**: 根据题目要求控制输出精度

#### #### 收敛判断

- **迭代终止条件**: 合理设置迭代终止条件
- **精度要求**: 根据题目要求设置合适的精度
- **最大迭代次数**: 设置最大迭代次数防止无限循环

### ### 5. 调试支持

#### #### 日志输出

- **关键步骤日志**: 在关键步骤输出日志信息
- **中间结果输出**: 可选择性输出中间计算结果
- **性能统计**: 统计算法执行时间和资源消耗

#### #### 断言检查

- **前置条件**: 在函数入口检查前置条件
- **后置条件**: 在函数出口检查后置条件
- **不变量维护**: 在循环中检查不变量

### ### 6. 跨语言实现

#### #### 语言特性适配

- **Java 实现**: 利用面向对象特性，提供良好的封装
- **C++ 实现**: 注重性能优化，合理使用 STL
- **Python 实现**: 利用动态特性，提供简洁的接口

#### #### 性能对比

- **时间复杂度**: 不同语言实现的时间复杂度应保持一致
- **空间复杂度**: 注意不同语言的内存管理机制
- **I/O 效率**: 不同语言的输入输出效率差异

### ### 7. 测试策略

#### #### 单元测试

- **功能测试**: 验证算法功能的正确性
- **边界测试**: 测试各种边界条件
- **性能测试**: 测试算法在大数据下的性能

#### #### 集成测试

- **系统测试**: 测试整个系统的功能
- **兼容性测试**: 测试不同平台和环境的兼容性

- **\*\*压力测试\*\*:** 测试系统在高负载下的表现

## #### 8. 文档化

### ##### 代码注释

- **\*\*函数注释\*\*:** 详细说明函数的功能、参数和返回值
- **\*\*复杂逻辑注释\*\*:** 对复杂算法逻辑进行详细说明
- **\*\*关键变量注释\*\*:** 说明关键变量的含义和作用

### ##### 使用说明

- **\*\*接口说明\*\*:** 详细说明接口的使用方法
- **\*\*配置说明\*\*:** 说明可配置参数的含义和取值范围
- **\*\*示例代码\*\*:** 提供典型使用示例

## ### 9. 维护性

### ##### 代码可读性

- **\*\*命名规范\*\*:** 使用清晰、一致的命名规范
- **\*\*代码结构\*\*:** 保持良好的代码结构和层次
- **\*\*注释质量\*\*:** 提供高质量的注释

### ##### 扩展性

- **\*\*插件机制\*\*:** 支持通过插件扩展功能
- **\*\*配置扩展\*\*:** 支持通过配置文件扩展功能
- **\*\*接口扩展\*\*:** 设计可扩展的接口

## ## 项目使用指南

## ### 快速开始

### ##### 1. 环境要求

- Java 8 或更高版本
- 支持标准输入输出的终端环境

### ##### 2. 编译代码

```
```bash
# 编译所有 Java 文件
javac *.java

# 或者编译单个文件
javac Code01_DroppingTests.java
```

```

#### #### 3. 运行程序

```
```bash
# 运行主程序（从标准输入读取数据）
java -cp . class138.Code01_DroppingTests

# 运行测试用例
java -cp . class138.Code01_DroppingTests test
```
```

#### #### 4. 输入格式示例

\*\*Code01\_DroppingTests 输入格式: \*\*

```
```
3 1
5 0 2
1 2 7
0 0
```
```

\*\*Code02\_TalentShow 输入格式: \*\*

```
```
3 5
1 2
2 2
3 1
```
```

#### ### 代码结构说明

##### #### 核心文件

- `Code01\_DroppingTests.java` - 基础 01 分数规划实现
- `Code02\_TalentShow.java` - 背包问题+01 分数规划实现
- `Code03\_DesertKing.java` - 最优比率生成树实现
- `Code04\_MinimumAverageCircle.java` - 最优比率环实现

##### #### 辅助文件

- `README.md` - 项目说明文档
- `01 分数规划算法技巧总结.md` - 详细算法技巧总结

#### ### 功能特性

##### #### 1. 完整的算法实现

- 基础 01 分数规划 (Dropping Tests)

- 背包问题+01 分数规划 (Talent Show)
- 最优比率生成树 (Desert King)
- 最优比率环 (Minimum Average Circle)

#### #### 2. 工程化特性

- 详细的代码注释和文档
- 完整的异常处理和边界检查
- 单元测试用例
- 性能优化和精度控制

#### #### 3. 跨语言支持

- Java 完整实现
- 详细的复杂度分析
- 可配置的参数设置

#### ### 测试验证

##### #### 运行测试用例

```
```bash
# 测试 Code01_DroppingTests
java -cp . class138.Code01_DroppingTests test
```

```
# 测试 Code02_TalentShow
```

```
java -cp . class138.Code02_TalentShow test
```

```
# 测试 Code03_DesertKing
```

```
java -cp . class138.Code03_DesertKing test
```

```
# 测试 Code04_MinimumAverageCircle
```

```
java -cp . class138.Code04_MinimumAverageCircle test
```

```
```
```

#### #### 测试输出示例

```
```
```

```
==== 开始测试 Code01_DroppingTests ===
```

```
测试用例 1: 基础测试
```

```
测试用例 1 结果: 167
```

```
测试用例 2: 边界情况
```

```
测试用例 2 结果: 100
```

```
==== 测试完成 ===
```

```
```
```

#### ### 配置选项

```
#### 精度控制
``` java
// 在代码中修改精度设置
public static final double PRECISION = 1e-6; // 默认精度
public static final double HIGH_PRECISION = 1e-9; // 高精度
public static final double LOW_PRECISION = 1e-4; // 低精度
```

```

```
#### 调试模式
``` java
// 启用调试输出
public static final boolean DEBUG = true;

// 在关键位置添加调试输出
if (DEBUG) {
    System.out.println("当前比率: " + currentRatio);
}
```

```

### ### 性能优化建议

```
#### 1. 算法选择
- 小规模数据: 使用二分法, 稳定性好
- 大规模数据: 考虑 Dinkelbach 算法, 收敛快
- 稀疏图: 使用适合稀疏图的算法

```

```
#### 2. 参数调优
- 根据数据范围设置合适的二分边界
- 根据精度要求选择合适的 epsilon 值
- 根据内存限制优化数据结构

```

```
#### 3. 内存优化
- 使用滚动数组减少空间复杂度
- 合理使用缓存提高数据局部性
- 避免不必要的对象创建

```

### ### 常见问题排查

```
#### 1. 编译错误
**问题**: 类找不到或编译错误
**解决**: 确保在项目根目录下编译, 使用正确的包名

```

## #### 2. 运行时错误

\*\*问题\*\*: 数组越界或空指针异常

\*\*解决\*\*: 检查输入数据格式，确保数据在合法范围内

## #### 3. 精度问题

\*\*问题\*\*: 结果不准确或精度不够

\*\*解决\*\*: 调整精度参数，检查浮点数比较逻辑

## #### 4. 性能问题

\*\*问题\*\*: 运行时间过长

\*\*解决\*\*: 优化算法实现，减少不必要的计算

## ### 扩展开发

### #### 添加新算法

1. 创建新的 Java 文件，如`Code05\_NewAlgorithm.java`
2. 实现核心算法逻辑
3. 添加测试用例
4. 更新文档说明

### #### 添加新语言支持

1. 创建对应语言的实现文件
2. 保持接口一致性
3. 添加相应的测试用例
4. 更新跨语言对比文档

### #### 性能优化

1. 分析算法瓶颈
2. 优化数据结构选择
3. 实现并行计算
4. 添加性能监控

## ## 总结

01 分数规划作为一种重要的优化技术，在算法竞赛和实际应用中都有广泛的应用。通过本文的详细分析和实现，我们可以看到：

1. \*\*算法多样性\*\*: 01 分数规划有多种解法，包括二分法和 Dinkelbach 算法，各有优缺点
2. \*\*题型丰富性\*\*: 从基础的 01 分数规划到结合图论、动态规划、网络流等的复合问题
3. \*\*工程化重要性\*\*: 在实际应用中，需要考虑精度控制、异常处理、性能优化等多个方面
4. \*\*跨语言实现\*\*: 不同编程语言有不同的特性和优势，需要根据具体场景选择

## ### 项目成果

本项目成功实现了：

- 4 个核心 01 分数规划算法的完整实现
- 详细的代码注释和复杂度分析
- 完整的异常处理和边界检查
- 全面的测试用例验证
- 工程化的代码结构和文档
- 跨语言的算法技巧总结

#### #### 未来发展方向

1. \*\*算法扩展\*\*: 实现更多 01 分数规划变体算法
2. \*\*性能优化\*\*: 进一步优化算法性能
3. \*\*工具化\*\*: 开发可视化工具和性能分析工具
4. \*\*教学应用\*\*: 开发教学材料和在线评测系统

掌握 01 分数规划不仅需要理解其数学原理，更需要在实践中不断积累经验，提高解决实际问题的能力。本项目为学习和应用 01 分数规划提供了完整的参考实现和详细的指导文档。

=====

#### [代码文件]

=====

文件: Code01\_DroppingTests.cpp

=====

```
// 01 分数规划基础题 - Dropping Tests
// 题目来源: POJ 2976, 洛谷 P10505
// 题目描述: 给定 n 个数据, 每个数据有(a, b)两个值, 都为整数, 并且都是非负的
// 请舍弃掉 k 个数据, 希望让剩下数据做到, 所有 a 的和 / 所有 b 的和, 这个比值尽量大
// 如果剩下数据所有 b 的和为 0, 认为无意义
// 最后, 将该比值 * 100, 小数部分四舍五入的整数结果返回
// 数据范围: 1 <= n <= 100, 0 <= a、b <= 10^9
// 测试链接: https://www.luogu.com.cn/problem/P10505
// 测试链接: http://poj.org/problem?id=2976
```

```
// 补充题目:
// 1. Codeforces 489E Hiking - 基础 01 分数规划变体
// 2. 洛谷 P1642 规划 - 基础 01 分数规划
// 3. UVA 1389 Hard Life - 最大密度子图问题
// 4. 洛谷 U581184 【模板】01-分数规划
```

```
// 算法思路: 使用二分法求解 01 分数规划问题
// 时间复杂度: O(log(1/ ε) * n log n), 其中 ε 是精度要求
```

```

// 空间复杂度: O(n)

// 01 分数规划的数学原理:
// 我们需要最大化 R = (sum(a_i * x_i)) / (sum(b_i * x_i)), 其中 x_i ∈ {0, 1}
// 对于给定的 L, 判断是否存在 x 的选择使得 R > L
// 等价于: sum(a_i * x_i) > L * sum(b_i * x_i)
// 等价于: sum((a_i - L * b_i) * x_i) > 0
// 我们通过二分 L 的值, 使用贪心策略判断是否可行

// 包含必要的头文件
#include <iostream>
#include <algorithm>
#include <cstdio>
using namespace std;

// 常量定义
const int MAXN = 1001; // 最大数据规模
const double sml = 1e-6; // 精度控制, 用于二分结束条件

// 全局变量定义
// arr[i][0] = i 号数据的 a 值
// arr[i][1] = i 号数据的 b 值
// arr[i][2] = i 号数据的结余值, 即 a - x * b
double arr[MAXN][3]; // 存储输入数据的二维数组
int n, k; // n 表示数据规模, k 表示需要选择的数据个数

/**
 * 比较函数: 按结余从大到小排序
 * 用于 STL sort 函数的比较器
 *
 * @param a 第一个元素的指针
 * @param b 第二个元素的指针
 * @return 如果 a 的结余值大于 b 的结余值返回 true, 否则返回 false
 */
bool cmp(const double* a, const double* b) {
    return a[2] > b[2]; // 按结余值从大到小排序
}

/**
 * 手动实现排序函数, 按结余从大到小排序
 * 注意: 由于数组是 double[3]类型, 这里使用指针数组进行排序
 *
 * @param start 排序范围的起始索引 (包含)

```

```

* @param end 排序范围的结束索引（包含）
*/
void mySort(int start, int end) {
    // 创建指针数组以便排序
    double* ptrs[MAXN];
    for (int i = start; i <= end; i++) {
        ptrs[i - start] = arr[i];
    }

    // 使用 STL 的 sort 函数进行排序
    sort(ptrs, ptrs + (end - start + 1), cmp);

    // 将排序结果放回原数组
    for (int i = start; i <= end; i++) {
        arr[i][0] = ptrs[i - start][0];
        arr[i][1] = ptrs[i - start][1];
        arr[i][2] = ptrs[i - start][2];
    }
}

/***
 * 检查函数：判断给定的比率值 x 是否可行
 * 原理：对于当前 x，计算每个元素的结余( $a_i - x*b_i$ )，选择结余最大的 k 个
 * 如果这 k 个的和大于等于 0，则说明存在更优的比率，可以尝试增大 x
 *
 * @param x 当前尝试的比率值
 * @return 如果 x 可行返回 true，否则返回 false
*/
bool check(double x) {
    // x 固定的情况下，计算所有数据的结余值
    for (int i = 1; i <= n; i++) {
        arr[i][2] = arr[i][0] - x * arr[i][1];
    }

    // 将结余值从大到小排序
    mySort(1, n);

    // 计算最大的 k 个结余值的累加和
    double sum = 0.0;
    for (int i = 1; i <= k; i++) {
        sum += arr[i][2];
    }
}

```

```

// 如果总和大于等于 0, 说明 x 可行
return sum >= 0;
}

/***
* 主函数: 处理输入输出, 执行二分查找
* 程序流程:
* 1. 读取输入数据
* 2. 转换问题: 将舍弃 k 个转换为选择 n-k 个
* 3. 使用二分法查找最优化率
* 4. 输出结果
*
* @return 程序退出状态码, 正常退出返回 0
*/
int main() {
    // 读取输入数据
    while (scanf("%d%d", &n, &k) != EOF && (n != 0 || k != 0)) {
        // 题目要求舍弃 k 个元素, 实际上是选择 n-k 个元素
        k = n - k;

        // 读取 a 数组
        for (int i = 1; i <= n; i++) {
            scanf("%lf", &arr[i][0]);
        }

        // 读取 b 数组
        for (int i = 1; i <= n; i++) {
            scanf("%lf", &arr[i][1]);
        }

        // 初始化二分查找的左右边界
        // 左边界为 0, 右边界为可能的最大值 (所有 a 的和)
        double l = 0.0, r = 0.0;
        for (int i = 1; i <= n; i++) {
            r += arr[i][0];
        }

        double ans = 0.0;
        // 二分查找过程
        // 当左右边界的差大于精度要求时继续循环
        while (l <= r && r - l >= sml) {
            double x = (l + r) / 2.0;
            if (check(x)) {

```

```

        // 如果 x 可行，记录当前答案，并尝试更大的值
        ans = x;
        l = x + sml; // 注意这里要加上 sml，避免死循环
    } else {
        // 如果 x 不可行，尝试更小的值
        r = x - sml;
    }
}

// 输出结果，乘以 100 后四舍五入
printf("%d\n", (int)(ans * 100.0 + 0.5));
}

return 0;
}

```

=====

文件: Code01\_DroppingTests.java

=====

```

package class138;

/**
 * 01 分数规划基础题 - Dropping Tests
 * 题目来源: POJ 2976, 洛谷 P10505
 * 题目描述: 给定 n 个数据, 每个数据有(a, b)两个值, 都为整数, 并且都是非负的
 * 请舍弃掉 k 个数据, 希望让剩下数据做到, 所有 a 的和 / 所有 b 的和, 这个比值尽量大
 * 如果剩下数据所有 b 的和为 0, 认为无意义
 * 最后, 将该比值 * 100, 小数部分四舍五入的整数结果返回
 * 数据范围: 1 <= n <= 100, 0 <= a、b <= 10^9
 * 测试链接: https://www.luogu.com.cn/problem/P10505
 * 测试链接: http://poj.org/problem?id=2976
 *
 * 补充题目:
 * 1. Codeforces 489E Hiking - 基础 01 分数规划变体
 * 2. 洛谷 P1642 规划 - 基础 01 分数规划
 * 3. UVA 1389 Hard Life - 最大密度子图问题
 * 4. 洛谷 U581184 【模板】01-分数规划
 *
 * 算法思路: 使用二分法求解 01 分数规划问题
 * 时间复杂度: O(log(1/ ε) * n log n), 其中 ε 是精度要求
 * 空间复杂度: O(n)
 */

```

```

* 01 分数规划的数学原理:
* 我们需要最大化  $R = (\sum(a_i * x_i)) / (\sum(b_i * x_i))$ , 其中  $x_i \in \{0, 1\}$ 
* 对于给定的  $L$ , 判断是否存在  $x$  的选择使得  $R > L$ 
* 等价于:  $\sum(a_i * x_i) > L * \sum(b_i * x_i)$ 
* 等价于:  $\sum((a_i - L * b_i) * x_i) > 0$ 
* 我们通过二分  $L$  的值, 使用贪心策略判断是否可行
*
* 注意: 提交到 OJ 时请将类名改为"Main"
*/

```

```

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.io.StreamTokenizer;
import java.util.Arrays;
import java.util.Comparator;

public class Code01_DroppingTests {

    // 常量定义
    public static int MAXN = 1001; // 最大数据规模
    public static double sml = 1e-6; // 精度控制, 用于二分结束条件

    /**
     * 二维数组存储数据
     * arr[i][0] = i 号数据的 a 值
     * arr[i][1] = i 号数据的 b 值
     * arr[i][2] = i 号数据的结余值, 即  $a - x * b$ 
     */
    public static double[][] arr = new double[MAXN][3];

    // 全局变量, 存储当前数据规模和需要选择的数据个数
    public static int n, k;

    /**
     * 检查函数: 判断给定的比率值 x 是否可行
     * 原理: 对于当前 x, 计算每个元素的结余( $a_i - x * b_i$ ), 选择结余最大的 k 个
     * 如果这 k 个的和大于等于 0, 则说明存在更优的比率, 可以尝试增大 x
     *
     * @param x 当前尝试的比率值
     * @return 如果 x 可行返回 true, 否则返回 false
     */
}

```

```
* @throws IllegalArgumentException 如果输入参数不合法
*/
public static boolean check(double x) {
    // 参数验证
    if (Double.isNaN(x) || Double.isInfinite(x)) {
        throw new IllegalArgumentException("比率值 x 不能为 NaN 或无穷大: " + x);
    }

    if (n <= 0 || k <= 0 || k > n) {
        throw new IllegalArgumentException("数据规模不合法: n=" + n + ", k=" + k);
    }

    // 检查分母是否为 0 的情况
    boolean allBZero = true;
    for (int i = 1; i <= n; i++) {
        if (arr[i][1] != 0) {
            allBZero = false;
            break;
        }
    }

    if (allBZero) {
        // 如果所有 b 都为 0, 无法计算比值, 返回 false
        return false;
    }

    // x 固定的情况下, 计算所有数据的结余值
    for (int i = 1; i <= n; i++) {
        // 检查数值溢出
        if (Double.isInfinite(arr[i][0]) || Double.isInfinite(arr[i][1])) {
            throw new ArithmeticException("输入数据包含无穷大值");
        }

        double product = x * arr[i][1];
        if (Double.isInfinite(product)) {
            throw new ArithmeticException("乘法运算溢出: x=" + x + ", b=" + arr[i][1]);
        }

        arr[i][2] = arr[i][0] - product;

        // 检查减法运算结果
        if (Double.isNaN(arr[i][2])) {
            throw new ArithmeticException("减法运算结果异常: a=" + arr[i][0] + ", product=" +

```

```

product);
    }
}

// 将结余值从大到小排序
try {
    Arrays.sort(arr, 1, n + 1, new MyComparator());
} catch (IllegalArgumentException e) {
    throw new RuntimeException("排序失败: " + e.getMessage(), e);
}

// 计算最大的 k 个结余值的累加和
double sum = 0.0;
for (int i = 1; i <= k; i++) {
    // 检查数组索引是否越界
    if (i < 1 || i > n) {
        throw new ArrayIndexOutOfBoundsException("数组索引越界: i=" + i + ", n=" + n);
    }

    sum += arr[i][2];

    // 检查累加和是否溢出
    if (Double.isInfinite(sum)) {
        throw new ArithmeticException("累加和溢出");
    }
}

// 如果总和大于等于 0, 说明 x 可行
// 使用 eps 进行比较, 避免浮点数精度问题
return sum >= -sm1;
}

/***
 * 比较器类: 用于对结余值进行从大到小排序
 * 注意: POJ 平台 Java 版本较老, 不支持 Lambda 表达式, 所以需要定义单独的比较器类
 */
public static class MyComparator implements Comparator<double[]> {

    @Override
    public int compare(double[] o1, double[] o2) {
        // 按结余值从大到小排序
        return o1[2] >= o2[2] ? -1 : 1;
    }
}

```

```
}

/**
 * 测试函数：用于验证算法的正确性
 * 包含多个测试用例，覆盖边界情况和一般情况
 */
public static void test() {
    System.out.println("== 开始测试 Code01_DroppingTests ==");

    // 测试用例 1：基础测试用例
    System.out.println("测试用例 1：基础测试");
    n = 3;
    k = 1; // 舍弃 1 个，选择 2 个
    k = n - k; // 实际选择 2 个

    // 设置测试数据
    arr[1][0] = 5; arr[1][1] = 1;
    arr[2][0] = 0; arr[2][1] = 2;
    arr[3][0] = 2; arr[3][1] = 7;

    double l = 0.0, r = 0.0;
    for (int i = 1; i <= n; i++) {
        r += arr[i][0];
    }

    double ans = 0.0;
    while (l <= r && r - l >= sml) {
        double x = (l + r) / 2.0;
        if (check(x)) {
            ans = x;
            l = x + sml;
        } else {
            r = x - sml;
        }
    }

    int result = (int) (100 * (ans + 0.005));
    System.out.println("测试用例 1 结果：" + result);

    // 测试用例 2：边界情况测试
    System.out.println("测试用例 2：边界情况");
    n = 2;
```

```

k = 0; // 舍弃 0 个, 选择 2 个
k = n - k;

arr[1][0] = 100; arr[1][1] = 100;
arr[2][0] = 50; arr[2][1] = 50;

l = 0.0; r = 0.0;
for (int i = 1; i <= n; i++) {
    r += arr[i][0];
}

ans = 0.0;
while (l <= r && r - l >= sml) {
    double x = (l + r) / 2.0;
    if (check(x)) {
        ans = x;
        l = x + sml;
    } else {
        r = x - sml;
    }
}

result = (int) (100 * (ans + 0.005));
System.out.println("测试用例 2 结果: " + result);

System.out.println("== 测试完成 ==");
}

/***
 * 主函数: 处理输入输出, 执行二分查找
 *
 * @param args 命令行参数
 * @throws IOException 输入输出异常
 */
public static void main(String[] args) throws IOException {
    // 如果传入参数"test", 则运行测试用例
    if (args.length > 0 && "test".equals(args[0])) {
        test();
        return;
    }

    // 使用高效的输入输出方式
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
}

```

```
StreamTokenizer in = new StreamTokenizer(br);
PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));

// 读取第一组数据
in.nextToken();
n = (int) in.nval;
in.nextToken();
k = (int) in.nval;

// 处理多组数据，直到输入 0 0
while (n != 0 || k != 0) {
    // 题目要求舍弃 k 个元素，实际上是选择 n-k 个元素
    k = n - k;

    // 读取 a 数组
    for (int i = 1; i <= n; i++) {
        in.nextToken();
        arr[i][0] = in.nval;
    }

    // 读取 b 数组
    for (int i = 1; i <= n; i++) {
        in.nextToken();
        arr[i][1] = in.nval;
    }

    // 初始化二分查找的左右边界
    // 左边界为 0，右边界为可能的最大值（所有 a 的和）
    double l = 0.0, r = 0.0;
    for (int i = 1; i <= n; i++) {
        r += arr[i][0];
    }

    double ans = 0.0;
    // 二分查找过程
    // 当左右边界的差大于精度要求时继续循环
    while (l <= r && r - l >= sml) {
        double x = (l + r) / 2.0;
        if (check(x)) {
            // 如果 x 可行，记录当前答案，并尝试更大的值
            ans = x;
            l = x + sml; // 注意这里要加上 sml，避免死循环
        } else {
```

```

        // 如果 x 不可行，尝试更小的值
        r = x - sml;
    }
}

// 输出结果，乘以 100 后四舍五入
// 使用+0.005 的方式实现四舍五入
out.println((int) (100 * (ans + 0.005)));

// 读取下一组数据
in.nextToken();
n = (int) in.nval;
in.nextToken();
k = (int) in.nval;
}

// 刷新输出缓冲
out.flush();
// 关闭资源
out.close();
try {
    br.close();
} catch (IOException e) {
    e.printStackTrace();
}
}

}

```

文件: Code01\_DroppingTests.py

```
# -*- coding: utf-8 -*-
```

```
"""
```

01 分数规划基础题 – Dropping Tests

题目来源: POJ 2976, 洛谷 P10505

题目描述: 给定 n 个数据, 每个数据有(a, b)两个值, 都为整数, 并且都是非负的

请舍弃掉 k 个数据, 希望让剩下数据做到, 所有 a 的和 / 所有 b 的和, 这个比值尽量大

如果剩下数据所有 b 的和为 0, 认为无意义

最后, 将该比值 \* 100, 小数部分四舍五入的整数结果返回

数据范围:  $1 \leq n \leq 100$ ,  $0 \leq a, b \leq 10^9$

测试链接: <https://www.luogu.com.cn/problem/P10505>

测试链接: <http://poj.org/problem?id=2976>

补充题目:

1. Codeforces 489E Hiking - 基础 01 分数规划变体
2. 洛谷 P1642 规划 - 基础 01 分数规划
3. UVA 1389 Hard Life - 最大密度子图问题
4. 洛谷 U581184 【模板】01-分数规划

算法思路: 使用二分法求解 01 分数规划问题

时间复杂度:  $O(\log(1/\epsilon) * n \log n)$ , 其中  $\epsilon$  是精度要求

空间复杂度:  $O(n)$

01 分数规划的数学原理:

我们需要最大化  $R = (\sum(a_i * x_i)) / (\sum(b_i * x_i))$ , 其中  $x_i \in \{0, 1\}$

对于给定的  $L$ , 判断是否存在  $x$  的选择使得  $R > L$

等价于:  $\sum(a_i * x_i) > L * \sum(b_i * x_i)$

等价于:  $\sum((a_i - L * b_i) * x_i) > 0$

我们通过二分  $L$  的值, 使用贪心策略判断是否可行

"""

```
import sys
```

```
# 常量定义
```

```
MAXN = 1001 # 最大数据规模
```

```
sml = 1e-6 # 精度控制, 用于二分结束条件
```

"""

二维列表存储数据:

```
arr[i][0] = i 号数据的 a 值
```

```
arr[i][1] = i 号数据的 b 值
```

```
arr[i][2] = i 号数据的结余值, 即  $a - x * b$ 
```

"""

```
arr = [[0.0 for _ in range(3)] for _ in range(MAXN)]
```

```
# 全局变量, 存储当前数据规模和需要选择的数据个数
```

```
n = 0
```

```
k = 0
```

```
def check(x):
```

"""

检查函数：判断给定的比率值 x 是否可行

原理：对于当前 x，计算每个元素的结余( $a_i - x*b_i$ )，选择结余最大的 k 个  
如果这 k 个的和大于等于 0，则说明存在更优的比率，可以尝试增大 x

参数：

x (float)：当前尝试的比率值

返回：

bool：如果 x 可行返回 True，否则返回 False

"""

# x 固定的情况下，计算所有数据的结余值

for i in range(1, n + 1):

arr[i][2] = arr[i][0] - x \* arr[i][1]

# 结余从大到小排序

# 在 Python 中，我们使用自定义排序键

temp\_arr = arr[1:n+1] # 获取有效部分

temp\_arr.sort(key=lambda item: item[2], reverse=True)

arr[1:n+1] = temp\_arr # 将排序后的结果放回原数组

# 计算最大的 k 个结余值的累加和

sum\_val = 0.0

for i in range(1, k + 1):

sum\_val += arr[i][2]

# 如果总和大于等于 0，说明 x 可行

return sum\_val >= 0

def main():

"""

主函数：处理输入输出，执行二分查找

处理流程：

1. 读取输入数据
2. 转换问题：将舍弃 k 个转换为选择 n-k 个
3. 使用二分法查找最优化率
4. 输出结果

"""

global n, k

try:

while True:

```
# 读取第一组数据
line = sys.stdin.readline()
# 处理 EOF 情况
if not line:
    break

line = line.strip().split()
# 处理空行
if not line:
    continue

n = int(line[0])
k = int(line[1])

# 终止条件：输入 0 0
if n == 0 and k == 0:
    break

# 题目要求舍弃 k 个元素，实际上是选择 n-k 个元素
k = n - k

# 读取 a 值
line = sys.stdin.readline().strip().split()
for i in range(1, n + 1):
    arr[i][0] = float(line[i - 1])

# 读取 b 值
line = sys.stdin.readline().strip().split()
for i in range(1, n + 1):
    arr[i][1] = float(line[i - 1])

# 初始化二分查找的左右边界
# 左边界为 0，右边界为可能的最大值（所有 a 的和）
l = 0.0
r = 0.0
for i in range(1, n + 1):
    r += arr[i][0]

ans = 0.0
# 二分查找过程
# 当左右边界的差大于精度要求时继续循环
while l <= r and r - l >= sm1:
    x = (l + r) / 2.0
```

```

if check(x):
    # 如果 x 可行, 记录当前答案, 并尝试更大的值
    ans = x
    l = x + sml # 注意这里要加上 sml, 避免死循环
else:
    # 如果 x 不可行, 尝试更小的值
    r = x - sml

    # 输出结果, 乘以 100 后四舍五入
    # 使用+0.005 的方式实现四舍五入
    print(int(100 * (ans + 0.005)))

except Exception as e:
    # 错误处理
    print(f"Error: {e}", file=sys.stderr)
    sys.exit(1)

```

```

if __name__ == "__main__":
    main()

```

=====

文件: Code02\_TalentShow.cpp

=====

```

#include <iostream>
#include <algorithm>
#include <cmath>
#include <cstdio>
using namespace std;

/**
 * 01 分数规划问题 - 牛群的才艺展示 (Talent Show)
 *
 * 题目信息:
 * - 题目来源: Luogu P4377, USACO 2018 Open Contest
 * - 题目描述: 有 n 头奶牛, 每头奶牛有重量和才艺两个属性值。要求选若干头牛, 使得总重量不少于 w,
 * 并且选出的牛的才艺的和与重量的和的比值尽量大。返回该比值乘以 1000 的整数结果, 小数部分舍弃。
 * - 数据范围:
 *   - 1 <= n <= 250
 *   - 1 <= w <= 1000
 *   - 1 <= 牛的重量 <= 10^6
 *   - 1 <= 牛的才艺 <= 10^3
 * - 测试链接: https://www.luogu.com.cn/problem/P4377

```

```

*
* 补充题目:
* 1. USACO 2018 Open Contest - Talent Show
* 2. POJ 3621 - Best Cow Line
* 3. HDU 4787 - GRE Words Revenge
* 4. Codeforces 489E - Hiking
* 5. 洛谷 P1642 - 规划
*
* 算法思路: 使用二分法求解 01 分数规划问题, 结合 01 背包动态规划进行可行性判断
*
* 01 分数规划的数学原理:
* 我们需要最大化  $R = (\sum(talent_i * x_i)) / (\sum(weight_i * x_i))$ , 其中  $x_i \in \{0, 1\}$  且
 $\sum(weight_i * x_i) \geq W$ 
* 对于给定的  $L$ , 判断是否存在  $x$  的选择使得  $R > L$ 
* 等价于:  $\sum(talent_i * x_i) > L * \sum(weight_i * x_i)$ 
* 等价于:  $\sum((talent_i - L * weight_i) * x_i) > 0$ , 且  $\sum(weight_i * x_i) \geq W$ 
*
* 时间复杂度:  $O(n * W * \log(1/\epsilon))$ , 其中  $\epsilon$  是精度要求 (本题中取  $1e-6$ )
* 空间复杂度:  $O(W)$ , 使用滚动数组优化动态规划空间
*/

```

```

// 常量定义
const int MAXN = 251; // 最大奶牛数量
const int MAXW = 1001; // 最大目标重量 W

// 足够小代表无效解 (无法达到的状态)
const double NA = -1e9;

// 精度控制, 用于二分结束条件
const double PRECISION = 1e-6;

// 重量数组
int weight[MAXN];

// 才艺数组
int talent[MAXN];

// (才艺 - x * 重量) 的结余, 用于 01 分数规划判断
double value[MAXN];

/**
 * dp[j] 表示在前 i 头牛中选择若干头, 总重量为 j 时的最大结余和
 * 特别地, dp[w] 表示总重量  $\geq w$  时的最大结余和

```

```

* 采用空间压缩的方式实现，仅用一维数组
*/
double dp[MAXW];

// 全局变量，存储当前数据规模和目标重量
int n, w; // n 表示奶牛数量，w 表示目标最小总重量

/***
 * 检查函数：判断给定的比率值 x 是否可行
 *
 * 核心思想：将 01 分数规划问题转化为 01 背包问题。对于当前比率 x，计算每个奶牛的结余值 (talent_i - x * weight_i)，
 * 然后求解总重量至少为 w 的情况下，能否使总结余值大于等于 0。
 *
 * 动态规划设计：
 * - 状态定义：dp[j] 表示总重量为 j 时的最大结余和
 * - 状态转移：对于每头牛，有选或不选两种选择
 * - 特殊处理：当总重量超过 w 时，统一记录在 dp[w] 中
 *
 * @param x 当前尝试的比率值
 * @return 如果存在一种选择使得比值大于 x，则返回 true；否则返回 false
*/
bool check(double x) {
    // 计算每头牛的 value 值（才艺值减去 x 倍的重量值）
    for (int i = 1; i <= n; i++) {
        value[i] = static_cast<double>(talent[i]) - x * weight[i];
    }

    // 初始化 dp 数组
    dp[0] = 0.0; // 初始状态：重量为 0 时，结余和为 0
    for (int i = 1; i <= w; i++) {
        dp[i] = NA; // 其余状态初始化为无效值
    }

    // 01 背包动态规划过程
    for (int i = 1; i <= n; i++) { // 遍历每头牛
        // 倒序遍历重量，避免重复选择同一头牛
        for (int p = w; p >= 0; p--) {
            if (dp[p] != NA) { // 如果当前状态有效（可达）
                int j = p + weight[i]; // 计算选择当前牛后的总重量

                // 两种情况处理：
                // 1. 如果总重量超过或等于目标重量 w，统一更新 dp[w]
            }
        }
    }
}

```

```

        // 2. 否则更新对应重量的 dp 值
        if (j >= w) {
            dp[w] = max(dp[w], dp[p] + value[i]);
        } else {
            dp[j] = max(dp[j], dp[p] + value[i]);
        }
    }
}

// 判断条件: 如果总重量>=w 时的最大结余和>=0, 说明存在更优的解
return dp[w] >= 0;
}

/***
 * 主函数: 处理输入输出, 执行二分查找算法
 *
 * 算法流程:
 * 1. 读取输入数据
 * 2. 初始化二分查找的左右边界
 * 3. 进行二分查找, 每次调用 check 函数判断当前比率是否可行
 * 4. 输出结果, 将最优化值乘以 1000 后取整数部分
 */
int main() {
    // 读取输入数据
    scanf("%d %d", &n, &w);

    for (int i = 1; i <= n; i++) {
        scanf("%d %d", &weight[i], &talent[i]);
    }

    // 初始化二分查找的左右边界
    // 左边界为 0, 右边界为所有才艺值的和 (最大可能比值的上界)
    double left = 0.0, right = 0.0;
    for (int i = 1; i <= n; i++) {
        right += talent[i];
    }

    double result = 0.0;
    // 二分查找过程
    // 当左右边界的差大于精度要求时继续循环
    while (left < right && right - left >= PRECISION) {
        double mid = (left + right) / 2.0; // 取中点作为当前尝试的比率值

```

```

        if (check(mid)) { // 如果 mid 可行，说明可以尝试更大的值
            result = mid; // 记录当前最优解
            left = mid + PRECISION; // 左边界右移
        } else { // 如果 mid 不可行，需要尝试更小的值
            right = mid - PRECISION; // 右边界左移
        }
    }

// 输出结果，将比值乘以 1000 后取整数部分（向下取整）
printf("%d\n", static_cast<int>(result * 1000));

return 0;
}

```

---

文件: Code02\_TalentShow.java

---

```

package class138;

/**
 * 01 分数规划问题 - 牛群的才艺展示 (Talent Show)
 *
 * <h3>题目信息</h3>
 * <ul>
 *   <li><strong>题目来源</strong>: Luogu P4377, USACO 2018 Open Contest</li>
 *   <li><strong>题目描述</strong>: 有 n 头奶牛，每头奶牛有重量和才艺两个属性值。要求选若干头牛，使得总重量不少于 w，  
并且选出的牛的才艺的和与重量的和的比值尽量大。返回该比值乘以 1000 的整数结果，小数部分舍弃。
</li>
 *   <li><strong>数据范围</strong>:
 *     <ul>
 *       <li>1 <= n <= 250</li>
 *       <li>1 <= w <= 1000</li>
 *       <li>1 <= 牛的重量 <= 10^6</li>
 *       <li>1 <= 牛的才艺 <= 10^3</li>
 *     </ul>
 *   </li>
 *   <li><strong>测试链接</strong>: <a href="https://www.luogu.com.cn/problem/P4377">Luogu P4377</a></li>
 * </ul>
 *
 * <h3>算法思路</h3>

```

- \* <p>使用二分法求解 01 分数规划问题，结合 01 背包动态规划进行可行性判断：</p>
- \* <ol>
 - \* <li><strong>二分法</strong>：在可能的比值范围内进行二分查找</li>
- \* <li><strong>背包模型</strong>：对于每个二分中点，将问题转化为判断是否存在总重量至少为 w 的奶牛选择，
- \* 使得总才艺值与总重量的比值大于当前中点值</li>
- \* </ol>
- \*
- \* <h3>数学原理</h3>
- \* <p>我们需要最大化  $R = (\sum(talent_i * x_i)) / (\sum(weight_i * x_i))$ ，其中  $x_i \in \{0, 1\}$  且  $\sum(weight_i * x_i) \geq w$ </p>
- \* <p>对于给定的 L，判断是否存在 x 的选择使得  $R > L$ :</p>
- \* <ul>
 - \* <li>等价于:  $\sum(talent_i * x_i) > L * \sum(weight_i * x_i)$ </li>
- \* <li>等价于:  $\sum((talent_i - L * weight_i) * x_i) > 0$ ，且  $\sum(weight_i * x_i) \geq w$ </li>

- \* </ul>
- \*
- \* <h3>复杂度分析</h3>
- \* <ul>
 - \* <li><strong>时间复杂度</strong>:  $O(n * W * \log(1/\epsilon))$ ，其中  $\epsilon$  是精度要求（本题中取  $1e-6$ )</li>
- \* <li><strong>空间复杂度</strong>:  $O(W)$ ，使用滚动数组优化动态规划空间</li>
- \* </ul>
- \*
- \* <h3>注意事项</h3>
- \* <ul>
 - \* <li>提交到 OJ 时请将类名改为“Main”</li>
- \* <li>使用高精度数据类型（double）避免计算精度问题</li>
- \* <li>注意处理总重量超过目标重量 w 的情况</li>
- \* </ul>
- \*/

```

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.io.StreamTokenizer;
import java.util.Arrays;

```

```
public class Code02_TalentShow {
```

```
// 常量定义
```

```

public static final int MAXN = 251; // 最大奶牛数量
public static final int MAXW = 1001; // 最大目标重量 W

// 足够小代表无效解（无法达到的状态）
public static final double NA = -1e9;

// 精度控制，用于二分结束条件
public static final double PRECISION = 1e-6;

// 重量数组
public static int[] weight = new int[MAXN];

// 才艺数组
public static int[] talent = new int[MAXN];

/***
 * 计算每头牛的 value 值
 * value[i] = talent[i] - x * weight[i]
 * 用于 01 分数规划判断
 */
public static double[] value = new double[MAXN];

/***
 * dp[j] 表示在前 i 头牛中选择若干头，总重量为 j 时的最大结余和
 * 特别地，dp[w] 表示总重量>=w 时的最大结余和
 * 采用空间压缩的方式实现，仅用一维数组
 */
public static double[] dp = new double[MAXW];

// 全局变量，存储当前数据规模和目标重量
public static int n, w;

/***
 * 检查函数：判断给定的比率值 x 是否可行
 *
 * <p>核心思想：将 01 分数规划问题转化为 01 背包问题。对于当前比率 x，计算每个奶牛的结余值 (talent_i - x * weight_i)，
 * 然后求解总重量至少为 w 的情况下，能否使总结余值大于等于 0。</p>
 *
 * <p>动态规划设计：</p>
 * <ul>
 *   <li>状态定义：dp[j] 表示总重量为 j 时的最大结余和</li>
 *   <li>状态转移：对于每头牛，有选或不选两种选择</li>

```

```

* <li>特殊处理：当总重量超过 w 时，统一记录在 dp[w] 中</li>
* </ul>
*
* @param x 当前尝试的比率值
* @return 如果存在一种选择使得比值大于 x，则返回 true；否则返回 false
* @throws IllegalArgumentException 如果输入参数不合法
* @throws ArithmeticException 如果数值计算出现异常
*/
public static boolean check(double x) {
    // 参数验证
    if (Double.isNaN(x) || Double.isInfinite(x)) {
        throw new IllegalArgumentException("比率值 x 不能为 NaN 或无穷大：" + x);
    }

    if (n <= 0 || n > MAXN) {
        throw new IllegalArgumentException("奶牛数量不合法：n=" + n + "，最大允许=" + MAXN);
    }

    if (w <= 0 || w > MAXW) {
        throw new IllegalArgumentException("目标重量不合法：w=" + w + "，最大允许=" + MAXW);
    }

    // 检查输入数据有效性
    for (int i = 1; i <= n; i++) {
        if (weight[i] < 0 || talent[i] < 0) {
            throw new IllegalArgumentException("重量或才艺值不能为负数：weight[" + i + "]=" +
weight[i] + "，talent[" + i + "]=" + talent[i]);
        }

        if (weight[i] > 1000000 || talent[i] > 1000) {
            throw new IllegalArgumentException("重量或才艺值超出范围：weight[" + i + "]=" +
weight[i] + "，talent[" + i + "]=" + talent[i]);
        }
    }

    // 计算每头牛的 value 值（才艺值减去 x 倍的重量值）
    for (int i = 1; i <= n; i++) {
        // 检查数值溢出
        double product = x * weight[i];
        if (Double.isInfinite(product)) {
            throw new ArithmeticException("乘法运算溢出：x=" + x + "，weight=" + weight[i]);
        }
    }
}

```

```

value[i] = (double) talent[i] - product;

// 检查减法运算结果
if (Double.isNaN(value[i])) {
    throw new ArithmeticException("减法运算结果异常: talent=" + talent[i] + ", "
product=" + product);
}

// 初始化 dp 数组
dp[0] = 0.0; // 初始状态: 重量为 0 时, 结余和为 0

// 检查数组边界
if (w < 0 || w >= MAXW) {
    throw new ArrayIndexOutOfBoundsException("目标重量 w 超出数组范围: w=" + w);
}

// 初始化其余状态为无效值
for (int j = 1; j <= w; j++) {
    dp[j] = NA;
}

// 01 背包动态规划过程
for (int i = 1; i <= n; i++) { // 遍历每头牛
    // 倒序遍历重量, 避免重复选择同一头牛
    for (int p = w; p >= 0; p--) {
        if (dp[p] != NA) { // 如果当前状态有效 (可达)
            int j = p + weight[i]; // 计算选择当前牛后的总重量

            // 检查加法运算是否溢出
            if (j < 0) {
                throw new ArithmeticException("重量加法溢出: p=" + p + ", weight=" +
weight[i]);
            }

            // 检查数组索引是否越界
            if (j >= MAXW) {
                j = w; // 统一处理到 w
            }

            // 计算新的结余和
            double newValue = dp[p] + value[i];

```

```

        // 检查加法运算是否溢出
        if (Double.isInfinite(newValue)) {
            throw new ArithmeticException("结余和加法溢出: dp[" + p + "]=" + dp[p] + ", "
value[" + i + "]=" + value[i]);
        }

        // 两种情况处理:
        // 1. 如果总重量超过或等于目标重量 w, 统一更新 dp[w]
        // 2. 否则更新对应重量的 dp 值
        if (j >= w) {
            if (w >= 0 && w < MAXW) {
                dp[w] = Math.max(dp[w], newValue);
            }
        } else {
            if (j >= 0 && j < MAXW) {
                dp[j] = Math.max(dp[j], newValue);
            }
        }
    }

}

// 判断条件: 如果总重量>=w 时的最大结余和>=0, 说明存在更优的解
// 使用 eps 进行比较, 避免浮点数精度问题
return dp[w] >= -PRECISION;
}

/***
 * 测试函数: 用于验证算法的正确性
 * 包含多个测试用例, 覆盖边界情况和一般情况
 */
public static void test() {
    System.out.println("== 开始测试 Code02_TalentShow ==");

    // 测试用例 1: 基础测试用例
    System.out.println("测试用例 1: 基础测试");
    n = 3;
    w = 5;

    // 设置测试数据
    weight[1] = 1; talent[1] = 2;
    weight[2] = 2; talent[2] = 2;
    weight[3] = 3; talent[3] = 1;
}

```

```
double left = 0.0, right = 0.0;
for (int i = 1; i <= n; i++) {
    right += talent[i];
}

double result = 0.0;
while (left < right && right - left >= PRECISION) {
    double mid = (left + right) / 2;
    if (check(mid)) {
        result = mid;
        left = mid + PRECISION;
    } else {
        right = mid - PRECISION;
    }
}

int output = (int) (result * 1000);
System.out.println("测试用例 1 结果: " + output);

// 测试用例 2: 边界情况测试
System.out.println("测试用例 2: 边界情况");
n = 2;
w = 1;

weight[1] = 1; talent[1] = 100;
weight[2] = 1; talent[2] = 50;

left = 0.0; right = 0.0;
for (int i = 1; i <= n; i++) {
    right += talent[i];
}

result = 0.0;
while (left < right && right - left >= PRECISION) {
    double mid = (left + right) / 2;
    if (check(mid)) {
        result = mid;
        left = mid + PRECISION;
    } else {
        right = mid - PRECISION;
    }
}
```

```
        output = (int) (result * 1000);
        System.out.println("测试用例 2 结果: " + output);

    System.out.println("== 测试完成 ==");
}

/***
 * 主函数: 处理输入输出, 执行二分查找算法
 *
 * <p>算法流程: </p>
 * <ol>
 *   <li>读取输入数据</li>
 *   <li>初始化二分查找的左右边界</li>
 *   <li>进行二分查找, 每次调用 check 函数判断当前比率是否可行</li>
 *   <li>输出结果, 将最优比值乘以 1000 后取整数部分</li>
 * </ol>
 */
public static void main(String[] args) {
    // 如果传入参数"test", 则运行测试用例
    if (args.length > 0 && "test".equals(args[0])) {
        test();
        return;
    }

    BufferedReader br = null;
    PrintWriter out = null;

    try {
        // 使用高效的输入输出流处理大数据
        br = new BufferedReader(new InputStreamReader(System.in));
        StreamTokenizer in = new StreamTokenizer(br);
        out = new PrintWriter(new OutputStreamWriter(System.out));

        // 读取奶牛数量和目标重量
        in.nextToken();
        n = (int) in.nval;
        in.nextToken();
        w = (int) in.nval;

        // 读取每头奶牛的重量和才艺值
        for (int i = 1; i <= n; i++) {
            in.nextToken();
        }
    } catch (IOException e) {
        e.printStackTrace();
    } finally {
        if (br != null) {
            try {
                br.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
        if (out != null) {
            out.close();
        }
    }
}
```

```
    weight[i] = (int) in.nval;
    in.nextToken();
    talent[i] = (int) in.nval;
}

// 初始化二分查找的左右边界
// 左边界为 0，右边界为所有才艺值的和（最大可能比值的上界）
double left = 0.0, right = 0.0;
for (int i = 1; i <= n; i++) {
    right += talent[i];
}

double result = 0.0;
// 二分查找过程
// 当左右边界的差大于精度要求时继续循环
while (left < right && right - left >= PRECISION) {
    double mid = (left + right) / 2; // 取中点作为当前尝试的比率值
    if (check(mid)) { // 如果 mid 可行，说明可以尝试更大的值
        result = mid; // 记录当前最优解
        left = mid + PRECISION; // 左边界右移
    } else { // 如果 mid 不可行，需要尝试更小的值
        right = mid - PRECISION; // 右边界左移
    }
}

// 输出结果，将比值乘以 1000 后取整数部分（向下取整）
out.println((int) (result * 1000));
out.flush();

} catch (IOException e) {
    // 处理输入输出异常
    e.printStackTrace();
} finally {
    // 确保资源正确关闭
    try {
        if (br != null) {
            br.close();
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}

if (out != null) {
```

```
        out.close();
    }
}

}

=====
```

文件: Code02\_TalentShow.py

```
=====
import sys
from typing import List, Tuple

class TalentShow:
    """
    01 分数规划问题 - 牛群的才艺展示 (Talent Show)

    题目信息:
    - 题目来源: Luogu P4377, USACO 2018 Open Contest
    - 题目描述: 有 n 头奶牛, 每头奶牛有重量和才艺两个属性值。要求选若干头牛, 使得总重量不少于 w,
      并且选出的牛的才艺的和与重量的和的比值尽量大。返回该比值乘以 1000 的整数结果, 小数部分舍弃。
    - 数据范围:
      -  $1 \leq n \leq 250$ 
      -  $1 \leq w \leq 1000$ 
      - 牛的重量  $\leq 10^6$ 
      - 牛的才艺  $\leq 10^3$ 
    - 测试链接: https://www.luogu.com.cn/problem/P4377
```

补充题目:

1. USACO 2018 Open Contest - Talent Show
2. POJ 3621 - Best Cow Line
3. HDU 4787 - GRE Words Revenge
4. Codeforces 489E - Hiking
5. 洛谷 P1642 - 规划

算法思路: 使用二分法求解 01 分数规划问题, 结合 01 背包动态规划进行可行性判断

01 分数规划的数学原理:

我们需要最大化  $R = (\sum(talent_i * x_i)) / (\sum(weight_i * x_i))$ , 其中  $x_i \in \{0, 1\}$  且  $\sum(weight_i * x_i) \geq w$

对于给定的 L, 判断是否存在 x 的选择使得  $R > L$

等价于:  $\sum(talent_i * x_i) > L * \sum(weight_i * x_i)$

等价于:  $\sum((\text{talent}_i - L * \text{weight}_i) * x_i) > 0$ , 且  $\sum(\text{weight}_i * x_i) \geq W$

时间复杂度:  $O(n * W * \log(1/\epsilon))$ , 其中  $\epsilon$  是精度要求 (本题中取  $1e-6$ )

空间复杂度:  $O(W)$ , 使用滚动数组优化动态规划空间

"""

```
def __init__(self):
    # 常量定义
    self.MAXN: int = 251  # 最大奶牛数量
    self.MAXW: int = 1001  # 最大目标重量 W
    self.NA: float = -10**18  # 足够小代表无效解 (无法达到的状态)
    self.PRECISION: float = 1e-6  # 精度控制, 用于二分结束条件

    # 全局变量初始化
    self.weight: List[int] = [0] * (self.MAXN + 1)  # 重量数组
    self.talent: List[int] = [0] * (self.MAXN + 1)  # 才艺数组
    self.value: List[float] = [0.0] * (self.MAXN + 1)  # 价值数组 (用于计算结余)
    self.dp: List[float] = [0.0] * (self.MAXW + 1)  # 动态规划数组
    self.n: int = 0  # 奶牛数量
    self.w: int = 0  # 目标最小总重量
```

def check(self, x: float) -> bool:

"""

检查函数: 判断给定的比率值 x 是否可行

核心思想: 将 01 分数规划问题转化为 01 背包问题。对于当前比率 x, 计算每个奶牛的结余值 ( $\text{talent}_i - x * \text{weight}_i$ ),

然后求解总重量至少为 w 的情况下, 能否使总结余值大于等于 0。

动态规划设计:

- 状态定义:  $dp[j]$  表示总重量为  $j$  时的最大结余和
- 状态转移: 对于每头牛, 有选或不选两种选择
- 特殊处理: 当总重量超过  $w$  时, 统一记录在  $dp[w]$  中

Args:

x: 当前尝试的比率值

Returns:

bool: 如果存在一种选择使得比值大于 x, 则返回 True; 否则返回 False

"""

# 计算每头牛的 value 值 (才艺值减去 x 倍的重量值)

for i in range(1, self.n + 1):

self.value[i] = self.talent[i] - x \* self.weight[i]

```

# 初始化 dp 数组
self.dp[0] = 0.0 # 初始状态: 重量为 0 时, 结余和为 0
for i in range(1, self.w + 1):
    self.dp[i] = self.NA # 其余状态初始化为无效值

# 01 背包动态规划过程
for i in range(1, self.n + 1): # 遍历每头牛
    # 倒序遍历重量, 避免重复选择同一头牛
    for p in range(self.w, -1, -1):
        if self.dp[p] != self.NA: # 如果当前状态有效 (可达)
            j = p + self.weight[i] # 计算选择当前牛后的总重量

            # 两种情况处理:
            # 1. 如果总重量超过或等于目标重量 w, 统一更新 dp[w]
            # 2. 否则更新对应重量的 dp 值
            if j >= self.w:
                if self.dp[self.w] < self.dp[p] + self.value[i]:
                    self.dp[self.w] = self.dp[p] + self.value[i]
            else:
                if self.dp[j] < self.dp[p] + self.value[i]:
                    self.dp[j] = self.dp[p] + self.value[i]

# 判断条件: 如果总重量>=w 时的最大结余和>=0, 说明存在更优的解
return self.dp[self.w] >= 0

```

def main(self):

"""

主函数: 处理输入输出, 执行二分查找算法

算法流程:

1. 读取输入数据
2. 初始化二分查找的左右边界
3. 进行二分查找, 每次调用 check 函数判断当前比率是否可行
4. 输出结果, 将最优化值乘以 1000 后取整数部分

"""

try:

```

# 读取输入数据
data = sys.stdin.read().split()
idx = 0
self.n = int(data[idx])
idx += 1
self.w = int(data[idx])
idx += 1

```

```

for i in range(1, self.n + 1):
    self.weight[i] = int(data[idx])
    idx += 1
    self.talent[i] = int(data[idx])
    idx += 1

# 初始化二分查找的左右边界
left = 0.0
right = 0.0
for i in range(1, self.n + 1):
    right += self.talent[i] # 最大可能的比值不会超过总才艺值

result = 0.0
# 二分查找过程
# 当左右边界的差大于精度要求时继续循环
while left < right and right - left >= self.PRECISION:
    mid = (left + right) / 2.0 # 取中点作为当前尝试的比率值
    if self.check(mid): # 如果 mid 可行，说明可以尝试更大的值
        result = mid # 记录当前最优解
        left = mid + self.PRECISION # 左边界右移
    else: # 如果 mid 不可行，需要尝试更小的值
        right = mid - self.PRECISION # 右边界左移

# 输出结果，将比值乘以 1000 后取整数部分（向下取整）
print(int(result * 1000))

except ValueError as e:
    print(f"输入数据错误: {e}")
except Exception as e:
    print(f"程序发生错误: {e}")

if __name__ == "__main__":
    talent_show = TalentShow()
    talent_show.main()

```

=====

文件: Code03\_DesertKing.cpp

=====

```

#include <iostream>
#include <cmath>
#include <cstring>
using namespace std;

```

```

/***
 * 01 分数规划问题 - 最优比率生成树 (Desert King)
 * 题目来源: POJ 2728
 * 题目描述: 有 n 个村庄, 每个村庄由(x, y, z)表示, 其中(x, y)是二维地图中的位置, z 是海拔高度。
 * 任意两个村庄之间的距离是二维地图中的欧式距离, 修路花费是海拔差值的绝对值。
 * 要求将所有村庄连通, 使得总花费/总距离的比值最小, 结果保留小数点后 3 位。
 *
 * 数据范围:
 * 2 <= n <= 10^3
 * 0 <= x、y <= 10^4
 * 0 <= z <= 10^7
 * 测试链接: http://poj.org/problem?id=2728
 *
 * 算法思路: 使用二分法求解 01 分数规划问题, 结合 Prim 算法求最小生成树进行可行性判断
 * 时间复杂度: O(n^2 * log(1/ ε)), 其中 ε 是精度要求
 * 空间复杂度: O(n^2)
 *
 * 01 分数规划的数学原理:
 * 我们需要最小化 R = (sum(cost_e)) / (sum(dist_e)), 其中 e 是生成树中的边
 * 对于给定的 L, 判断是否存在生成树使得 R < L
 * 等价于: sum(cost_e) < L * sum(dist_e)
 * 等价于: sum(cost_e - L * dist_e) < 0
 * 我们通过二分 L 的值, 使用 Prim 算法计算最小生成树的权值和来判断是否可行
 * 如果最小生成树的权值和 < 0, 则说明 L 可行, 可以尝试更小的值
 */

```

```

const int MAXN = 1001; // 最大村庄数量
const double sml = 1e-6; // 精度控制, 用于二分结束条件

// 村庄的坐标和海拔
int x[MAXN]; // x 坐标
int y[MAXN]; // y 坐标
int z[MAXN]; // 海拔高度

// 存储任意两村庄间的距离和花费
double dist[MAXN][MAXN]; // 欧式距离
double cost[MAXN][MAXN]; // 海拔差绝对值 (花费)

// Prim 算法所需的辅助数组
bool visit[MAXN]; // 标记村庄是否已加入生成树
double value[MAXN]; // 存储每个村庄到生成树的最小边权 (cost_e - x * dist_e)

```

```

int n; // 村庄数量

/**
 * 邻接矩阵结构下的 Prim 算法，从节点 1 出发计算最小生成树的权值和
 * 边权为 cost_e - x * dist_e，用于 01 分数规划的可行性判断
 *
 * @param x 当前尝试的比率值
 * @return 最小生成树的权值和
 */
double prim(double x) {
    // 初始化 visit 数组和 value 数组
    for (int i = 1; i <= n; i++) {
        visit[i] = false;
        value[i] = cost[1][i] - x * dist[1][i]; // 初始化为从节点 1 出发的边权
    }

    visit[1] = true; // 标记节点 1 已加入生成树
    double sum = 0; // 用于存储最小生成树的权值和

    // 最小生成树一定有 n-1 条边，需要进行 n-1 轮选择
    for (int i = 1; i <= n - 1; i++) {
        // 在未加入生成树的节点中，找到到生成树点集距离最小的点
        double minDist = 1e20;
        int next = 0;
        for (int j = 1; j <= n; j++) {
            if (!visit[j] && value[j] < minDist) {
                minDist = value[j];
                next = j;
            }
        }
    }

    // 将选中的边加入生成树，更新总和
    sum += minDist;
    visit[next] = true; // 标记新节点已加入生成树

    // 查看新加入的节点能否更新其他未加入节点到生成树的最小距离
    for (int j = 1; j <= n; j++) {
        if (!visit[j]) {
            double newValue = cost[next][j] - x * dist[next][j];
            if (value[j] > newValue) {
                value[j] = newValue;
            }
        }
    }
}

```

```

    }

}

return sum; // 返回最小生成树的权值和
}

/***
 * 主函数: 处理输入输出, 执行二分查找
 */
int main() {
    // 读取村庄数量
    while (scanf("%d", &n) != EOF && n != 0) {
        // 读取每个村庄的坐标和海拔
        for (int i = 1; i <= n; i++) {
            scanf("%d %d %d", &x[i], &y[i], &z[i]);
        }

        // 预处理计算任意两个村庄之间的距离和花费
        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= n; j++) {
                if (i != j) {
                    // 计算欧式距离
                    double dx = x[i] - x[j];
                    double dy = y[i] - y[j];
                    dist[i][j] = sqrt(dx * dx + dy * dy);

                    // 计算海拔差绝对值作为花费
                    cost[i][j] = abs(z[i] - z[j]);
                }
            }
        }
    }

    // 初始化二分查找的左右边界
    // 左边界为 0, 右边界可以根据数据范围估算
    double l = 0.0, r = 100.0; // 最大可能的比值, 根据数据范围设置
    double ans = 0.0;

    // 二分查找过程
    // 当左右边界的差大于精度要求时继续循环
    while (l < r && r - l >= sml) {
        double mid = (l + r) / 2.0;

        // 调用 Prim 算法计算当前比率下的最小生成树权值和
    }
}

```

```

// 如果权值和 <= 0, 说明可以找到更小的比值, 调整右边界
if (prim(mid) <= 0) {
    ans = mid;
    r = mid - sml;
} else {
    // 否则调整左边界
    l = mid + sml;
}
}

// 输出结果, 保留 3 位小数
printf("%.3f\n", ans);
}

return 0;
}

```

---

文件: Code03\_DesertKing.java

---

```

package class138;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.io.StreamTokenizer;

/**
 * 01 分数规划问题 - 最优比率生成树 (Desert King)
 * 题目来源: POJ 2728
 * 题目描述: 有 n 个村庄, 每个村庄由 (x, y, z) 表示, 其中 (x, y) 是二维地图中的位置, z 是海拔高度。
 * 任意两个村庄之间的距离是二维地图中的欧式距离, 修路花费是海拔差值的绝对值。
 * 要求将所有村庄连通, 使得总花费/总距离的比值最小, 结果保留小数点后 3 位。
 *
 * 数据范围:
 * 2 <= n <= 10^3
 * 0 <= x、y <= 10^4
 * 0 <= z <= 10^7
 * 测试链接: http://poj.org/problem?id=2728
 * 注: 提交代码时请将类名改为"Main", 以通过在线评测。

```

```

*
* 算法思路：使用二分法求解 01 分数规划问题，结合 Prim 算法求最小生成树进行可行性判断
* 时间复杂度： $O(n^2 * \log(1/\epsilon))$ ，其中  $\epsilon$  是精度要求
* 空间复杂度： $O(n^2)$ 
*
* 01 分数规划的数学原理：
* 我们需要最小化  $R = (\sum(cost_e)) / (\sum(dist_e))$ ，其中  $e$  是生成树中的边
* 对于给定的  $L$ ，判断是否存在生成树使得  $R < L$ 
* 等价于： $\sum(cost_e) < L * \sum(dist_e)$ 
* 等价于： $\sum(cost_e - L * dist_e) < 0$ 
* 我们通过二分  $L$  的值，使用 Prim 算法计算最小生成树的权值和来判断是否可行
* 如果最小生成树的权值和  $< 0$ ，则说明  $L$  可行，可以尝试更小的值
*/
public class Code03_DesertKing {

    /**
     * 常量定义
     */
    public static final int MAXN = 1001; // 最大村庄数量
    public static final double PRECISION = 1e-6; // 精度控制，用于二分结束条件

    /**
     * 数据存储数组
     */
    public static int[] x = new int[MAXN]; // 村庄的 x 坐标
    public static int[] y = new int[MAXN]; // 村庄的 y 坐标
    public static int[] z = new int[MAXN]; // 村庄的海拔高度
    public static double[][] dist = new double[MAXN][MAXN]; // 任意两村庄间的欧式距离
    public static double[][] cost = new double[MAXN][MAXN]; // 任意两村庄间的修路花费（海拔差绝对值）

    /**
     * Prim 算法所需的辅助数组
     */
    public static boolean[] visit = new boolean[MAXN]; // 标记村庄是否已加入生成树
    public static double[] value = new double[MAXN]; // 存储每个村庄到生成树的最小边权 ( $cost_e - L * dist_e$ )

    /**
     * 村庄数量
     */
    public static int n;
}

```

```

/**
 * 邻接矩阵结构下的 Prim 算法，从节点 1 出发计算最小生成树的权值和
 * 边权为 cost_e - L * dist_e，用于 01 分数规划的可行性判断
 *
 * @param L 当前尝试的比率值
 * @return 最小生成树的权值和
 */
public static double prim(double L) {
    // 初始化 visit 数组和 value 数组
    for (int i = 1; i <= n; i++) {
        visit[i] = false;
        value[i] = cost[1][i] - L * dist[1][i]; // 初始化为从节点 1 出发的边权
    }

    visit[1] = true; // 标记节点 1 已加入生成树
    double totalWeight = 0; // 用于存储最小生成树的权值和

    // 最小生成树一定有 n-1 条边，需要进行 n-1 轮选择
    for (int i = 1; i <= n - 1; i++) {
        // 在未加入生成树的节点中，找到到生成树点集距离最小的点
        double minEdge = Double.MAX_VALUE;
        int nextNode = 0;
        for (int j = 1; j <= n; j++) {
            if (!visit[j] && value[j] < minEdge) {
                minEdge = value[j];
                nextNode = j;
            }
        }
    }

    // 将选中的边加入生成树，更新总和
    totalWeight += minEdge;
    visit[nextNode] = true; // 标记新节点已加入生成树

    // 查看新加入的节点能否更新其他未加入节点到生成树的最小距离
    for (int j = 1; j <= n; j++) {
        if (!visit[j]) {
            double newEdgeValue = cost[nextNode][j] - L * dist[nextNode][j];
            if (value[j] > newEdgeValue) {
                value[j] = newEdgeValue;
            }
        }
    }
}

```

```
    return totalWeight; // 返回最小生成树的权值和
}

/***
 * 主函数：处理输入输出，执行二分查找
 *
 * @param args 命令行参数
 * @throws IOException 输入输出异常
 */
public static void main(String[] args) throws IOException {
    // 输入输出流设置，提高效率
    BufferedReader br = null;
    StreamTokenizer in = null;
    PrintWriter out = null;

    try {
        br = new BufferedReader(new InputStreamReader(System.in));
        in = new StreamTokenizer(br);
        out = new PrintWriter(new OutputStreamWriter(System.out));

        // 读取村庄数量
        in.nextToken();
        n = (int) in.nval;

        // 处理多组测试用例，直到输入 0
        while (n != 0) {
            // 读取每个村庄的坐标和海拔
            for (int i = 1; i <= n; i++) {
                in.nextToken();
                x[i] = (int) in.nval;
                in.nextToken();
                y[i] = (int) in.nval;
                in.nextToken();
                z[i] = (int) in.nval;
            }

            // 预处理计算任意两个村庄之间的距离和花费
            for (int i = 1; i <= n; i++) {
                for (int j = 1; j <= n; j++) {
                    if (i != j) {
                        // 计算欧式距离
                        double dx = x[i] - x[j];
```

```

        double dy = y[i] - y[j];
        dist[i][j] = Math.sqrt(dx * dx + dy * dy);

        // 计算海拔差绝对值作为花费
        cost[i][j] = Math.abs(z[i] - z[j]);
    }
}

// 初始化二分查找的左右边界
// 左边界为 0, 右边界可以根据数据范围估算
double left = 0.0, right = 100.0; // 最大可能的比值, 根据数据范围设置
double result = 0.0;

// 二分查找过程
// 当左右边界的差大于精度要求时继续循环
while (left < right && right - left >= PRECISION) {
    double mid = (left + right) / 2.0;

    // 调用 Prim 算法计算当前比率下的最小生成树权值和
    // 如果权值和 <= 0, 说明可以找到更小的比值, 调整右边界
    if (prim(mid) <= 0) {
        result = mid;
        right = mid - PRECISION;
    } else {
        // 否则调整左边界
        left = mid + PRECISION;
    }
}

// 输出结果, 保留 3 位小数
out.printf("%.3f\n", result);

// 读取下一组测试用例的村庄数量
in.nextToken();
n = (int) in.nval;
}

} finally {
    // 确保资源正确关闭
    if (out != null) {
        out.flush();
        out.close();
    }
}

```

```

        if (br != null) {
            try {
                br.close();
            } catch (IOException e) {
                // 忽略关闭时的异常
            }
        }
    }

}

```

}

=====

文件: Code03\_DesertKing.py

=====

```
# -*- coding: utf-8 -*-
```

"""

01 分数规划问题 - 最优比率生成树 (Desert King)

题目来源: POJ 2728

题目描述: 有 n 个村庄, 每个村庄由(x, y, z)表示, 其中(x, y)是二维地图中的位置, z 是海拔高度。任意两个村庄之间的距离是二维地图中的欧式距离, 修路花费是海拔差值的绝对值。

要求将所有村庄连通, 使得总花费/总距离的比值最小, 结果保留小数点后 3 位。

数据范围:

$2 \leq n \leq 10^3$

$0 \leq x, y \leq 10^4$

$0 \leq z \leq 10^7$

测试链接: <http://poj.org/problem?id=2728>

算法思路: 使用二分法求解 01 分数规划问题, 结合 Prim 算法求最小生成树进行可行性判断

时间复杂度:  $O(n^2 * \log(1/\epsilon))$ , 其中  $\epsilon$  是精度要求

空间复杂度:  $O(n^2)$

01 分数规划的数学原理:

我们需要最小化  $R = (\text{sum}(\text{cost}_e)) / (\text{sum}(\text{dist}_e))$ , 其中 e 是生成树中的边

对于给定的 L, 判断是否存在生成树使得  $R < L$

等价于:  $\text{sum}(\text{cost}_e) < L * \text{sum}(\text{dist}_e)$

等价于:  $\text{sum}(\text{cost}_e - L * \text{dist}_e) < 0$

我们通过二分 L 的值, 使用 Prim 算法计算最小生成树的权值和来判断是否可行

如果最小生成树的权值和 < 0, 则说明 L 可行, 可以尝试更小的值

```
"""
```

```
import sys
import math

# 常量定义
MAXN = 1001 # 最大村庄数量
PRECISION = 1e-6 # 精度控制, 用于二分结束条件

# 全局变量初始化
x = [0] * MAXN # 村庄的 x 坐标
y = [0] * MAXN # 村庄的 y 坐标
z = [0] * MAXN # 村庄的海拔高度

dist = [[0.0 for _ in range(MAXN)] for _ in range(MAXN)] # 任意两村庄间的欧式距离
cost = [[0.0 for _ in range(MAXN)] for _ in range(MAXN)] # 任意两村庄间的修路花费 (海拔差绝对值)
visit = [False] * MAXN # 标记村庄是否已加入生成树
value = [0.0] * MAXN # 存储每个村庄到生成树的最小边权 (cost_e - L * dist_e)

n = 0 # 村庄数量
```

```
def prim(L):
```

```
"""
```

```
邻接矩阵结构下的 Prim 算法, 从节点 1 出发计算最小生成树的权值和  
边权为 cost_e - L * dist_e, 用于 01 分数规划的可行性判断
```

参数:

L (float): 当前尝试的比率值

返回:

float: 最小生成树的权值和

```
"""
```

```
# 初始化 visit 数组和 value 数组
```

```
for i in range(1, n + 1):
```

```
    visit[i] = False
```

```
    value[i] = cost[1][i] - L * dist[1][i] # 初始化为从节点 1 出发的边权
```

```
visit[1] = True # 标记节点 1 已加入生成树
```

```
total_weight = 0.0 # 用于存储最小生成树的权值和
```

```
# 最小生成树一定有 n-1 条边, 需要进行 n-1 轮选择
```

```
for i in range(1, n):
```

```
# 在未加入生成树的节点中，找到到生成树点集距离最小的点
min_edge = float('inf')
next_node = 0
for j in range(1, n + 1):
    if not visit[j] and value[j] < min_edge:
        min_edge = value[j]
        next_node = j

# 将选中的边加入生成树，更新总和
total_weight += min_edge
visit[next_node] = True # 标记新节点已加入生成树

# 查看新加入的节点能否更新其他未加入节点到生成树的最小距离
for j in range(1, n + 1):
    if not visit[j]:
        new_edge_value = cost[next_node][j] - L * dist[next_node][j]
        if value[j] > new_edge_value:
            value[j] = new_edge_value

return total_weight # 返回最小生成树的权值和
```

```
def read_input_line():
    """
    读取输入行，处理可能的输入错误

    返回：
    list: 分割后的输入数据列表
    """

    EOFError: 当遇到文件结束符时
    """
    line = sys.stdin.readline()
    if not line: # 检查是否到达文件末尾
        raise EOFError("输入已结束")
    return line.strip().split()
```

```
def main():
    """
    主函数：处理输入输出，执行二分查找算法求解最优化生成树问题
    """
    global n

    try:
```

```

# 读取村庄数量
line = read_input_line()
n = int(line[0])

# 处理多组测试用例，直到输入 0
while n != 0:
    # 读取每个村庄的坐标和海拔
    for i in range(1, n + 1):
        line = read_input_line()
        x[i] = int(line[0])
        y[i] = int(line[1])
        z[i] = int(line[2])

    # 预处理计算任意两个村庄之间的距离和花费
    for i in range(1, n + 1):
        for j in range(1, n + 1):
            if i != j:
                # 计算欧式距离
                dx = x[i] - x[j]
                dy = y[i] - y[j]
                dist[i][j] = math.sqrt(dx * dx + dy * dy)

                # 计算海拔差绝对值作为花费
                cost[i][j] = abs(z[i] - z[j])

    # 初始化二分查找的左右边界
    # 左边界为 0，右边界可以根据数据范围估算
    left = 0.0
    right = 100.0 # 最大可能的比值，根据数据范围设置
    result = 0.0

    # 二分查找过程
    # 当左右边界的差大于精度要求时继续循环
    while left < right and right - left >= PRECISION:
        mid = (left + right) / 2.0

        # 调用 Prim 算法计算当前比率下的最小生成树权值和
        # 如果权值和 <= 0，说明可以找到更小的比值，调整右边界
        if prim(mid) <= 0:
            result = mid
            right = mid - PRECISION
        else:
            # 否则调整左边界

```

```

        left = mid + PRECISION

    # 输出结果，保留 3 位小数
    print("%.3f" % result)

    # 读取下一组测试用例的村庄数量
    line = read_input_line()
    n = int(line[0])

except EOFError:
    # 处理输入结束的情况
    pass
except (ValueError, IndexError) as e:
    # 处理输入格式错误
    print(f"输入格式错误: {e}", file=sys.stderr)

if __name__ == "__main__":
    main()

```

=====

文件: Code04\_MinimumAverageCircle.cpp

=====

```

/**
 * 最小圈问题 - 01 分数规划解法
 *
 * <h3>题目信息</h3>
 * <ul>
 *   <li><strong>题目来源</strong>: Luogu P3199</li>
 *   <li><strong>题目描述</strong>: 给定一个有向带权图，求所有环的平均值中最小的平均值。
 *   <li>环的平均值定义为：环中边的权值和 / 环中边的数量。</li>
 *   <li><strong>数据范围</strong>:
 *     <ul>
 *       <li>1 <= n <= 3000 (节点数) </li>
 *       <li>1 <= m <= 10000 (边数) </li>
 *       <li>-10^7 <= 边权 <= 10^7</li>
 *     </ul>
 *   </li>
 *   <li><strong>测试链接</strong>: <a href="https://www.luogu.com.cn/problem/P3199">Luogu P3199</a></li>
 * </ul>
 *
 * <h3>算法思路</h3>
 * <p>使用 01 分数规划 + 二分查找 + DFS 判负环的方法: </p>

```

- \* <ol>
- \* <li><strong>01 分数规划</strong>: 将环平均值最小化问题转化为判定性问题</li>
- \* <li><strong>二分查找</strong>: 在可能的平均值范围内进行二分</li>
- \* <li><strong>DFS 判负环</strong>: 通过 DFS 递归判断是否存在负环</li>
- \* </ol>
- \*
- \* <h3>数学原理</h3>
- \* <p>我们需要最小化  $R = (\sum w(e)) / |C|$ , 其中  $e \in \text{环 } C$ ,  $|C|$  是环的边数。</p>
- \* <p>对于给定的  $L$ , 判断是否存在环  $C$  使得  $R < L$ : </p>
- \* <ul>
 - \* <li>等价于:  $\sum w(e) < L * |C|$ </li>
- \* <li>等价于:  $\sum (w(e) - L) < 0$ </li>

- \* </ul>
- \* <p>这相当于将每条边的权值更新为  $(w(e) - L)$ , 然后判断图中是否存在负环。</p>
- \*
- \* <h3>复杂度分析</h3>
- \* <ul>
 - \* <li><strong>时间复杂度</strong>:  $O(\log(1/\epsilon) * n * m)$ , 其中  $\epsilon$  是精度要求 (本题中取  $1e-9$ )</li>
- \* <li><strong>空间复杂度</strong>:  $O(n + m)$ , 使用邻接表存储图结构</li>
- \* </ul>
- \*
- \* <h3>注意事项</h3>
- \* <ul>
 - \* <li>使用 DFS 判负环比 SPFA 更高效, 具有更强的剪枝能力</li>
- \* <li>需要处理精度问题, 避免浮点数比较误差</li>
- \* <li>图可能不连通, 需要从每个节点开始搜索</li>
- \* </ul>
- \*/

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

// 常量定义
const int MAXN = 3001;          // 最大节点数
const int MAXM = 10001;          // 最大边数
const double MAXE = 1e7;         // 最大边权绝对值
const double PRECISION = 1e-9;   // 精度控制

// 边结构体, 用于邻接表存储
```

```

struct Edge {
    int to;      // 目标节点
    double w;    // 边的权值
    Edge(int to, double w) : to(to), w(w) {}
};

// 图的邻接表存储（比链式前向星更适合 C++）
vector<vector<Edge>> graph;

// DFS 判负环需要的数据结构
double value[MAXN]; // 每个点的累积边权
bool path[MAXN];    // 每个点是否在当前递归路径上

// 全局变量，存储节点数和边数
int n, m;

/***
 * DFS 递归判断负环
 *
 * <p>这是 SPFA 算法的递归实现版本，具有更强的剪枝能力：</p>
 * <ul>
 *   <li>只有当可以松弛时才继续递归，避免不必要的搜索</li>
 *   <li>使用路径标记检测环的存在</li>
 *   <li>从每个节点开始搜索，确保覆盖所有连通分量</li>
 * </ul>
 *
 * @param u 当前访问的节点
 * @param L 当前尝试的比率值
 * @return 如果从当前节点出发存在负环，返回 true；否则返回 false
 */
bool dfs(int u, double L) {
    // 标记当前节点在递归路径上
    path[u] = true;

    // 遍历当前节点的所有出边
    for (const Edge& e : graph[u]) {
        int v = e.to;
        // 更新边权: w(e) - L
        double w = e.w - L;

        // 松弛操作：如果通过 u 到 v 可以使 value[v] 更小
        if (value[v] > value[u] + w) {
            value[v] = value[u] + w;
            if (dfs(v, w)) return true;
        }
    }
}

```

```

        // 如果 v 已经在当前递归路径上，说明找到了负环
        // 或者从 v 出发递归找到了负环
        if (path[v] || dfs(v, L)) {
            return true;
        }
    }

    // 回溯：标记当前节点不在递归路径上
    path[u] = false;
    return false;
}

/***
 * 检查函数：判断给定的平均值 L 是否可行
 *
 * <p>核心思想：将原图的边权更新为( $w(e) - L$ )，然后判断图中是否存在负环。</p>
 * <p>如果存在负环，说明存在平均值小于 L 的环；否则说明当前平均值过大。</p>
 *
 * @param L 当前尝试的平均值
 * @return 如果存在平均值小于 L 的环，返回 true；否则返回 false
 */
bool check(double L) {
    // 初始化距离数组和路径标记数组
    memset(value, 0, sizeof(value));
    memset(path, false, sizeof(path));

    // 从每个节点开始 DFS 搜索，确保覆盖所有连通分量
    for (int i = 1; i <= n; ++i) {
        if (dfs(i, L)) {
            return true;
        }
    }

    return false;
}

/***
 * 主函数：处理输入输出，执行二分查找算法
 *
 * <p>算法流程：</p>
 * <ol>

```

```

* <li>读取输入数据（节点数、边数、边权）</li>
* <li>初始化二分查找的左右边界</li>
* <li>进行二分查找，每次调用 check 函数判断当前平均值是否可行</li>
* <li>输出结果，保留 8 位小数</li>
* </ol>
*/
int main() {
    // 读取节点数和边数
    cin >> n >> m;

    // 初始化图的邻接表
    graph.resize(n + 1);

    // 读取每条边的信息并添加到图中
    for (int i = 0; i < m; ++i) {
        int u, v;
        double w;
        cin >> u >> v >> w;
        graph[u].emplace_back(v, w);
    }

    // 初始化二分查找的左右边界
    // 左边界为最小可能边权，右边界为最大可能边权
    double left = -MAXE;
    double right = MAXE;
    double result = 0.0;

    // 二分查找过程
    while (left < right && right - left >= PRECISION) {
        double mid = (left + right) / 2.0;

        if (check(mid)) {
            // 如果存在平均值小于 mid 的环，调整右边界
            right = mid - PRECISION;
        } else {
            // 否则记录当前结果并调整左边界
            result = mid;
            left = mid + PRECISION;
        }
    }

    // 输出结果，保留 8 位小数
    cout << fixed << setprecision(8) << result << endl;
}

```

```
    return 0;  
}
```

=====

文件: Code04\_MinimumAverageCircle.java

=====

```
package class138;
```

```
/**  
 * 最小圈问题 - 01 分数规划解法  
 *  
 * 题目描述: 给定一个有向带权图, 求所有环的平均值中最小的平均值  
 * 环的平均值定义为: 环中边的权值和 / 环中边的数量  
 *  
 * 数据范围:  
 * 1 <= n <= 3000  
 * 1 <= m <= 10000  
 * -10^7 <= 边权 <= 10^7  
 *  
 * 测试链接: https://www.luogu.com.cn/problem/P3199  
 *  
 * 算法思路: 01 分数规划 + 二分查找 + DFS 判负环  
 *  
 * 01 分数规划的数学原理:  
 * 我们需要找到环 C, 使得  $(\sum w(e))/|C|$  最小, 其中  $e \in C$ ,  $|C|$  是环的边数  
 *  
 * 对于给定的 L, 判断是否存在环 C, 使得  $(\sum w(e))/|C| < L$   
 * 等价于:  $\sum w(e) < L * |C|$   
 * 等价于:  $\sum (w(e) - L) < 0$   
 *  
 * 这相当于在原图每条边的权值减去 L 后, 判断图中是否存在负环  
 *  
 * 时间复杂度:  $O(\log(1/\epsilon) * n * m)$ , 其中  $\epsilon$  是精度要求  
 * 空间复杂度:  $O(n + m)$   
 */
```

```
import java.io.BufferedReader;  
import java.io.IOException;  
import java.io.InputStreamReader;  
import java.io.OutputStreamWriter;  
import java.io.PrintWriter;
```

```
import java.io.StreamTokenizer;
import java.util.Arrays;

public class Code04_MinimumAverageCircle {

    // 常量定义
    public static final int MAXN = 3001; // 最大节点数
    public static final int MAXM = 10001; // 最大边数
    public static final double MAXE = 1e7; // 最大边权绝对值
    public static final double PRECISION = 1e-9; // 精度控制

    // 链式前向星存储图
    public static int[] head = new int[MAXN]; // 每个节点的第一条边
    public static int[] next = new int[MAXM]; // 每条边的下一条边
    public static int[] to = new int[MAXM]; // 每条边的目标节点
    public static double[] weight = new double[MAXM]; // 每条边的权值
    public static int cnt; // 边的计数器

    // DFS 判断负环需要的数据结构
    public static double[] value = new double[MAXN]; // 每个点的累积边权
    public static boolean[] path = new boolean[MAXN]; // 每个点是否在当前递归路径上

    public static int n, m; // 节点数和边数

    /**
     * 初始化图的存储结构
     */
    public static void prepare() {
        cnt = 1;
        Arrays.fill(head, 1, n + 1, 0);
    }

    /**
     * 向图中添加一条有向边
     *
     * @param u 边的起点
     * @param v 边的终点
     * @param w 边的权值
     */
    public static void addEdge(int u, int v, double w) {
        next[cnt] = head[u];
        to[cnt] = v;
        weight[cnt] = w;
    }
}
```

```

head[u] = cnt++;
}

/***
 * 检查是否存在一个环，其平均值小于 L
 * 相当于在原图每条边减去 L 后，判断图中是否存在负环
 *
 * @param L 当前的比值候选值
 * @return 是否存在这样的环
 */

public static boolean check(double L) {
    Arrays.fill(value, 1, n + 1, 0);
    Arrays.fill(path, 1, n + 1, false);
    // 从超级源点 0 出发，访问所有节点
    return dfs(0, L);
}

/***
 * DFS 判断负环
 * 这实际上是 SPFA 算法的递归实现，具有很强的剪枝能力
 *
 * @param u 当前访问的节点
 * @param L 当前的比值候选值
 * @return 是否存在负环
 */

public static boolean dfs(int u, double L) {
    if (u == 0) {
        // 超级源点，访问所有节点
        for (int i = 1; i <= n; i++) {
            if (dfs(i, L)) {
                return true;
            }
        }
    } else {
        // 标记当前节点在递归路径上
        path[u] = true;

        // 遍历 u 的所有出边
        for (int e = head[u]; e != 0; e = next[e]) {
            int v = to[e];
            double w = weight[e] - L; // 边权减去 L

            // 只有当可以松弛时才继续递归，这是一个重要的剪枝
        }
    }
}

```

```

        if (value[v] > value[u] + w) {
            value[v] = value[u] + w; // 更新 v 的累积边权

            // 如果 v 已经在当前递归路径上，说明找到了负环
            // 或者从 v 出发递归找到了负环
            if (path[v] || dfs(v, L)) {
                return true;
            }
        }
    }

    // 回溯，标记当前节点不在递归路径上
    path[u] = false;
}

return false;
}

/***
 * 主函数
 *
 * @param args 命令行参数
 * @throws IOException 输入输出异常
 */
public static void main(String[] args) throws IOException {
    // 初始化输入输出流
    BufferedReader br = null;
    StreamTokenizer in = null;
    PrintWriter out = null;

    try {
        br = new BufferedReader(new InputStreamReader(System.in));
        in = new StreamTokenizer(br);
        out = new PrintWriter(new OutputStreamWriter(System.out));

        // 读取节点数和边数
        in.nextToken();
        n = (int) in.nval;
        in.nextToken();
        m = (int) in.nval;

        // 初始化图的存储结构
        prepare();
    }
}

```

```

// 读取每条边并添加到图中
for (int i = 1; i <= m; i++) {
    in.nextToken();
    int u = (int) in.nval;
    in.nextToken();
    int v = (int) in.nval;
    in.nextToken();
    double w = in.nval;
    addEdge(u, v, w);
}

// 初始化二分查找的左右边界
double left = -MAXE;
double right = MAXE;
double result = 0.0;

// 二分查找过程
while (left < right && right - left >= PRECISION) {
    double mid = (left + right) / 2.0;

    // 检查是否存在环的平均值小于 mid
    if (check(mid)) {
        // 如果存在，则调整右边界
        right = mid - PRECISION;
    } else {
        // 否则调整左边界，并记录当前结果
        result = mid;
        left = mid + PRECISION;
    }
}

// 输出结果，保留 8 位小数
out.printf("%.8f\n", result);
out.flush();

} finally {
    // 确保资源正确关闭
    if (out != null) {
        try {
            out.close();
        } catch (Exception e) {
            // 忽略关闭时的异常
        }
    }
}

```

```

    if (br != null) {
        try {
            br.close();
        } catch (IOException e) {
            // 忽略关闭时的异常
        }
    }
}
=====
```

文件: Code04\_MinimumAverageCircle.py

```

# 最小圈问题 - 01 分数规划解法
# 题目描述: 给定一个有向带权图, 求所有环的平均值中最小的平均值
# 环的平均值定义为: 环中边的权值和 / 环中边的数量
# 输入: n 个节点, m 条有向边及各自的权值
# 输出: 最小的环平均值, 保留 8 位小数
# 测试链接: https://www.luogu.com.cn/problem/P3199
```

, , ,

算法思路: 01 分数规划 + 二分查找 + DFS 判负环

01 分数规划的数学原理:

我们需要找到环 C, 使得  $(\sum w(e))/|C|$  最小, 其中  $e \in C$ ,  $|C|$  是环的边数

对于给定的 L, 判断是否存在环 C, 使得  $(\sum w(e))/|C| < L$

等价于:  $\sum w(e) < L * |C|$

等价于:  $\sum (w(e) - L) < 0$

这相当于在原图每条边的权值减去 L 后, 判断图中是否存在负环

通过二分 L 的值, 我们可以逐步逼近最小的环平均值

, , ,

```

import sys
from sys import stdin

sys.setrecursionlimit(1 << 25) # 设置递归深度, 避免 DFS 时栈溢出

# 常量定义
```

```

MAXN = 3001 # 最大节点数
MAXM = 10001 # 最大边数
MAXE = 1e7 # 最大边权绝对值
PRECISION = 1e-9 # 精度控制

# 全局变量
graph = [] # 邻接表存储图
value = [] # 每个点的累积边权
path = [] # 每个点是否在当前递归路径上
n, m = 0, 0 # 节点数和边数

def dfs(u, L):
    """
    DFS 判断负环
    在每条边减去 L 后，判断图中是否存在负环

    参数:
        u (int): 当前访问的节点
        L (float): 当前的比值候选值

    返回:
        bool: 是否存在负环
    """
    # 标记当前节点在递归路径上
    path[u] = True

    # 遍历 u 的所有出边
    for v, w in graph[u]:
        # 边权减去 L
        new_weight = w - L

        # 松弛操作：如果通过 u 到 v 可以使 value[v] 更小
        if value[v] > value[u] + new_weight:
            value[v] = value[u] + new_weight

        # 如果 v 已经在当前递归路径上，说明找到了一个负环
        # 或者从 v 出发找到了负环
        if path[v] or dfs(v, L):
            return True

    # 回溯，标记当前节点不在递归路径上
    path[u] = False
    return False

```

```

def check(L):
    """
    检查是否存在一个环，其平均值小于 L

    参数:
        L (float): 当前的比值候选值

    返回:
        bool: 是否存在这样的环
    """
    # 初始化每个节点的累积边权为 0
    for i in range(1, n + 1):
        value[i] = 0.0

    # 初始化每个节点不在任何递归路径上
    for i in range(1, n + 1):
        path[i] = False

    # 对每个节点进行 DFS，因为图可能不连通
    # 如果从任何一个节点出发能找到负环，则返回 True
    for i in range(1, n + 1):
        if dfs(i, L):
            return True

    return False

def main():
    """
    主函数: 读取输入，执行二分查找算法求解最小圈问题
    """

    global n, m, graph, value, path

    # 读取输入
    input_lines = [line.strip() for line in stdin.readlines()]
    ptr = 0

    # 读取节点数和边数
    while ptr < len(input_lines):
        if input_lines[ptr].strip(): # 跳过空行
            break
        ptr += 1

```

```

n, m = map(int, input_lines[ptr].split())
ptr += 1

# 初始化图的邻接表
graph = [[] for _ in range(n + 1)] # 节点编号从 1 开始

# 初始化全局数组
value = [0.0] * (n + 1)
path = [False] * (n + 1)

# 读取每条边
edges_read = 0
while ptr < len(input_lines) and edges_read < m:
    if not input_lines[ptr].strip(): # 跳过空行
        ptr += 1
        continue

    parts = input_lines[ptr].split()
    u = int(parts[0])
    v = int(parts[1])
    w = float(parts[2])
    graph[u].append((v, w))

    ptr += 1
    edges_read += 1

# 初始化二分查找的左右边界
# 左边界为最小可能的边权，右边界为最大可能的边权
left = -MAXE
right = MAXE
result = 0.0

# 二分查找过程
# 当左右边界的差大于精度要求时继续循环
while left < right and right - left >= PRECISION:
    mid = (left + right) / 2.0

    # 检查是否存在环的平均值小于 mid
    if check(mid):
        # 如果存在，则调整右边界
        right = mid - PRECISION
    else:
        # 否则调整左边界，并记录当前结果

```

```
result = mid
left = mid + PRECISION

# 输出结果, 保留 8 位小数
print("%.8f" % result)

if __name__ == "__main__":
    main()

=====
```

文件: Code05\_BestTeam.cpp

```
=====
/***
 * 最佳团体问题 - 树形背包 + 01 分数规划解法
 *
 * 题目描述:
 * 给定一棵树, 节点编号 0~n, 0 号节点是整棵树的头
 * 编号 1~n 的节点, 每个节点都有招募花费和战斗值, 0 号节点这两个值都是 0
 * 当招募某个节点时, 必须招募该节点及其所有祖先节点
 * 除了 0 号节点之外, 一共可以招募 k 个人, 希望让战斗值之和/招募花费之和的比值尽量大
 *
 * 数据范围:
 * 1 <= k <= n <= 2500
 * 0 <= 招募花费、战斗值 <= 10^4
 *
 * 测试链接: https://www.luogu.com.cn/problem/P4322
 *
 * 算法思路: 01 分数规划 + 树形背包 + DFN 序优化
 *
 * 01 分数规划的数学原理:
 * 我们需要找到 k 个节点的集合 S(不包括 0 号节点), 使得  $(\sum \text{strength}[i]) / (\sum \text{cost}[i])$  最大
 * 且 S 满足树形依赖关系(选子必选父)
 *
 * 对于给定的 L, 判断是否存在合法集合 S, 使得  $\sum \text{strength}[i] / \sum \text{cost}[i] > L$ 
 * 等价于:  $\sum (\text{strength}[i] - L * \text{cost}[i]) > 0$ 
 *
 * 我们使用树形背包来解决依赖关系的选择问题, 同时通过 DFN 序优化树形 DP
 *
 * 时间复杂度:  $O(\log(1/\epsilon) * n^2)$ , 其中  $\epsilon$  是精度要求
 * 空间复杂度:  $O(n^2)$ 
*/

```

```

#include <cstdio>
#include <cstring>
#include <algorithm>

// 常量定义
const int MAXN = 3001; // 最大节点数
const int LIMIT = 10000; // 最大可能的比值上限
const double NA = -1e9; // 无效解标记
const double PRECISION = 1e-6; // 精度控制

// 链式前向星存储树结构
int head[MAXN]; // 每个节点的第一条边
int next_edge[MAXN]; // 每条边的下一条边
int to[MAXN]; // 每条边的目标节点
int edgeCnt; // 边的计数器

// 节点属性
int cost[MAXN]; // 招募花费, 下标为节点原始编号
int strength[MAXN]; // 战斗值, 下标为节点原始编号

// DFN 序相关
int dfn[MAXN]; // dfn[a] = b, 表示原始 a 号节点的 dfn 编号为 b
int dfnCnt; // dfn 序计数
int size[MAXN]; // 子树大小, 下标为节点 dfn 编号

// 01 分数规划和树形 DP 相关
double value[MAXN]; // (战斗值 - x * 招募花费) 的结余
double dp[MAXN][MAXN]; // 树形 DP 数组

int k, n; // 需要招募的人数和总人数

/**
 * 初始化图的存储结构
 */
void prepare() {
    edgeCnt = 1;
    dfnCnt = 0;
    for (int i = 1; i <= n; i++) {
        head[i] = 0;
    }
}

/**

```

```

* 向树中添加一条有向边（父节点到子节点）
*
* @param u 父节点
* @param v 子节点
*/
void addEdge(int u, int v) {
    next_edge[edgeCnt] = head[u];
    to[edgeCnt] = v;
    head[u] = edgeCnt++;
}

/***
* 计算每个节点的 DFN 编号和子树大小
* 使用后序遍历，将树结构转换为线性结构，便于树形背包 DP
*
* @param u 当前节点
* @return 当前子树的大小
*/
int dfs(int u) {
    int i = ++dfnCnt;
    dfn[u] = i; // 记录当前节点的 DFN 编号
    size[i] = 1; // 初始子树大小为 1（包含自己）

    // 遍历所有子节点
    for (int e = head[u], v; e != 0; e = next_edge[e]) {
        v = to[e];
        size[i] += dfs(v); // 累加子节点的子树大小
    }

    return size[i];
}

/***
* 检查是否存在一种招募方式，使得比值大于 x
* 使用树形背包 DP 来求解最大结余和
*
* @param x 当前尝试的比值
* @return 是否存在这样的招募方式
*/
bool check(double x) {
    // 计算每个节点的结余值：战斗值 - x * 招募花费
    for (int i = 0; i <= n; i++) {
        value[dfn[i]] = static_cast<double>(strength[i]) - x * cost[i];
    }
}

```

```

}

// 初始化越界位置为无效解
for (int j = 1; j <= k; j++) {
    dp[dfnCnt + 1][j] = NA;
}

// 树形背包 DP 核心逻辑（基于 DFN 序的后序遍历）
for (int i = dfnCnt; i >= 2; i--) { // 从后往前处理，0 号节点的 DFN 是 1，跳过
    for (int j = 1; j <= k; j++) { // 枚举招募人数
        // 两种选择：
        // 1. 不选当前节点所在子树：dp[i + size[i]][j]
        // 2. 选当前节点：value[i] + dp[i + 1][j - 1]
        dp[i][j] = std::max(
            dp[i + size[i]][j], // 不选当前子树
            value[i] + dp[i + 1][j - 1] // 选当前节点，然后从子节点中选 j-1 个
        );
    }
}

// 原始的 0 号节点的 DFN 编号是 1，其他节点的 DFN 编号从 2 开始
// 0 号节点的战斗值和招募花费都是 0，我们需要从其他节点中招募 k 个
return dp[2][k] >= 0; // 如果最大结余和>=0，说明存在比值>=x 的方案
}

/***
 * 主函数
 *
 * @return 0 表示程序正常结束
 */
int main() {
    // 读取输入：需要招募的人数 k 和总人数 n
    scanf("%d%d", &k, &n);

    // 初始化树结构
    prepare();

    // 读取每个节点的信息：招募花费、战斗值和父节点
    for (int i = 1; i <= n; i++) {
        int parent;
        scanf("%d%d%d", &cost[i], &strength[i], &parent);
        addEdge(parent, i); // 添加父节点到子节点的边
    }
}

```

```

// 计算 DFN 序和子树大小
dfs(0);

// 初始化二分查找的左右边界
double left = 0.0;
double right = LIMIT;
double result = 0.0;

// 二分查找最优比值
while (left < right && right - left >= PRECISION) {
    double mid = (left + right) / 2.0;

    if (check(mid)) {
        // 如果存在比值>=mid 的方案，尝试更大的值
        result = mid;
        left = mid + PRECISION;
    } else {
        // 否则尝试更小的值
        right = mid - PRECISION;
    }
}

// 输出结果，保留 3 位小数
printf("%.3f\n", result);

return 0;
}

```

文件: Code05\_BestTeam.java

```

=====
package class138;

/**
 * 最佳团体问题 - 树形背包 + 01 分数规划解法
 *
 * 题目描述:
 * 给定一棵树，节点编号 0~n，0 号节点是整棵树的头
 * 编号 1~n 的节点，每个节点都有招募花费和战斗值，0 号节点这两个值都是 0
 * 当招募某个节点时，必须招募该节点及其所有祖先节点
 * 除了 0 号节点之外，一共可以招募 k 个人，希望让战斗值之和/招募花费之和的比值尽量大

```

```

*
* 数据范围:
* 1 <= k <= n <= 2500
* 0 <= 招募花费、战斗值 <= 10^4
*
* 测试链接: https://www.luogu.com.cn/problem/P4322
*
* 算法思路: 01 分数规划 + 树形背包 + DFN 序优化
*
* 01 分数规划的数学原理:
* 我们需要找到 k 个节点的集合 S(不包括 0 号节点), 使得  $(\sum \text{strength}[i]) / (\sum \text{cost}[i])$  最大
* 且 S 满足树形依赖关系(选子必选父)
*
* 对于给定的 L, 判断是否存在合法集合 S, 使得  $\sum \text{strength}[i] / \sum \text{cost}[i] > L$ 
* 等价于:  $\sum (\text{strength}[i] - L * \text{cost}[i]) > 0$ 
*
* 我们使用树形背包来解决依赖关系的选择问题, 同时通过 DFN 序优化树形 DP
*
* 时间复杂度:  $O(\log(1/\epsilon) * n^2)$ , 其中  $\epsilon$  是精度要求
* 空间复杂度:  $O(n^2)$ 
*/

```

```

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.io.StreamTokenizer;
import java.util.Arrays;

public class Code05_BestTeam {

    // 常量定义
    public static final int MAXN = 3001; // 最大节点数
    public static final int LIMIT = 10000; // 最大可能的比值上限
    public static final double NA = -1e9; // 无效解标记
    public static final double PRECISION = 1e-6; // 精度控制

    // 链式前向星存储树结构
    public static int[] head = new int[MAXN]; // 每个节点的第一条边
    public static int[] next = new int[MAXN]; // 每条边的下一条边
    public static int[] to = new int[MAXN]; // 每条边的目标节点
    public static int edgeCnt; // 边的计数器
}

```

```

// 节点属性
public static int[] cost = new int[MAXN];      // 招募花费, 下标为节点原始编号
public static int[] strength = new int[MAXN]; // 战斗值, 下标为节点原始编号

// DFN 序相关
public static int[] dfn = new int[MAXN]; // dfn[a] = b, 表示原始 a 号节点的 dfn 编号为 b
public static int dfnCnt; // dfn 序计数
public static int[] size = new int[MAXN]; // 子树大小, 下标为节点 dfn 编号

// 01 分数规划和树形 DP 相关
public static double[] value = new double[MAXN]; // (战斗值 - x * 招募花费) 的结余
public static double[][] dp = new double[MAXN][MAXN]; // 树形 DP 数组

public static int k, n; // 需要招募的人数和总人数

/**
 * 初始化图的存储结构
 */
public static void prepare() {
    edgeCnt = 1;
    dfnCnt = 0;
    Arrays.fill(head, 1, n + 1, 0);
}

/**
 * 向树中添加一条有向边 (父节点到子节点)
 *
 * @param u 父节点
 * @param v 子节点
 */
public static void addEdge(int u, int v) {
    next[edgeCnt] = head[u];
    to[edgeCnt] = v;
    head[u] = edgeCnt++;
}

/**
 * 计算每个节点的 DFN 编号和子树大小
 * 使用后序遍历, 将树结构转换为线性结构, 便于树形背包 DP
 *
 * @param u 当前节点
 * @return 当前子树的大小

```

```

*/
public static int dfs(int u) {
    int i = ++dfnCnt;
    dfn[u] = i; // 记录当前节点的 DFN 编号
    size[i] = 1; // 初始子树大小为 1 (包含自己)

    // 遍历所有子节点
    for (int e = head[u], v; e != 0; e = next[e]) {
        v = to[e];
        size[i] += dfs(v); // 累加子节点的子树大小
    }

    return size[i];
}

/***
 * 检查是否存在一种招募方式，使得比值大于 x
 * 使用树形背包 DP 来求解最大结余和
 *
 * @param x 当前尝试的比值
 * @return 是否存在这样的招募方式
 */
public static boolean check(double x) {
    // 计算每个节点的结余值：战斗值 - x * 招募花费
    for (int i = 0; i <= n; i++) {
        value[dfn[i]] = (double) strength[i] - x * cost[i];
    }

    // 初始化越界位置为无效解
    for (int j = 1; j <= k; j++) {
        dp[dfnCnt + 1][j] = NA;
    }

    // 树形背包 DP 核心逻辑（基于 DFN 序的后序遍历）
    for (int i = dfnCnt; i >= 2; i--) { // 从后往前处理，0 号节点的 DFN 是 1，跳过
        for (int j = 1; j <= k; j++) { // 枚举招募人数
            // 两种选择：
            // 1. 不选当前节点所在子树：dp[i + size[i]][j]
            // 2. 选当前节点：value[i] + dp[i + 1][j - 1]
            dp[i][j] = Math.max(
                dp[i + size[i]][j], // 不选当前子树
                value[i] + dp[i + 1][j - 1] // 选当前节点，然后从子节点中选 j-1 个
            );
        }
    }
}

```

```

    }

}

// 原始的 0 号节点的 DFN 编号是 1，其他节点的 DFN 编号从 2 开始
// 0 号节点的战斗值和招募花费都是 0，我们需要从其他节点中招募 k 个
return dp[2][k] >= 0; // 如果最大结余和>=0，说明存在比值>=x 的方案
}

/***
 * 主函数
 *
 * @param args 命令行参数
 * @throws IOException 输入输出异常
 */
public static void main(String[] args) throws IOException {
    // 初始化输入输出流
    BufferedReader br = null;
    StreamTokenizer in = null;
    PrintWriter out = null;

    try {
        br = new BufferedReader(new InputStreamReader(System.in));
        in = new StreamTokenizer(br);
        out = new PrintWriter(new OutputStreamWriter(System.out));

        // 读取输入：需要招募的人数 k 和总人数 n
        in.nextToken();
        k = (int) in.nval;
        in.nextToken();
        n = (int) in.nval;

        // 初始化树结构
        prepare();

        // 读取每个节点的信息：招募花费、战斗值和父节点
        for (int i = 1; i <= n; i++) {
            in.nextToken();
            cost[i] = (int) in.nval;
            in.nextToken();
            strength[i] = (int) in.nval;
            in.nextToken();
            int parent = (int) in.nval;
            addEdge(parent, i); // 添加父节点到子节点的边
        }
    } catch (IOException e) {
        e.printStackTrace();
    } finally {
        if (in != null) {
            try {
                in.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
        if (out != null) {
            out.close();
        }
    }
}

```

```
}

// 计算 DFN 序和子树大小
dfs(0);

// 初始化二分查找的左右边界
double left = 0.0;
double right = LIMIT;
double result = 0.0;

// 二分查找最优化值
while (left < right && right - left >= PRECISION) {
    double mid = (left + right) / 2.0;

    if (check(mid)) {
        // 如果存在比值>=mid 的方案，尝试更大的值
        result = mid;
        left = mid + PRECISION;
    } else {
        // 否则尝试更小的值
        right = mid - PRECISION;
    }
}

// 输出结果，保留 3 位小数
out.printf("%.3f\n", result);
out.flush();

} finally {
    // 确保资源正确关闭
    if (out != null) {
        try {
            out.close();
        } catch (Exception e) {
            // 忽略关闭时的异常
        }
    }
    if (br != null) {
        try {
            br.close();
        } catch (IOException e) {
            // 忽略关闭时的异常
        }
    }
}
```

```
    }
}
}
```

---

文件: Code05\_BestTeam.py

---

```
# -*- coding: utf-8 -*-
```

```
"""
```

最佳团体问题 - 树形背包 + 01 分数规划解法

题目描述:

给定一棵树, 节点编号  $0 \sim n$ , 0 号节点是整棵树的头

编号  $1 \sim n$  的节点, 每个节点都有招募花费和战斗值, 0 号节点这两个值都是 0

当招募某个节点时, 必须招募该节点及其所有祖先节点

除了 0 号节点之外, 一共可以招募  $k$  个人, 希望让战斗值之和/招募花费之和的比值尽量大

数据范围:

$1 \leq k \leq n \leq 2500$

$0 \leq$  招募花费、战斗值  $\leq 10^4$

测试链接: <https://www.luogu.com.cn/problem/P4322>

算法思路: 01 分数规划 + 树形背包 + DFN 序优化

01 分数规划的数学原理:

我们需要找到  $k$  个节点的集合  $S$  (不包括 0 号节点), 使得  $(\sum \text{strength}[i]) / (\sum \text{cost}[i])$  最大且  $S$  满足树形依赖关系 (选子必选父)

对于给定的  $L$ , 判断是否存在合法集合  $S$ , 使得  $\sum \text{strength}[i] / \sum \text{cost}[i] > L$

等价于:  $\sum (\text{strength}[i] - L * \text{cost}[i]) > 0$

我们使用树形背包来解决依赖关系的选择问题, 同时通过 DFN 序优化树形 DP

时间复杂度:  $O(\log(1/\epsilon) * n^2)$ , 其中  $\epsilon$  是精度要求

空间复杂度:  $O(n^2)$

```
"""
```

```
import sys
from typing import List, Tuple
```

```

# 常量定义
MAXN: int = 3001          # 最大节点数
LIMIT: int = 10000         # 最大可能的比值上限
NA: float = -1e9           # 无效解标记
PRECISION: float = 1e-6   # 精度控制

# 全局变量
# 链式前向星存储树结构
head: List[int] = [0] * MAXN      # 每个节点的第一条边
next_edge: List[int] = [0] * MAXN  # 每条边的下一条边
to: List[int] = [0] * MAXN        # 每条边的目标节点
edgeCnt: int = 0                  # 边的计数器

# 节点属性
cost: List[int] = [0] * MAXN      # 招募花费, 下标为节点原始编号
strength: List[int] = [0] * MAXN  # 战斗值, 下标为节点原始编号

# DFN 序相关
dfn: List[int] = [0] * MAXN      # dfn[a] = b, 表示原始 a 号节点的 dfn 编号为 b
dfnCnt: int = 0                  # dfn 序计数
size: List[int] = [0] * MAXN      # 子树大小, 下标为节点 dfn 编号

# 01 分数规划和树形 DP 相关
value: List[float] = [0.0] * MAXN # (战斗值 - x * 招募花费) 的结余
dp: List[List[float]] = [[0.0 for _ in range(MAXN)] for _ in range(MAXN)] # 树形 DP 数组

k: int = 0    # 需要招募的人数
n: int = 0    # 总人数

def prepare() -> None:
    """
    初始化图的存储结构
    重置边计数器、dfn 序计数器，并清空邻接表
    """
    global edgeCnt, dfnCnt
    edgeCnt = 1
    dfnCnt = 0
    for i in range(1, n + 1):
        head[i] = 0

def addEdge(u: int, v: int) -> None:
    """
    向树中添加一条有向边（父节点到子节点）
    """

```

Args:

u: 父节点

v: 子节点

"""

```
global edgeCnt
```

```
next_edge[edgeCnt] = head[u]
```

```
to[edgeCnt] = v
```

```
head[u] = edgeCnt
```

```
edgeCnt += 1
```

```
def dfs(u: int) -> int:
```

"""

计算每个节点的 DFN 编号和子树大小

使用后序遍历，将树结构转换为线性结构，便于树形背包 DP

Args:

u: 当前节点

Returns:

int: 当前子树的大小

"""

```
global dfnCnt
```

```
i: int = dfnCnt + 1
```

```
dfnCnt = i
```

```
dfn[u] = i      # 记录当前节点的 DFN 编号
```

```
size[i] = 1      # 初始子树大小为 1 (包含自己)
```

# 遍历所有子节点

```
e: int = head[u]
```

```
while e != 0:
```

```
    v: int = to[e]
```

```
    size[i] += dfs(v)  # 累加子节点的子树大小
```

```
    e = next_edge[e]
```

```
return size[i]
```

```
def check(x: float) -> bool:
```

"""

检查是否存在一种招募方式，使得比值大于 x

使用树形背包 DP 来求解最大结余和

Args:

x: 当前尝试的比值

Returns:

bool: 是否存在这样的招募方式

"""

# 计算每个节点的结余值: 战斗值 - x \* 招募花费

for i in range(n + 1):

    value[dfn[i]] = float(strength[i]) - x \* cost[i]

# 初始化越界位置为无效解

for j in range(1, k + 1):

    dp[dfnCnt + 1][j] = NA

# 树形背包 DP 核心逻辑 (基于 DFN 序的后序遍历)

for i in range(dfnCnt, 1, -1): # 从后往前处理, 0 号节点的 DFN 是 1, 跳过

    for j in range(1, k + 1): # 枚举招募人数

        # 两种选择:

        # 1. 不选当前节点所在子树: dp[i + size[i]][j]

        # 2. 选当前节点: value[i] + dp[i + 1][j - 1]

        dp[i][j] = max(

            dp[i + size[i]][j], # 不选当前子树

            value[i] + dp[i + 1][j - 1] # 选当前节点, 然后从子节点中选 j-1 个

)

# 原始的 0 号节点的 DFN 编号是 1, 其他节点的 DFN 编号从 2 开始

# 0 号节点的战斗值和招募花费都是 0, 我们需要从其他节点中招募 k 个

return dp[2][k] >= 0 # 如果最大结余和>=0, 说明存在比值>=x 的方案

def main() -> None:

"""

主函数

处理输入、构建树结构、执行算法并输出结果

"""

global k, n

try:

    # 读取输入: 需要招募的人数 k 和总人数 n

    line = sys.stdin.readline().split()

    k = int(line[0])

    n = int(line[1])

    # 初始化树结构

    prepare()

```
# 读取每个节点的信息：招募花费、战斗值和父节点
for i in range(1, n + 1):
    try:
        line = sys.stdin.readline().split()
        cost[i] = int(line[0])
        strength[i] = int(line[1])
        parent = int(line[2])
        addEdge(parent, i) # 添加父节点到子节点的边
    except (IndexError, ValueError):
        # 处理输入格式错误的情况
        print(f"输入数据格式错误，第{i+1}行数据不符合要求")
return

# 计算 DFN 序和子树大小
dfs(0)

# 初始化二分查找的左右边界
left: float = 0.0
right: float = LIMIT
result: float = 0.0

# 二分查找最优比值
while left < right and right - left >= PRECISION:
    mid: float = (left + right) / 2.0

    if check(mid):
        # 如果存在比值>=mid 的方案，尝试更大的值
        result = mid
        left = mid + PRECISION
    else:
        # 否则尝试更小的值
        right = mid - PRECISION

# 输出结果，保留 3 位小数
print("%.3f" % result)

except Exception as e:
    # 捕获所有异常并输出错误信息
    print(f"程序运行出错: {str(e)}")

if __name__ == "__main__":
    main()
```

文件: Code06\_SightseeingCows. cpp

```
#include <cstdio>
#include <cstring>
using namespace std;

// 观光奶牛 (Sightseeing Cows)
// 给定一个有向图, 每个点有一个点权 value[i], 每条边有一个边权 weight[i]
// 找到一个环, 使得环上点权和除以边权和最大
// 输出最大比率值, 保留两位小数
// 1 <= n <= 1000
// 1 <= m <= 5000
// 1 <= value[i], weight[i] <= 1000
// 测试链接 : https://www.luogu.com.cn/problem/P2868
// 测试链接 : http://poj.org/problem?id=3621

const int MAXN = 1001;
const int MAXM = 5001;
const double NA = -1e9;
const double sml = 1e-9;

// 链式前向星
int head[MAXN], next[MAXM], to[MAXM], cnt;
double weight[MAXM];

// 点权
int value[MAXN];

// dfs 判断正环, 每个点的累积点权
double dist[MAXN];

// dfs 判断正环, 每个点是否是递归路径上的点
bool path[MAXN];

int n, m;

void prepare() {
    cnt = 1;
    for (int i = 0; i < MAXN; i++) head[i] = 0;
}
```

```

void addEdge(int u, int v, double w) {
    next[cnt] = head[u];
    to[cnt] = v;
    weight[cnt] = w;
    head[u] = cnt++;
}

// 其实就是 spfa 的递归版，时间复杂度 O(n * m)
bool dfs(int u, double x) {
    if (u == 0) {
        // 认为 0 号点是超级源点，具有通往所有点的有向边
        for (int i = 1; i <= n; i++) {
            if (dfs(i, x)) {
                return true;
            }
        }
    } else {
        path[u] = true;
        for (int e = head[u]; e != 0; e = next[e]) {
            int v = to[e];
            double w = weight[e] - x; // 边权减去比率值
            // 只有 v 的累积点权变大，才会递归，非常强的剪枝
            // 如果递归路径回到 v，并且此时是 v 的累积点权更大的情况，说明遇到了正环
            // 或者后续递归找到了正环
            // 直接返回 true
            if (dist[v] < dist[u] + w) {
                dist[v] = dist[u] + w;
                if (path[v] || dfs(v, x)) {
                    return true;
                }
            }
        }
        path[u] = false;
    }
    return false;
}

// 所有边权都减去 x，返回图中是否存在正环
bool check(double x) {
    for (int i = 1; i <= n; i++) {
        dist[i] = 0;
        path[i] = false;
    }
}

```

```

    }
    return dfs(0, x);
}

int main() {
    scanf("%d%d", &n, &m);
    prepare();
    for (int i = 1; i <= n; i++) {
        scanf("%d", &value[i]);
    }
    for (int i = 1; i <= m; i++) {
        int u, v;
        double w;
        scanf("%d%d%lf", &u, &v, &w);
        addEdge(u, v, w);
    }
    double l = 0, r = 1000, x, ans = 0;
    while (l < r && r - l >= sml) {
        x = (l + r) / 2;
        // 如果存在正环，说明可以找到更大的比率值，向右二分
        // 如果不存在正环，说明当前比率值过大，向左二分
        if (check(x)) {
            ans = x;
            l = x + sml;
        } else {
            r = x - sml;
        }
    }
    printf("%.2f\n", ans);
    return 0;
}

```

=====

文件: Code06\_SightseeingCows.java

```

=====
package class138;

/**
 * 最优比率环问题 - 观光奶牛 (Sightseeing Cows)
 *
 * <h3>题目信息</h3>
 * <ul>

```

- \* <li><strong>题目来源</strong>: Luogu P2868, POJ 3621</li>
- \* <li><strong>题目描述</strong>: 给定一个有向图, 每个节点有点权  $value[i]$ , 每条边有边权  $weight[i]$ 。
- \* 要求找到一个环, 使得环上所有点权之和与边权之和的比值最大。</li>
- \* <li><strong>数据范围</strong>:
  - \* <ul><li> $1 \leq n \leq 1000$  (节点数)</li><li> $1 \leq m \leq 5000$  (边数)</li><li> $1 \leq value[i], weight[i] \leq 1000$ </li></ul>
- \* </li>
- \* <li><strong>测试链接</strong>:
  - \* <a href="https://www.luogu.com.cn/problem/P2868">Luogu P2868</a>,
  - \* <a href="http://poj.org/problem?id=3621">POJ 3621</a>
- \* </li>
- \* </ul>
- \*
- \* <h3>算法思路</h3>
- \* <p>使用 01 分数规划 + 二分查找 + DFS 判正环的方法: </p>
- \* <ol>
  - \* <li><strong>01 分数规划</strong>: 将比值最大化问题转化为判定性问题</li>
  - \* <li><strong>二分查找</strong>: 在可能的比值范围内进行二分</li>
  - \* <li><strong>DFS 判正环</strong>: 通过 DFS 递归判断是否存在正环</li>
- \* </ol>
- \*
- \* <h3>数学原理</h3>
- \* <p>我们需要最大化  $R = (\sum value[i]) / (\sum weight[e])$ , 其中  $i \in$  环  $C$ ,  $e \in$  环  $C$ </p>
- \* <p>对于给定的  $L$ , 判断是否存在环  $C$  使得  $R > L$ : </p>
- \* <ul>
  - \* <li>等价于:  $\sum value[i] > L * \sum weight[e]$ </li>
  - \* <li>等价于:  $\sum (value[i] - L * weight[e]) > 0$ </li>
- \* </ul>
- \* <p>这相当于将每条边的权值更新为  $(value[i] - L * weight[e])$ , 然后判断图中是否存在正环。</p>
- \*
- \* <h3>复杂度分析</h3>
- \* <ul>
  - \* <li><strong>时间复杂度</strong>:  $O(\log(1/\epsilon) * n * m)$ , 其中  $\epsilon$  是精度要求 (本题中取  $1e-9$ )</li>
  - \* <li><strong>空间复杂度</strong>:  $O(n + m)$ , 使用链式前向星存储图结构</li>
- \* </ul>
- \*
- \* <h3>注意事项</h3>
- \* <ul>

```
* <li>使用 DFS 判正环比 SPFA 更高效，具有更强的剪枝能力</li>
* <li>需要处理精度问题，避免浮点数比较误差</li>
* <li>使用超级源点技术，从所有节点开始搜索</li>
* </ul>
*/
```

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.io.StreamTokenizer;
import java.util.Arrays;

public class Code06_SightseeingCows {

    // 常量定义
    public static final int MAXN = 1001; // 最大节点数
    public static final int MAXM = 5001; // 最大边数

    // 足够小代表无效解
    public static final double NA = -1e9;

    // 最小精度控制，用于二分结束条件
    public static final double PRECISION = 1e-9;

    // 链式前向星存储图结构
    public static int[] head = new int[MAXN]; // 每个节点的第一条边
    public static int[] next = new int[MAXM]; // 每条边的下一条边
    public static int[] to = new int[MAXM]; // 每条边的目标节点
    public static double[] weight = new double[MAXM]; // 每条边的权值
    public static int cnt; // 边的计数器

    // 点权数组，存储每个节点的权值
    public static int[] value = new int[MAXN];

    // DFS 判正环需要的数据结构
    public static double[] dist = new double[MAXN]; // 每个点的累积边权
    public static boolean[] path = new boolean[MAXN]; // 每个点是否在当前递归路径上

    // 全局变量，存储节点数和边数
    public static int n, m;
```

```

/**
 * 初始化图的存储结构
 * 重置链式前向星的计数器，清空 head 数组
 */
public static void prepare() {
    cnt = 1;
    Arrays.fill(head, 1, n + 1, 0);
}

/**
 * 向图中添加一条有向边
 *
 * @param u 边的起点
 * @param v 边的终点
 * @param w 边的权值
 */
public static void addEdge(int u, int v, double w) {
    next[cnt] = head[u];
    to[cnt] = v;
    weight[cnt] = w;
    head[u] = cnt++;
}

/**
 * 检查函数：判断给定的比率值 x 是否可行
 *
 * <p>核心思想：将原图的边权更新为(value[i] - x * weight[e])，然后判断图中是否存在正环。</p>
 * <p>如果存在正环，说明存在比值大于 x 的环；否则说明当前比值过大。</p>
 *
 * @param x 当前尝试的比率值
 * @return 如果存在比值大于 x 的环，返回 true；否则返回 false
 */
public static boolean check(double x) {
    // 初始化距离数组和路径标记数组
    Arrays.fill(dist, 1, n + 1, 0);
    Arrays.fill(path, 1, n + 1, false);

    // 从超级源点 0 开始 DFS 搜索
    return dfs(0, x);
}

/**
 * DFS 递归判断正环

```

```

*
* <p>这是 SPFA 算法的递归实现版本，具有更强的剪枝能力：</p>
* <ul>
*   <li>只有当可以松弛时才继续递归，避免不必要的搜索</li>
*   <li>使用路径标记检测环的存在</li>
*   <li>从超级源点出发，确保访问所有连通分量</li>
* </ul>
*
* @param u 当前访问的节点
* @param x 当前尝试的比率值
* @return 如果从当前节点出发存在正环，返回 true；否则返回 false
*/
public static boolean dfs(int u, double x) {
    if (u == 0) {
        // 超级源点 0：具有通往所有节点的虚拟边
        // 从每个节点开始 DFS 搜索，确保覆盖所有连通分量
        for (int i = 1; i <= n; i++) {
            if (dfs(i, x)) {
                return true;
            }
        }
    } else {
        // 标记当前节点在递归路径上
        path[u] = true;

        // 遍历当前节点的所有出边
        for (int e = head[u]; e != 0; e = next[e]) {
            int v = to[e];
            // 更新边权: value[i] - x * weight[e]
            double w = value[v] - x * weight[e];

            // 只有当可以松弛时才继续递归（强剪枝）
            if (dist[v] < dist[u] + w) {
                dist[v] = dist[u] + w; // 更新累积边权

                // 如果 v 已经在当前递归路径上，说明找到了正环
                // 或者从 v 出发递归找到了正环
                if (path[v] || dfs(v, x)) {
                    return true;
                }
            }
        }
    }
}

```

```

        // 回溯：标记当前节点不在递归路径上
        path[u] = false;
    }
    return false;
}

/**
 * 主函数：处理输入输出，执行二分查找算法
 *
 * <p>算法流程：</p>
 * <ol>
 *   <li>读取输入数据（节点数、边数、点权、边权）</li>
 *   <li>初始化二分查找的左右边界</li>
 *   <li>进行二分查找，每次调用 check 函数判断当前比率是否可行</li>
 *   <li>输出结果，保留两位小数</li>
 * </ol>
 *
 * @param args 命令行参数
 * @throws IOException 输入输出异常
 */
public static void main(String[] args) throws IOException {
    // 初始化输入输出流
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    StreamTokenizer in = new StreamTokenizer(br);
    PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));

    try {
        // 读取节点数和边数
        in.nextToken();
        n = (int) in.nval;
        in.nextToken();
        m = (int) in.nval;

        // 初始化图的存储结构
        prepare();

        // 读取每个节点的点权
        for (int i = 1; i <= n; i++) {
            in.nextToken();
            value[i] = (int) in.nval;
        }

        // 读取每条边的信息并添加到图中
    }
}

```

```
for (int i = 1; i <= m; i++) {
    in.nextToken();
    int u = (int) in.nval;
    in.nextToken();
    int v = (int) in.nval;
    in.nextToken();
    double w = in.nval;
    addEdge(u, v, w);
}

// 初始化二分查找的左右边界
// 左边界为 0，右边界为最大可能比值（根据数据范围估算）
double left = 0.0;
double right = 1000.0;
double result = 0.0;

// 二分查找过程
while (left < right && right - left >= PRECISION) {
    double mid = (left + right) / 2.0;

    if (check(mid)) {
        // 如果存在比值大于 mid 的环，记录当前结果并尝试更大的值
        result = mid;
        left = mid + PRECISION;
    } else {
        // 否则尝试更小的值
        right = mid - PRECISION;
    }
}

// 输出结果，保留两位小数
out.printf("%.2f\n", result);
out.flush();

} finally {
    // 确保资源正确关闭
    try {
        out.close();
        br.close();
    } catch (Exception e) {
        // 忽略关闭时的异常
    }
}
```

```
    }  
}
```

```
=====文件: Code06_SightseeingCows. py=====
```

```
# -*- coding: utf-8 -*-  
  
# 观光奶牛 (Sightseeing Cows)  
# 给定一个有向图, 每个点有一个点权 value[i], 每条边有一个边权 weight[i]  
# 找到一个环, 使得环上点权和除以边权和最大  
# 输出最大比率值, 保留两位小数  
# 1 <= n <= 1000  
# 1 <= m <= 5000  
# 1 <= value[i], weight[i] <= 1000  
# 测试链接 : https://www.luogu.com.cn/problem/P2868  
# 测试链接 : http://poj.org/problem?id=3621
```

```
import sys  
from collections import deque
```

```
# 常量定义
```

```
MAXN = 1001  
MAXM = 5001  
NA = -1e9  
sm1 = 1e-9
```

```
# 全局变量
```

```
head = [0] * MAXN  
next_edge = [0] * MAXM  
to = [0] * MAXM  
weight = [0.0] * MAXM  
cnt = 0
```

```
# 点权
```

```
value = [0] * MAXN
```

```
# dfs 判断正环, 每个点的累积点权
```

```
dist = [0.0] * MAXN
```

```
# dfs 判断正环, 每个点是否是递归路径上的点
```

```
path = [False] * MAXN
```

```

n = 0
m = 0

def prepare():
    """初始化链式前向星结构"""
    global cnt
    cnt = 1
    for i in range(n + 1):
        head[i] = 0

def addEdge(u, v, w):
    """添加边到链式前向星结构"""
    global cnt
    next_edge[cnt] = head[u]
    to[cnt] = v
    weight[cnt] = w
    head[u] = cnt
    cnt += 1

def dfs(u, x):
    """DFS 判断是否存在正环"""
    if u == 0:
        # 认为 0 号点是超级源点，具有通往所有点的有向边
        for i in range(1, n + 1):
            if dfs(i, x):
                return True
    else:
        path[u] = True
        e = head[u]
        while e != 0:
            v = to[e]
            w = weight[e] - x # 边权减去比率值
            # 只有 v 的累积点权变大，才会递归，非常强的剪枝
            # 如果递归路径回到 v，并且此时是 v 的累积点权更大的情况，说明遇到了正环
            # 或者后续递归找到了正环，直接返回 True
            if dist[v] < dist[u] + w:
                dist[v] = dist[u] + w
                if path[v] or dfs(v, x):
                    return True
            e = next_edge[e]
        path[u] = False
    return False

```

```

def check(x):
    """检查是否存在比率值为 x 的正环"""
    # 初始化距离和路径数组
    for i in range(1, n + 1):
        dist[i] = 0
        path[i] = False
    return dfs(0, x)

def main():
    """主函数"""
    global n, m

    # 读取输入
    line = sys.stdin.readline().split()
    n = int(line[0])
    m = int(line[1])

    prepare()

    # 读取点权
    line = sys.stdin.readline().split()
    for i in range(1, n + 1):
        value[i] = int(line[i - 1])

    # 读取边信息
    for i in range(1, m + 1):
        line = sys.stdin.readline().split()
        u = int(line[0])
        v = int(line[1])
        w = float(line[2])
        addEdge(u, v, w)

    # 二分答案
    l = 0.0
    r = 1000.0
    ans = 0.0

    while l < r and r - l >= sml:
        x = (l + r) / 2
        # 如果存在正环，说明可以找到更大的比率值，向右二分
        # 如果不存在正环，说明当前比率值过大，向左二分
        if check(x):

```

```

ans = x
l = x + sml
else:
    r = x - sml

print("%.2f" % ans)

if __name__ == "__main__":
    main()
=====
```

文件: Code07\_MinimumDensityPath.cpp

```
=====
```

```

#include <cstdio>
#include <cstring>
using namespace std;

// 最小密度路径
// 给定一个有向图，每条边有两个权值 a[i] 和 b[i]
// 定义路径密度为路径上所有 a[i] 的和除以所有 b[i] 的和
// 求所有简单路径中密度最小的值
// 1 <= n <= 50
// 1 <= m <= 300
// 0 <= a[i], b[i] <= 100000
// b[i] > 0
// 测试链接 : https://www.luogu.com.cn/problem/P1730
```

```

const int MAXN = 51;
const int MAXM = 301;
const double INF = 1e20;
const double sml = 1e-9;
```

```

// 链式前向星
int head[MAXN];
int next_edge[MAXM];
int to[MAXM];
int a[MAXM];
int b[MAXM];
int cnt;

// floyd 数组, f[i][j][k] 表示从 i 到 j, 只经过编号不超过 k 的点的最小密度
// sumA[i][j][k] 表示从 i 到 j, 只经过编号不超过 k 的点的最小密度路径的 a 值和
```

```

// sumB[i][j][k]表示从 i 到 j, 只经过编号不超过 k 的点的最小密度路径的 b 值和
double f[MAXN][MAXN][MAXN];
double sumA[MAXN][MAXN][MAXN];
double sumB[MAXN][MAXN][MAXN];

int n, m;

void prepare() {
    cnt = 1;
    memset(head, 0, sizeof(head));

    // 初始化 floyd 数组
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++) {
            for (int k = 0; k <= n; k++) {
                f[i][j][k] = INF;
                sumA[i][j][k] = 0;
                sumB[i][j][k] = 0;
            }
        }
    }
}

// 初始化直接相连的边
for (int e = 1; e < cnt; e++) {
    int u = to[e ^ 1]; // 反向边
    int v = to[e];
    if (b[e] > 0) { // 避免除零错误
        f[u][v][0] = (double)a[e] / b[e];
        sumA[u][v][0] = a[e];
        sumB[u][v][0] = b[e];
    }
}
}

void addEdge(int u, int v, int aa, int bb) {
    next_edge[cnt] = head[u];
    to[cnt] = v;
    a[cnt] = aa;
    b[cnt] = bb;
    head[u] = cnt++;

    // 添加反向边, 便于查找
    next_edge[cnt] = head[v];
}

```

```

to[cnt] = u;
a[cnt] = aa;
b[cnt] = bb;
head[v] = cnt++;
}

// Floyd 变形，计算所有点对之间的最小密度路径
void floyd() {
    for (int k = 1; k <= n; k++) {
        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= n; j++) {
                // 通过点 k 转移
                double newSumA = sumA[i][k][k-1] + sumA[k][j][k-1];
                double newSumB = sumB[i][k][k-1] + sumB[k][j][k-1];

                if (newSumB > 0) { // 避免除零错误
                    double newDensity = newSumA / newSumB;

                    if (newDensity < f[i][j][k-1]) {
                        f[i][j][k] = newDensity;
                        sumA[i][j][k] = newSumA;
                        sumB[i][j][k] = newSumB;
                    } else {
                        f[i][j][k] = f[i][j][k-1];
                        sumA[i][j][k] = sumA[i][j][k-1];
                        sumB[i][j][k] = sumB[i][j][k-1];
                    }
                }
            }
        }
    }
}

int main() {
    scanf("%d%d", &n, &m);

    for (int i = 1; i <= m; i++) {
        int u, v, aa, bb;
        scanf("%d%d%d%d", &u, &v, &aa, &bb);
        addEdge(u, v, aa, bb);
    }

    prepare();
}

```

```

floyd();

// 寻找最小密度
double ans = INF;
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
        if (f[i][j][n] < ans) {
            ans = f[i][j][n];
        }
    }
}

printf("%.10f\n", ans);

return 0;
}

```

---

文件: Code07\_MinimumDensityPath.java

---

```

package class138;

/**
 * 最小密度路径问题 - 01 分数规划解法
 *
 * <h3>题目信息</h3>
 * <ul>
 *   <li><strong>题目来源</strong>: Luogu P1730</li>
 *   <li><strong>题目描述</strong>: 给定一个有向图, 每条边有两个权值 a[i] 和 b[i]。
 *   定义路径密度为路径上所有 a[i] 的和除以所有 b[i] 的和。要求找到所有简单路径中密度最小的值。</li>
 *   <li><strong>数据范围</strong>:
 *     <ul>
 *       <li>1 <= n <= 50 (节点数) </li>
 *       <li>1 <= m <= 300 (边数) </li>
 *       <li>0 <= a[i], b[i] <= 100000</li>
 *       <li>b[i] > 0 (确保分母不为零) </li>
 *     </ul>
 *   </li>
 *   <li><strong>测试链接</strong>: <a href="https://www.luogu.com.cn/problem/P1730">Luogu P1730</a></li>
 * </ul>
 *

```

- \* <h3>算法思路</h3>
- \* <p>使用 Floyd 变形算法直接计算所有点对之间的最小密度路径: </p>
- \* <ol>
 - \* <li><strong>Floyd 变形</strong>: 在标准的 Floyd 算法基础上, 维护路径的 a 值和与 b 值和</li>
- \* <li><strong>动态规划状态</strong>:  $f[i][j][k]$  表示从 i 到 j, 只经过编号不超过 k 的点的最小密度</li>
- \* <li><strong>状态转移</strong>: 通过中间点 k 更新路径密度</li>
- \* </ol>
- \*
- \* <h3>数学原理</h3>
- \* <p>我们需要找到所有简单路径中密度最小的路径, 即最小化  $R = (\sum a[i]) / (\sum b[i])$ 。</p>
- \* <p>由于数据规模较小 ( $n \leq 50$ ), 可以直接使用 Floyd 算法计算所有点对之间的最小密度路径。</p>
- \*
- \* <h3>复杂度分析</h3>
- \* <ul>
 - \* <li><strong>时间复杂度</strong>:  $O(n^3)$ , 标准的 Floyd 算法复杂度</li>
- \* <li><strong>空间复杂度</strong>:  $O(n^3)$ , 需要存储三维 DP 数组</li>

- \* </ul>
- \*
- \* <h3>注意事项</h3>
- \* <ul>
 - \* <li>由于  $n$  较小, 可以直接使用  $O(n^3)$  的算法</li>
- \* <li>需要处理浮点数精度问题</li>
- \* <li>确保分母  $b[i]$  不为零</li>
- \* </ul>
- \*/

```

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.io.StreamTokenizer;
import java.util.Arrays;

public class Code07_MinimumDensityPath {

    // 常量定义
    public static final int MAXN = 51;    // 最大节点数
    public static final int MAXM = 301;   // 最大边数
    public static final double INF = 1e20; // 无穷大值, 用于初始化
    public static final double PRECISION = 1e-9; // 精度控制
}

```

```

// 链式前向星存储图结构

public static int[] head = new int[MAXN]; // 每个节点的第一条边
public static int[] next = new int[MAXM]; // 每条边的下一条边
public static int[] to = new int[MAXM]; // 每条边的目标节点
public static int[] a = new int[MAXM]; // 每条边的 a 权值
public static int[] b = new int[MAXM]; // 每条边的 b 权值
public static int cnt; // 边的计数器

/***
 * Floyd 变形算法的 DP 数组
 * f[i][j][k]: 从节点 i 到节点 j, 只经过编号不超过 k 的节点的最小密度
 * sumA[i][j][k]: 对应路径的 a 权值和
 * sumB[i][j][k]: 对应路径的 b 权值和
 */
public static double[][][] f = new double[MAXN][MAXN][MAXN];
public static double[][][] sumA = new double[MAXN][MAXN][MAXN];
public static double[][][] sumB = new double[MAXN][MAXN][MAXN];

// 全局变量, 存储节点数和边数
public static int n, m;

/***
 * 初始化图的存储结构和 Floyd 数组
 */
public static void prepare() {
    cnt = 1;
    Arrays.fill(head, 1, n + 1, 0);

    // 初始化 Floyd 数组为无穷大
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++) {
            for (int k = 0; k <= n; k++) {
                f[i][j][k] = INF;
                sumA[i][j][k] = 0;
                sumB[i][j][k] = 0;
            }
        }
    }
}

// 初始化直接相连的边
for (int e = 1; e < cnt; e++) {
    int u = to[e ^ 1]; // 反向边 (获取起点)
    int v = to[e]; // 正向边 (获取终点)
}

```

```

// 直接边的密度 = a[e] / b[e]
f[u][v][0] = (double) a[e] / b[e];
sumA[u][v][0] = a[e];
sumB[u][v][0] = b[e];
}

}

/***
* 向图中添加一条无向边（实际存储为两条有向边）
*
* @param u 边的起点
* @param v 边的终点
* @param aa 边的 a 权值
* @param bb 边的 b 权值
*/
public static void addEdge(int u, int v, int aa, int bb) {
    // 添加正向边: u -> v
    next[cnt] = head[u];
    to[cnt] = v;
    a[cnt] = aa;
    b[cnt] = bb;
    head[u] = cnt++;

    // 添加反向边: v -> u (便于查找)
    next[cnt] = head[v];
    to[cnt] = u;
    a[cnt] = aa;
    b[cnt] = bb;
    head[v] = cnt++;
}

/***
* Floyd 变形算法，计算所有点对之间的最小密度路径
*
* <p>算法流程：</p>
* <ol>
*   <li>外层循环枚举中间点 k</li>
*   <li>内层循环枚举起点 i 和终点 j</li>
*   <li>通过中间点 k 更新路径密度</li>
* </ol>
*/
public static void floyd() {
    for (int k = 1; k <= n; k++) {

```

```

        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= n; j++) {
                // 通过中间点 k 转移: 路径 i->k->j
                double newSumA = sumA[i][k][k-1] + sumA[k][j][k-1];
                double newSumB = sumB[i][k][k-1] + sumB[k][j][k-1];
                double newDensity = newSumA / newSumB;

                if (newDensity < f[i][j][k-1]) {
                    // 如果通过 k 的路径密度更小, 更新 DP 数组
                    f[i][j][k] = newDensity;
                    sumA[i][j][k] = newSumA;
                    sumB[i][j][k] = newSumB;
                } else {
                    // 否则保持原来的路径
                    f[i][j][k] = f[i][j][k-1];
                    sumA[i][j][k] = sumA[i][j][k-1];
                    sumB[i][j][k] = sumB[i][j][k-1];
                }
            }
        }
    }

    /**
     * 主函数: 处理输入输出, 执行 Floyd 算法
     *
     * <p>算法流程: </p>
     * <ol>
     *   <li>读取输入数据 (节点数、边数、边权) </li>
     *   <li>初始化图结构和 Floyd 数组</li>
     *   <li>运行 Floyd 算法计算所有点对的最小密度路径</li>
     *   <li>找到全局最小密度并输出</li>
     * </ol>
     *
     * @param args 命令行参数
     * @throws IOException 输入输出异常
     */
    public static void main(String[] args) throws IOException {
        // 初始化输入输出流
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        StreamTokenizer in = new StreamTokenizer(br);
        PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));

```

```
try {
    // 读取节点数和边数
    in.nextToken();
    n = (int) in.nval;
    in.nextToken();
    m = (int) in.nval;

    // 读取每条边的信息并添加到图中
    for (int i = 1; i <= m; i++) {
        in.nextToken();
        int u = (int) in.nval;
        in.nextToken();
        int v = (int) in.nval;
        in.nextToken();
        int aa = (int) in.nval;
        in.nextToken();
        int bb = (int) in.nval;
        addEdge(u, v, aa, bb);
    }

    // 初始化图结构和 Floyd 数组
    prepare();

    // 运行 Floyd 算法计算所有点对的最小密度路径
    floyd();

    // 寻找全局最小密度
    double minDensity = INF;
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++) {
            if (f[i][j][n] < minDensity) {
                minDensity = f[i][j][n];
            }
        }
    }

    // 输出结果，保留 10 位小数
    out.printf("%.10f\n", minDensity);
    out.flush();
}

} finally {
    // 确保资源正确关闭
    try {
```

```
        out.close();
        br.close();
    } catch (Exception e) {
        // 忽略关闭时的异常
    }
}
=====
```

文件: Code07\_MinimumDensityPath.py

```
# -*- coding: utf-8 -*-
# 最小密度路径
# 给定一个有向图, 每条边有两个权值 a[i] 和 b[i]
# 定义路径密度为路径上所有 a[i] 的和除以所有 b[i] 的和
# 求所有简单路径中密度最小的值
# 1 <= n <= 50
# 1 <= m <= 300
# 0 <= a[i], b[i] <= 100000
# b[i] > 0
# 测试链接 : https://www.luogu.com.cn/problem/P1730
```

```
import sys
```

```
# 常量定义
MAXN = 51
MAXM = 301
INF = 1e20
sm1 = 1e-9
```

```
# 全局变量
head = [0] * MAXN
next_edge = [0] * MAXM
to = [0] * MAXM
a = [0] * MAXM
b = [0] * MAXM
cnt = 0
```

```
# floyd 数组, f[i][j][k] 表示从 i 到 j, 只经过编号不超过 k 的点的最小密度
# sumA[i][j][k] 表示从 i 到 j, 只经过编号不超过 k 的点的最小密度路径的 a 值和
```

```

# sumB[i][j][k]表示从 i 到 j, 只经过编号不超过 k 的点的最小密度路径的 b 值和
f = [[[INF for k in range(MAXN)] for j in range(MAXN)] for i in range(MAXN)]
sumA = [[[0.0 for k in range(MAXN)] for j in range(MAXN)] for i in range(MAXN)]
sumB = [[[0.0 for k in range(MAXN)] for j in range(MAXN)] for i in range(MAXN)]

n = 0
m = 0

def prepare():
    """初始化链式前向星结构和 Floyd 数组"""
    global cnt

    cnt = 1
    for i in range(n + 1):
        head[i] = 0

    # 初始化 floyd 数组
    for i in range(1, n + 1):
        for j in range(1, n + 1):
            for k in range(n + 1):
                f[i][j][k] = INF
                sumA[i][j][k] = 0
                sumB[i][j][k] = 0

    # 初始化直接相连的边
    for e in range(1, cnt):
        u = to[e ^ 1]  # 反向边
        v = to[e]
        if b[e] > 0:  # 避免除零错误
            f[u][v][0] = float(a[e]) / b[e]
            sumA[u][v][0] = a[e]
            sumB[u][v][0] = b[e]

def addEdge(u, v, aa, bb):
    """添加边到链式前向星结构"""
    global cnt
    next_edge[cnt] = head[u]
    to[cnt] = v
    a[cnt] = aa
    b[cnt] = bb
    head[u] = cnt
    cnt += 1

```

```

# 添加反向边，便于查找
next_edge[cnt] = head[v]
to[cnt] = u
a[cnt] = aa
b[cnt] = bb
head[v] = cnt
cnt += 1

# Floyd 变形，计算所有点对之间的最小密度路径
def floyd():
    """Floyd 变形算法计算最小密度路径"""
    for k in range(1, n + 1):
        for i in range(1, n + 1):
            for j in range(1, n + 1):
                # 通过点 k 转移
                newSumA = sumA[i][k][k-1] + sumA[k][j][k-1]
                newSumB = sumB[i][k][k-1] + sumB[k][j][k-1]

                if newSumB > 0: # 避免除零错误
                    newDensity = newSumA / newSumB

                    if newDensity < f[i][j][k-1]:
                        f[i][j][k] = newDensity
                        sumA[i][j][k] = newSumA
                        sumB[i][j][k] = newSumB
                    else:
                        f[i][j][k] = f[i][j][k-1]
                        sumA[i][j][k] = sumA[i][j][k-1]
                        sumB[i][j][k] = sumB[i][j][k-1]

    def main():
        """主函数"""
        global n, m

        # 读取输入
        line = sys.stdin.readline().split()
        n = int(line[0])
        m = int(line[1])

        for i in range(1, m + 1):
            line = sys.stdin.readline().split()
            u = int(line[0])
            v = int(line[1])

```

```

aa = int(line[2])
bb = int(line[3])
addEdge(u, v, aa, bb)

prepare()
floyd()

# 寻找最小密度
ans = INF
for i in range(1, n + 1):
    for j in range(1, n + 1):
        if f[i][j][n] < ans:
            ans = f[i][j][n]

print("%.10f" % ans)

if __name__ == "__main__":
    main()

```

=====

文件: Code08\_DinkelbachExample.cpp

=====

```

#include <cstdio>
#include <cmath>
using namespace std;

// 使用 Dinkelbach 算法解决 01 分数规划问题
// 给定 n 个物品，每个物品有两个属性 a[i] 和 b[i]
// 选择一些物品使得选中物品的 a 值和与 b 值和的比值最大
// 1 <= n <= 100000
// 1 <= a[i], b[i] <= 100

const int MAXN = 100001;

// a[i] 表示选取 i 的收益
int a[MAXN];

// b[i] 表示选取 i 的代价
int b[MAXN];

// d[i] = a[i] - L * b[i]，其中 L 为当前比率值
double d[MAXN];

```

```

int n;

// 使用 Dinkelbach 算法求解 01 分数规划
double dinkelbach() {
    double L = 0; // 初始比率值
    double epsilon = 1e-9; // 精度要求

    while (true) {
        // 根据当前比率值 L 计算 d 数组
        for (int i = 1; i <= n; i++) {
            d[i] = a[i] - L * b[i];
        }

        // 贪心选择 d[i] > 0 的物品，使得 sum(d[i] * x[i]) 最大
        double sumD = 0;
        double sumA = 0;
        double sumB = 0;

        for (int i = 1; i <= n; i++) {
            if (d[i] > 0) { // 选择 d[i] > 0 的物品
                sumD += d[i];
                sumA += a[i];
                sumB += b[i];
            }
        }

        // 如果 sumD <= 0，说明已经找到最优解
        if (sumD <= 0) {
            return L;
        }

        // 更新比率值
        double newL = sumA / sumB;

        // 如果新旧比率值差小于精度要求，则停止迭代
        if (fabs(newL - L) < epsilon) {
            return newL;
        }

        L = newL;
    }
}

```

```

int main() {
    scanf("%d", &n);

    for (int i = 1; i <= n; i++) {
        scanf("%d", &a[i]);
    }

    for (int i = 1; i <= n; i++) {
        scanf("%d", &b[i]);
    }

    double result = dinkelbach();
    printf("%.6f\n", result);

    return 0;
}

```

文件: Code08\_DinkelbachExample.java

```

=====
package class138;

/**
 * Dinkelbach 算法示例 - 01 分数规划的高效迭代解法
 *
 * <h3>题目信息</h3>
 * <ul>
 *   <li><strong>题目描述</strong>: 给定 n 个物品，每个物品有两个属性 a[i] 和 b[i]。
 *   <li>选择一些物品使得选中物品的 a 值和与 b 值和的比值最大。</li>
 *   <li><strong>数据范围</strong>:
 *     <ul>
 *       <li>1 <= n <= 100000 (物品数量)</li>
 *       <li>1 <= a[i], b[i] <= 100</li>
 *     </ul>
 *   </li>
 * </ul>
 *
 * <h3>算法思路</h3>
 * <p>使用 Dinkelbach 算法求解 01 分数规划问题，相比二分法具有更快的收敛速度：</p>
 * <ol>
 *   <li><strong>迭代逼近</strong>: 通过迭代方式不断逼近最优解</li>

```

- \* <li><strong>贪心选择</strong>: 每次迭代选择  $d[i] > 0$  的物品</li>
- \* <li><strong>收敛判断</strong>: 根据精度要求判断是否收敛</li>
- \* </ol>
- \*
- \* <h3>数学原理</h3>
- \* <p>Dinkelbach 算法的核心思想: </p>
- \* <p>设当前比率为  $L$ , 计算  $d[i] = a[i] - L * b[i]$ , 选择所有  $d[i] > 0$  的物品,  
得到新的比率  $L' = (\sum a[i]) / (\sum b[i])$ , 重复此过程直到收敛。</p>
- \*
- \* <h3>复杂度分析</h3>
- \* <ul>
- \* <li><strong>时间复杂度</strong>:  $O(k * n)$ , 其中  $k$  是迭代次数, 通常为  $O(\log(1/\epsilon))$ </li>
- \* <li><strong>空间复杂度</strong>:  $O(n)$ </li>
- \* </ul>
- \*
- \* <h3>与二分法的对比</h3>
- \* <ul>
- \* <li><strong>优点</strong>: 收敛速度更快, 通常只需要几次迭代</li>
- \* <li><strong>缺点</strong>: 实现相对复杂, 需要处理迭代收敛</li>
- \* <li><strong>适用场景</strong>: 大规模数据, 对性能要求高的场景</li>
- \* </ul>

\*/

```

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.io.StreamTokenizer;
import java.util.Arrays;

public class Code08_DinkelbachExample {

    // 常量定义
    public static final int MAXN = 100001; // 最大物品数量
    public static final double EPSILON = 1e-9; // 精度要求

    // 物品属性数组
    public static int[] a = new int[MAXN]; // 收益值数组
    public static int[] b = new int[MAXN]; // 代价值数组

    // 结余值数组:  $d[i] = a[i] - L * b[i]$ 
    public static double[] d = new double[MAXN];
}

```

```

// 全局变量，存储物品数量
public static int n;

/**
 * Dinkelbach 算法求解 01 分数规划
 *
 * <p>算法流程：</p>
 * <ol>
 *   <li>初始化比率值  $L = 0$ </li>
 *   <li>循环迭代直到收敛：
 *     <ul>
 *       <li>计算每个物品的结余值  $d[i] = a[i] - L * b[i]$ </li>
 *       <li>贪心选择所有  $d[i] > 0$  的物品</li>
 *       <li>计算新的比率值  $L' = (\sum a[i]) / (\sum b[i])$ </li>
 *       <li>如果  $|L' - L| < \epsilon$  或  $\text{sumD} \leq 0$ ，则停止迭代</li>
 *     </ul>
 *   </li>
 * </ol>
 *
 * @return 最优比率值
 */
public static double dinkelbach() {
    double L = 0.0; // 初始比率值

    while (true) {
        // 根据当前比率值 L 计算每个物品的结余值
        for (int i = 1; i <= n; i++) {
            d[i] = a[i] - L * b[i];
        }

        // 贪心选择：选择所有结余值为正的物品
        double sumD = 0.0; // 结余值总和
        double sumA = 0.0; // 选中物品的 a 值和
        double sumB = 0.0; // 选中物品的 b 值和

        for (int i = 1; i <= n; i++) {
            if (d[i] > 0) {
                sumD += d[i];
                sumA += a[i];
                sumB += b[i];
            }
        }
    }
}

```

```

// 收敛判断：如果结余值总和<=0，说明已经找到最优解
if (sumD <= 0) {
    return L;
}

// 计算新的比率值
double newL = sumA / sumB;

// 收敛判断：如果新旧比率值差小于精度要求，则停止迭代
if (Math.abs(newL - L) < EPSILON) {
    return newL;
}

// 更新比率值，继续迭代
L = newL;
}

/**
 * 主函数：处理输入输出，执行 Dinkelbach 算法
 *
 * <p>算法流程：</p>
 * <ol>
 *   <li>读取输入数据（物品数量、每个物品的 a 值和 b 值）</li>
 *   <li>调用 Dinkelbach 算法求解最优化率</li>
 *   <li>输出结果，保留 6 位小数</li>
 * </ol>
 *
 * @param args 命令行参数
 * @throws IOException 输入输出异常
 */
public static void main(String[] args) throws IOException {
    // 初始化输入输出流
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    StreamTokenizer in = new StreamTokenizer(br);
    PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));

    try {
        // 读取物品数量
        in.nextToken();
        n = (int) in.nval;
    }

```

```

// 读取每个物品的 a 值（收益值）
for (int i = 1; i <= n; i++) {
    in.nextToken();
    a[i] = (int) in.nval;
}

// 读取每个物品的 b 值（代价值）
for (int i = 1; i <= n; i++) {
    in.nextToken();
    b[i] = (int) in.nval;
}

// 调用 Dinkelbach 算法求解最优比率
double result = dinkelbach();

// 输出结果，保留 6 位小数
out.printf("%.6f\n", result);
out.flush();

} finally {
    // 确保资源正确关闭
    try {
        out.close();
        br.close();
    } catch (Exception e) {
        // 忽略关闭时的异常
    }
}
}

```

文件: Code08\_DinkelbachExample.py

```

# -*- coding: utf-8 -*-

# 使用 Dinkelbach 算法解决 01 分数规划问题
# 给定 n 个物品，每个物品有两个属性 a[i] 和 b[i]
# 选择一些物品使得选中物品的 a 值和与 b 值和的比值最大
# 1 <= n <= 100000
# 1 <= a[i], b[i] <= 100

```

```
import sys

# 常量定义
MAXN = 100001

# a[i]表示选取 i 的收益
a = [0] * MAXN

# b[i]表示选取 i 的代价
b = [0] * MAXN

# d[i] = a[i] - L * b[i]， 其中 L 为当前比率值
d = [0.0] * MAXN

n = 0

# 使用 Dinkelbach 算法求解 01 分数规划
def dinkelbach():
    """使用 Dinkelbach 算法求解 01 分数规划"""
    L = 0.0 # 初始比率值
    epsilon = 1e-9 # 精度要求

    while True:
        # 根据当前比率值 L 计算 d 数组
        for i in range(1, n + 1):
            d[i] = a[i] - L * b[i]

        # 贪心选择 d[i] > 0 的物品，使得 sum(d[i] * x[i]) 最大
        sumD = 0.0
        sumA = 0.0
        sumB = 0.0

        for i in range(1, n + 1):
            if d[i] > 0: # 选择 d[i] > 0 的物品
                sumD += d[i]
                sumA += a[i]
                sumB += b[i]

        # 如果 sumD <= 0，说明已经找到最优解
        if sumD <= 0:
            return L

        # 更新比率值
```

```

newL = sumA / sumB if sumB > 0 else 0

# 如果新旧比率值差小于精度要求，则停止迭代
if abs(newL - L) < epsilon:
    return newL

L = newL

def main():
    """主函数"""
    global n

    # 读取输入
    line = sys.stdin.readline().split()
    n = int(line[0])

    line = sys.stdin.readline().split()
    for i in range(1, n + 1):
        a[i] = int(line[i - 1])

    line = sys.stdin.readline().split()
    for i in range(1, n + 1):
        b[i] = int(line[i - 1])

    result = dinkelbach()
    print("%.6f" % result)

if __name__ == "__main__":
    main()

```

=====

文件: Code09\_MaximumDensitySubgraph.cpp

=====

```

#include <cstdio>
#include <cstring>
using namespace std;

// 最大密度子图
// 给定一个无向图，找到一个子图使得其密度最大
// 密度定义为子图中边数除以点数
// 1 <= n <= 1000
// 0 <= m <= 10000

```

```
// 测试链接 : https://www.luogu.com.cn/problem/UVA1389
```

```
const int MAXN = 1001;
const int MAXM = 10001;
const double sml = 1e-9;
```

```
// 链式前向星
int head[MAXN];
int next_edge[MAXM * 2]; // 无向图, 边数翻倍
int to[MAXM * 2];
int cnt;
```

```
// 度数
int degree[MAXN];
```

```
// 超级源点和超级汇点
int S, T;
```

```
int n, m;
```

```
void prepare() {
    cnt = 1;
    memset(head, 0, sizeof(head));
    memset(degree, 0, sizeof(degree));
    S = 0; // 超级源点
    T = n + 1; // 超级汇点
}
```

```
void addEdge(int u, int v) {
    // 无向图添加双向边
    next_edge[cnt] = head[u];
    to[cnt] = v;
    head[u] = cnt++;

    next_edge[cnt] = head[v];
    to[cnt] = u;
    head[v] = cnt++;

    degree[u]++;
    degree[v]++;
}
```

```
// 检查是否存在密度大于 x 的子图
```

```

// 这需要构建网络流模型并求解最大流
// 由于网络流实现较为复杂，这里只给出框架
bool check(double x) {
    /*
     * 构造网络流模型：
     * 1. 每个点 i 拆成 i 和 i' 两个点
     * 2. S 向每个点 i 连容量为 m 的边
     * 3. 每个点 i 向 T 连容量为  $2*x + m - \text{degree}[i]$  的边
     * 4. 原图中的每条边 (i, j)，连接 i' 到 j' 和 j' 到 i'，容量为 1
     * 5. 每个点 i 连接到 i'，容量为无穷大
     *
     * 如果最大流 < m*n，则存在密度大于 x 的子图
    */
}

// 实际实现需要网络流算法，此处省略具体代码
// 返回示例值
return true;
}

int main() {
    while (scanf("%d%d", &n, &m) != EOF) {
        if (n == 0 && m == 0) break;

        prepare();

        for (int i = 1; i <= m; i++) {
            int u, v;
            scanf("%d%d", &u, &v);
            addEdge(u, v);
        }

        // 特殊情况：如果没有边，密度为 0
        if (m == 0) {
            printf("0\n");
            continue;
        }

        // 二分答案求解最大密度
        double l = 0, r = m, ans = 0;
        while (r - l >= 1e-6) {
            double x = (l + r) / 2;
            if (check(x)) {
                ans = x;
            }
        }
    }
}

```

```

    l = x + sml;
} else {
    r = x - sml;
}
}

printf("%.8f\n", ans);
}

return 0;
}
=====
```

文件: Code09\_MaximumDensitySubgraph.java

```

package class138;

// 最大密度子图
// 给定一个无向图，找到一个子图使得其密度最大
// 密度定义为子图中边数除以点数
// 1 <= n <= 1000
// 0 <= m <= 10000
// 测试链接 : https://www.luogu.com.cn/problem/UVA1389
```

```

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.io.StreamTokenizer;
import java.util.Arrays;
```

```
public class Code09_MaximumDensitySubgraph {
```

```
    public static int MAXN = 1001;
```

```
    public static int MAXM = 10001;
```

```
    // 最小精度
```

```
    public static double sml = 1e-9;
```

```
// 链式前向星
```

```
public static int[] head = new int[MAXN];
public static int[] next = new int[MAXM * 2]; // 无向图，边数翻倍
public static int[] to = new int[MAXM * 2];
public static int cnt;

// 度数
public static int[] degree = new int[MAXN];

// 超级源点和超级汇点
public static int S, T;

// 网络流相关变量
// 这里省略网络流的具体实现，因为涉及较多代码

public static int n, m;

public static void prepare() {
    cnt = 1;
    Arrays.fill(head, 1, n + 1, 0);
    Arrays.fill(degree, 1, n + 1, 0);
    S = 0; // 超级源点
    T = n + 1; // 超级汇点
}

public static void addEdge(int u, int v) {
    // 无向图添加双向边
    next[cnt] = head[u];
    to[cnt] = v;
    head[u] = cnt++;

    next[cnt] = head[v];
    to[cnt] = u;
    head[v] = cnt++;

    degree[u]++;
    degree[v]++;
}

// 检查是否存在密度大于 x 的子图
// 这需要构建网络流模型并求解最大流
// 由于网络流实现较为复杂，这里只给出框架
public static boolean check(double x) {
    /*

```

```

* 构造网络流模型:
* 1. 每个点 i 拆成 i 和 i' 两个点
* 2. S 向每个点 i 连容量为 m 的边
* 3. 每个点 i 向 T 连容量为  $2*x+m-degree[i]$  的边
* 4. 原图中的每条边 (i, j), 连接 i' 到 j' 和 j' 到 i', 容量为 1
* 5. 每个点 i 连接到 i', 容量为无穷大
*
* 如果最大流 < m*n, 则存在密度大于 x 的子图
*/

```

```

// 实际实现需要网络流算法, 此处省略具体代码
// 返回示例值
return true;
}

```

```

public static void main(String[] args) throws IOException {
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    StreamTokenizer in = new StreamTokenizer(br);
    PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));

    while (true) {
        if (in.nextToken() == StreamTokenizer.TT_EOF) break;
        n = (int) in.nval;
        in.nextToken();
        m = (int) in.nval;

        if (n == 0 && m == 0) break;

        prepare();

        for (int i = 1; i <= m; i++) {
            in.nextToken();
            int u = (int) in.nval;
            in.nextToken();
            int v = (int) in.nval;
            addEdge(u, v);
        }

        // 特殊情况: 如果没有边, 密度为 0
        if (m == 0) {
            out.println("0");
            continue;
        }
    }
}

```

```

// 二分答案求解最大密度
double l = 0, r = m, ans = 0;
while (r - l >= sml) {
    double x = (l + r) / 2;
    if (check(x)) {
        ans = x;
        l = x + sml;
    } else {
        r = x - sml;
    }
}

// 输出结果
out.printf("%.8f\n", ans);
}

out.flush();
out.close();
br.close();
}
}

```

=====

文件: Code09\_MaximumDensitySubgraph.py

```

=====

# -*- coding: utf-8 -*-

# 最大密度子图
# 给定一个无向图，找到一个子图使得其密度最大
# 密度定义为子图中边数除以点数
# 1 <= n <= 1000
# 0 <= m <= 10000
# 测试链接：https://www.luogu.com.cn/problem/UVA1389
```

```
import sys
```

```
# 常量定义
```

```
MAXN = 1001
```

```
MAXM = 10001
```

```
sml = 1e-9
```

```
# 全局变量
head = [0] * MAXN
next_edge = [0] * (MAXM * 2)  # 无向图，边数翻倍
to = [0] * (MAXM * 2)
cnt = 0
```

```
# 度数
degree = [0] * MAXN
```

```
# 超级源点和超级汇点
```

```
S = 0
T = 0
```

```
n = 0
m = 0
```

```
def prepare():
    """初始化图结构"""
    global cnt, S, T
    cnt = 1
    for i in range(n + 1):
        head[i] = 0
        degree[i] = 0
    S = 0      # 超级源点
    T = n + 1 # 超级汇点
```

```
def addEdge(u, v):
    """添加无向边"""
    global cnt
    # 无向图添加双向边
    next_edge[cnt] = head[u]
    to[cnt] = v
    head[u] = cnt
    cnt += 1
```

```
    next_edge[cnt] = head[v]
    to[cnt] = u
    head[v] = cnt
    cnt += 1

    degree[u] += 1
    degree[v] += 1
```

```

# 检查是否存在密度大于 x 的子图
# 这需要构建网络流模型并求解最大流
# 由于网络流实现较为复杂，这里只给出框架
def check(x):
    """
    检查是否存在密度大于 x 的子图
    构造网络流模型：
    1. 每个点 i 拆成 i 和 i' 两个点
    2. S 向每个点 i 连容量为 m 的边
    3. 每个点 i 向 T 连容量为  $2*x+m-degree[i]$  的边
    4. 原图中的每条边(i, j)，连接 i' 到 j' 和 j' 到 i'，容量为 1
    5. 每个点 i 连接到 i'，容量为无穷大

```

如果最大流  $< m*n$ ，则存在密度大于 x 的子图

实际实现需要网络流算法，此处省略具体代码，返回示例值

```

"""
return True

```

```

def main():
    """主函数"""
    global n, m

    # 由于是多组测试数据，这里简化处理
    for line in sys.stdin:
        values = line.split()
        if len(values) < 2:
            continue

        n = int(values[0])
        m = int(values[1])

        if n == 0 and m == 0:
            break

        prepare()

        for i in range(m):
            line = sys.stdin.readline()
            values = line.split()
            u = int(values[0])
            v = int(values[1])
            addEdge(u, v)

```

```

# 特殊情况：如果没有边，密度为 0
if m == 0:
    print("0")
    continue

# 二分答案求解最大密度
l = 0.0
r = float(m)
ans = 0.0
while r - l >= sml:
    x = (l + r) / 2
    if check(x):
        ans = x
        l = x + sml
    else:
        r = x - sml

# 输出结果
print("%.8f" % ans)

if __name__ == "__main__":
    main()

```

=====

文件: Other1.java

=====

```

package class138;

// 题目 1, 01 分数规划模版题，另一种二分的写法
// 思路是不变的，二分的写法多种多样
// 代码中打注释的位置，就是更简单的二分逻辑，其他代码没有变化
// 测试链接 : https://www.luogu.com.cn/problem/P10505
// 测试链接 : http://poj.org/problem?id=2976
// 提交以下的 code，提交时请把类名改成"Main"，可以通过所有测试用例

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.io.StreamTokenizer;
```

```
import java.util.Arrays;
import java.util.Comparator;

public class Other1 {

    public static int MAXN = 1001;

    public static double[][] arr = new double[MAXN][3];

    public static int n, k;

    public static boolean check(double x) {
        for (int i = 1; i <= n; i++) {
            arr[i][2] = arr[i][0] - x * arr[i][1];
        }
        Arrays.sort(arr, 1, n + 1, new MyComparator());
        double sum = 0;
        for (int i = 1; i <= k; i++) {
            sum += arr[i][2];
        }
        return sum >= 0;
    }

    public static class MyComparator implements Comparator<double[]> {

        @Override
        public int compare(double[] o1, double[] o2) {
            return o1[2] >= o2[2] ? -1 : 1;
        }
    }

    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        StreamTokenizer in = new StreamTokenizer(br);
        PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));
        in.nextToken();
        n = (int) in.nval;
        in.nextToken();
        k = (int) in.nval;
        while (n != 0 || k != 0) {
            k = n - k;
            for (int i = 1; i <= n; i++) {

```

```

        in.nextToken();
        arr[i][0] = in.nval;
    }
    for (int i = 1; i <= n; i++) {
        in.nextToken();
        arr[i][1] = in.nval;
    }
    double l = 0, r = 0, x;
    for (int i = 1; i <= n; i++) {
        r += arr[i][0];
    }
    // 二分进行 60 次，足够达到题目要求的精度
    // 二分完成后，l 就是答案
    for (int i = 1; i <= 60; i++) {
        x = (l + r) / 2;
        if (check(x)) {
            l = x;
        } else {
            r = x;
        }
    }
    out.println((int) (100 * (l + 0.005)));
    in.nextToken();
    n = (int) in.nval;
    in.nextToken();
    k = (int) in.nval;
}
out.flush();
out.close();
br.close();
}

}

```

}

=====

文件: Other2.java

```

=====
package class138;

// 题目 2，牛群的才艺展示，另一种二分的写法
// 思路是不变的，二分的写法多种多样
// 代码中打注释的位置，就是更简单的二分逻辑，其他代码没有变化

```

```
// 测试链接 : https://www.luogu.com.cn/problem/P4377
// 提交以下的 code, 提交时请把类名改成"Main", 可以通过所有测试用例

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.io.StreamTokenizer;
import java.util.Arrays;

public class Other2 {

    public static int MAXN = 251;

    public static int MAXW = 1001;

    public static double NA = -1e9;

    public static int[] weight = new int[MAXN];

    public static int[] talent = new int[MAXN];

    public static double[] value = new double[MAXN];

    public static double[] dp = new double[MAXW];

    public static int n, w;

    public static boolean check(double x) {
        for (int i = 1; i <= n; i++) {
            value[i] = (double) talent[i] - x * weight[i];
        }
        dp[0] = 0;
        Arrays.fill(dp, 1, w + 1, NA);
        for (int i = 1; i <= n; i++) {
            for (int p = w, j; p >= 0; p--) {
                j = (int) (p + weight[i]);
                if (j >= w) {
                    dp[w] = Math.max(dp[w], dp[p] + value[i]);
                } else {
                    dp[j] = Math.max(dp[j], dp[p] + value[i]);
                }
            }
        }
    }
}
```

```
        }
    }
    return dp[w] >= 0;
}

public static void main(String[] args) throws IOException {
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    StreamTokenizer in = new StreamTokenizer(br);
    PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));
    in.nextToken();
    n = (int) in.nval;
    in.nextToken();
    w = (int) in.nval;
    for (int i = 1; i <= n; i++) {
        in.nextToken();
        weight[i] = (int) in.nval;
        in.nextToken();
        talent[i] = (int) in.nval;
    }
    double l = 0, r = 0, x;
    for (int i = 1; i <= n; i++) {
        r += talent[i];
    }
    // 二分进行 60 次，足够达到题目要求的精度
    // 二分完成后，l 就是答案
    for (int i = 1; i <= 60; i++) {
        x = (l + r) / 2;
        if (check(x)) {
            l = x;
        } else {
            r = x;
        }
    }
    out.println((int) (l * 1000));
    out.flush();
    out.close();
    br.close();
}
}
```

}

=====

文件: Other3.java

```
=====
package class138;

// 题目 3, 最优比率生成树, 另一种二分的写法
// 思路是不变的, 二分的写法多种多样
// 代码中打注释的位置, 就是更简单的二分逻辑, 其他代码没有变化
// 测试链接 : http://poj.org/problem?id=2728
// 提交以下的 code, 提交时请把类名改成"Main", 可以通过所有测试用例

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.io.StreamTokenizer;

public class Other3 {

    public static int MAXN = 1001;

    public static int[] x = new int[MAXN];
    public static int[] y = new int[MAXN];
    public static int[] z = new int[MAXN];
    public static double[][] dist = new double[MAXN][MAXN];
    public static double[][] cost = new double[MAXN][MAXN];
    public static boolean[] visit = new boolean[MAXN];
    public static double[] value = new double[MAXN];
    public static int n;

    public static double prim(double x) {
        for (int i = 1; i <= n; i++) {
            visit[i] = false;
            value[i] = cost[1][i] - x * dist[1][i];
        }
        visit[1] = true;
```

```

double sum = 0;
for (int i = 1; i <= n - 1; i++) {
    double minDist = Double.MAX_VALUE;
    int next = 0;
    for (int j = 1; j <= n; j++) {
        if (!visit[j] && value[j] < minDist) {
            minDist = value[j];
            next = j;
        }
    }
    sum += minDist;
    visit[next] = true;
    for (int j = 1; j <= n; j++) {
        if (!visit[j] && value[j] > cost[next][j] - x * dist[next][j]) {
            value[j] = cost[next][j] - x * dist[next][j];
        }
    }
}
return sum;
}

public static void main(String[] args) throws IOException {
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    StreamTokenizer in = new StreamTokenizer(br);
    PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));
    in.nextToken();
    n = (int) in.nval;
    while (n != 0) {
        for (int i = 1; i <= n; i++) {
            in.nextToken();
            x[i] = (int) in.nval;
            in.nextToken();
            y[i] = (int) in.nval;
            in.nextToken();
            z[i] = (int) in.nval;
        }
        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= n; j++) {
                if (i != j) {
                    dist[i][j] = Math.sqrt((x[i] - x[j]) * (x[i] - x[j]) + (y[i] - y[j]) *
(y[i] - y[j]));
                    cost[i][j] = Math.abs(z[i] - z[j]);
                }
            }
        }
    }
}

```

```
        }

    }

    double l = 0, r = 100, x;
    // 二分进行 60 次，足够达到题目要求的精度
    // 二分完成后，l 就是答案
    for (int i = 1; i <= 60; i++) {
        x = (l + r) / 2;
        if (prim(x) <= 0) {
            r = x;
        } else {
            l = x;
        }
    }
    out.printf("%.3f\n", l);
    in.nextToken();
    n = (int) in.nval;
}

out.flush();
out.close();
br.close();
}
```

}

=====

文件: Other4.java

```
=====
package class138;

// 题目 4，最小圈，另一种二分的写法
// 思路是不变的，二分的写法多种多样
// 代码中打注释的位置，就是更简单的二分逻辑，其他代码没有变化
// 测试链接：https://www.luogu.com.cn/problem/P3199
// 提交以下的 code，提交时请把类名改成"Main"，可以通过所有测试用例
```

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.io.StreamTokenizer;
import java.util.Arrays;
```

```
public class Other4 {  
  
    public static int MAXN = 3001;  
  
    public static int MAXM = 10001;  
  
    public static double MAXE = 1e7;  
  
    public static int[] head = new int[MAXN];  
  
    public static int[] next = new int[MAXM];  
  
    public static int[] to = new int[MAXM];  
  
    public static double[] weight = new double[MAXM];  
  
    public static int cnt;  
  
    public static double[] value = new double[MAXN];  
  
    public static boolean[] path = new boolean[MAXN];  
  
    public static int n, m;  
  
    public static void prepare() {  
        cnt = 1;  
        Arrays.fill(head, 1, n + 1, 0);  
    }  
  
    public static void addEdge(int u, int v, double w) {  
        next[cnt] = head[u];  
        to[cnt] = v;  
        weight[cnt] = w;  
        head[u] = cnt++;  
    }  
  
    public static boolean check(double x) {  
        Arrays.fill(value, 1, n + 1, 0);  
        Arrays.fill(path, 1, n + 1, false);  
        return dfs(0, x);  
    }  
}
```

```

public static boolean dfs(int u, double x) {
    if (u == 0) {
        for (int i = 1; i <= n; i++) {
            if (dfs(i, x)) {
                return true;
            }
        }
    } else {
        path[u] = true;
        for (int e = head[u]; e != 0; e = next[e]) {
            int v = to[e];
            double w = weight[e] - x;
            if (value[v] > value[u] + w) {
                value[v] = value[u] + w;
                if (path[v] || dfs(v, x)) {
                    return true;
                }
            }
        }
        path[u] = false;
    }
    return false;
}

```

```

public static void main(String[] args) throws IOException {
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    StreamTokenizer in = new StreamTokenizer(br);
    PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));
    in.nextToken();
    n = (int) in.nval;
    in.nextToken();
    m = (int) in.nval;
    prepare();
    for (int i = 1; i <= m; i++) {
        in.nextToken();
        int u = (int) in.nval;
        in.nextToken();
        int v = (int) in.nval;
        in.nextToken();
        double w = in.nval;
        addEdge(u, v, w);
    }
    double l = -MAXE, r = MAXE, x;

```

```
// 二分进行 60 次，足够达到题目要求的精度
// 二分完成后，l 就是答案
for (int i = 1; i <= 60; i++) {
    x = (l + r) / 2;
    if (check(x)) {
        r = x;
    } else {
        l = x;
    }
}
out.printf("%.8f\n", l);
out.flush();
out.close();
br.close();
}
```

}

=====

文件: Other5.java

=====

```
package class138;

// 题目 5，最佳团体，另一种二分的写法
// 思路是不变的，二分的写法多种多样
// 代码中打注释的位置，就是更简单的二分逻辑，其他代码没有变化
// 测试链接：https://www.luogu.com.cn/problem/P4322
// 提交以下的 code，提交时请把类名改成"Main"，可以通过所有测试用例
```

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.io.StreamTokenizer;
import java.util.Arrays;
```

```
public class Other5 {
```

```
    public static int MAXN = 3001;
```

```
    public static int LIMIT = 10000;
```

```
public static double NA = -1e9;

public static int[] head = new int[MAXN];

public static int[] next = new int[MAXN];

public static int[] to = new int[MAXN];

public static int edgeCnt;

public static int[] cost = new int[MAXN];

public static int[] strength = new int[MAXN];

public static int[] dfn = new int[MAXN];

public static int dfnCnt;

public static double[] value = new double[MAXN];

public static int[] size = new int[MAXN];

public static double[][] dp = new double[MAXN][MAXN];

public static int k, n;

public static void prepare() {
    edgeCnt = 1;
    dfnCnt = 0;
    Arrays.fill(head, 1, n + 1, 0);
}

public static void addEdge(int u, int v) {
    next[edgeCnt] = head[u];
    to[edgeCnt] = v;
    head[u] = edgeCnt++;
}

public static int dfs(int u) {
    int i = ++dfnCnt;
    dfn[u] = i;
    size[i] = 1;
```

```

        for (int e = head[u], v; e != 0; e = next[e]) {
            v = to[e];
            size[i] += dfs(v);
        }
        return size[i];
    }

public static boolean check(double x) {
    for (int i = 0; i <= n; i++) {
        value[dfn[i]] = (double) strength[i] - x * cost[i];
    }
    for (int j = 1; j <= k; j++) {
        dp[dfnCnt + 1][j] = NA;
    }
    for (int i = dfnCnt; i >= 2; i--) {
        for (int j = 1; j <= k; j++) {
            dp[i][j] = Math.max(dp[i + size[i]][j], value[i] + dp[i + 1][j - 1]);
        }
    }
    return dp[2][k] >= 0;
}

public static void main(String[] args) throws IOException {
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    StreamTokenizer in = new StreamTokenizer(br);
    PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));
    in.nextToken();
    k = (int) in.nval;
    in.nextToken();
    n = (int) in.nval;
    prepare();
    for (int i = 1; i <= n; i++) {
        in.nextToken();
        cost[i] = (int) in.nval;
        in.nextToken();
        strength[i] = (int) in.nval;
        in.nextToken();
        addEdge((int) in.nval, i);
    }
    dfs(0);
    double l = 0, r = LIMIT, x;
    // 二分进行 60 次，足够达到题目要求的精度
    // 二分完成后，l 就是答案
}

```

```
for (int i = 1; i <= 60; i++) {  
    x = (l + r) / 2;  
    if (check(x)) {  
        l = x;  
    } else {  
        r = x;  
    }  
}  
out.printf("%.3f\n", l);  
out.flush();  
out.close();  
br.close();  
}  
}
```

---