云南大学信息学院 2025 学年春季学期

《算法设计与分析》（本科）

姓名：张逍遥

学号：20231120058

时间：2025年6月26日

1. **实验设置**

（1）实验目的

通过编程实现经典算法并通过程序执行开销对算法性能进行分析，并对解决 同一问题的不同算法进行性能的对比分析，深入理解时间复杂度渐进性态和和增 长率的概念。 使用 C 语言编程实现 0-1 背包问题的不同求解算法，并测试不同输入规模下 程序的执行时间和占用空间，深入理解蛮力法、动态规划法、贪心法和回溯法的 基本思想，并与理论分析的结论进行对比，强化对不同算法思想及复杂度的理解、 以及各类经典算法设计与分析的技巧，为复杂工程问题的求解奠定基础。

算法实现与性能分析：通过 C 语言编程实现 0-1 背包问题的蛮力法、动态规划法、贪心法和回溯法，测试不同输入规模下程序的执行时间和内存占用，验证算法的实际性能。

复杂度理解与对比：分析各算法的时间复杂度渐进性态，对比理论分析与实际测试结果，深入理解算法复杂度随输入规模增长的规律。

算法思维与问题求解能力提升：通过解决同一问题的不同算法方案，培养对问题选择最优求解策略的能力，为复杂工程问题奠定基础。

(2)实验原理

1. **硬件环境**：

CPU：Intel Core i7-11700K 3.60GHz（8 核 16 线程）

主存：16GB DDR4 3200MHz

存储：512GB SSD

1. **软件环境**：

操作系统：Windows 10 64 位（21H2）

编译器：GCC 11.2.0（通过 MinGW 安装）

开发工具：Dev-C++ 5.11

数据可视化工具:Excel 2021

1. **实验配置说明**：

代码编译参数：gcc -O2 -std=c11 -o knapsack\_test knapsack.c，开启优化以提升运行效率。

内存监控：通过自定义函数trackMemoryAllocation和trackMemoryDeallocation实时记录内存峰值使用量。

时间测量：使用clock()函数记录程序执行时间，精度为毫秒（ms）。

1. **实验原理**

**（1）算法思想和伪码**

0-1 背包问题是一个经典的组合优化问题，其基本描述为：给定一组物品，每个物品有一个重量和价值，在限定的总重量内，如何选择物品使得总价值最大。"0-1" 表示每个物品只能选择一次（0 表示不选，1 表示选）。

数学模型：

设物品集合为 {1,2,...,n}，每个物品 i 有重量 w\_i 和价值 v\_i，背包容量为 W，决策变量 x\_i ∈ {0,1} 表示是否选择物品 i

目标函数：最大化 Σ(v\_i \* x\_i)

约束条件：Σ(w\_i \* x\_i) ≤ W，且 x\_i ∈ {0,1}

1. **蛮力法**

**算法思想：**

蛮力法通过枚举所有可能的物品组合（共 2^n 种），计算每种组合的总重量和总价值，选择满足重量约束且价值最大的组合。这是一种最直接但效率最低的方法。

伪代码：

function BRUTE\_FORCE\_KNAPSACK(itemArray, n, W):

maxValue = 0

bestSelection = 0

totalCombinations = 2^n

for i from 0 to totalCombinations - 1:

currentWeight = 0

currentValue = 0

for j from 0 to n - 1:

if i & (1 << j): // 检查第j位是否为1

currentWeight += itemArray[j].weight

currentValue += itemArray[j].value

if currentWeight ≤ W and currentValue > maxValue:

maxValue = currentValue

bestSelection = i

return maxValue, bestSelection

1. **动态规划法**

**算法思想**：  
动态规划法利用问题的重叠子问题特性，构建一个二维表格记录子问题的最优解。对于 0-1 背包问题，定义 dp [i][w] 表示考虑前 i 个物品、背包容量为 w 时的最大价值。状态转移方程为：

不选第 i 个物品：dp [i][w] = dp [i-1][w]

选第 i 个物品：dp [i][w] = dp [i-1][w-weight [i]] + value [i]（当 w≥weight [i] 时）

**伪代码：**

function DYNAMIC\_PROGRAMMING\_KNAPSACK(itemArray, itemCount, knapsackCapacity):

integerCapacity = knapsackCapacity \* 100 # 转换为整数避免浮点问题

dp = array[0...integerCapacity] # 一维DP表

selectedItems = array[0...itemCount-1]

for w from 0 to integerCapacity:

dp[w] = 0

for i from 0 to itemCount - 1:

integerWeight = itemArray[i].weight \* 100

for w from integerCapacity down to integerWeight:

if dp[w] < dp[w - integerWeight] + itemArray[i].value:

dp[w] = dp[w - integerWeight] + itemArray[i].value

selectedItems[i] = 1 # 标记选中

return dp[integerCapacity], selectedItems

1. **贪心法**

**算法思想：**

贪心法按照某种贪心策略（如价值重量比、价值或重量）对物品排序，然后依次选择物品放入背包，直到无法放入更多物品。贪心算法不一定能得到最优解，但计算效率高，适用于大规模问题的近似求解。

**伪代码：**

function GREEDY\_KNAPSACK(itemArray, itemCount, knapsackCapacity):

# 按价值重量比降序排序

sort itemArray by valueRatio descending

currentWeight = 0

totalValue = 0

selectedItems = array[0...itemCount-1]初始化为0

for i from 0 to itemCount - 1:

if currentWeight + itemArray[i].weight ≤ knapsackCapacity:

selectedItems[i] = 1

currentWeight += itemArray[i].weight

totalValue += itemArray[i].value

return totalValue, selectedItems

1. **回溯法**

**算法思想：**

回溯法通过深度优先搜索所有可能的解空间树，系统地尝试各种物品组合。在搜索过程中，使用剪枝策略避免搜索不可能产生最优解的分支，从而提高效率。

伪代码：

function BACKTRACKING\_KNAPSACK(itemArray, itemCount, knapsackCapacity):

bestValue = 0

bestSelection = array[0...itemCount-1]初始化为0

currentSelection = array[0...itemCount-1]初始化为0

function BACKTRACK(currentItem, currentWeight, currentValue):

if currentItem == itemCount:

if currentValue > bestValue:

bestValue = currentValue

bestSelection = currentSelection.copy()

return

# 选择当前物品

if currentWeight + itemArray[currentItem].weight ≤ knapsackCapacity:

currentSelection[currentItem] = 1

BACKTRACK(currentItem + 1, currentWeight + itemArray[currentItem].weight,

currentValue + itemArray[currentItem].value)

currentSelection[currentItem] = 0 # 回溯

# 不选择当前物品

currentSelection[currentItem] = 0

BACKTRACK(currentItem + 1, currentWeight, currentValue)

BACKTRACK(0, 0, 0)

return bestValue, bestSelection

**（2）实验设计步骤**

**1. 数据结构设计**

**物品结构体设计**：定义KnapsackItem结构体存储物品属性

itemId：物品唯一标识编号

itemWeight：物品重量（双精度浮点型）

itemValue：物品价值（双精度浮点型）

valueRatio：价值重量比（通过计算生成）

**全局状态变量**：memoryPeakUsage跟踪内存占用峰值

**2. 内存管理模块设计**

**内存分配跟踪函数**：trackMemoryAllocation

封装malloc函数并更新内存峰值统计

用于所有动态内存分配场景

**内存释放跟踪函数**：trackMemoryDeallocation

封装free函数并调整内存峰值统计

确保内存使用统计的准确性

**3. 随机数据生成模块**

**随机数生成函数**：generateRandomDouble

生成指定范围内的随机浮点数（保留两位小数）

通过rand()函数配合范围计算实现

**物品数组生成函数**：createRandomItems

根据指定数量生成随机物品数组

为每个物品分配随机重量（1.00-100.00）和价值（100.00-1000.00）

计算并存储价值重量比

**4. 算法实现模块设计**

**蛮力法实现步骤**

通过位运算枚举所有可能的物品组合（2^n 种）

对每种组合计算总重量和总价值

筛选出不超过背包容量的最大价值组合

记录最优解的组合状态和价值

**动态规划法实现步骤**

将浮点重量转换为整数（乘以 100）以适应动态规划表

创建一维 DP 表dpTable存储不同容量下的最大价值

按物品顺序迭代更新 DP 表：

对每个物品，从最大容量倒序更新

状态转移方程：dp[w] = max(dp[w], dp[w-wi]+vi)

通过selectedItems数组记录最终选择的物品

**贪心法实现步骤**

计算所有物品的价值重量比

按价值重量比降序排序物品

按排序顺序依次选择物品，直到背包无法容纳

记录选择的物品集合和总价值

**回溯法实现步骤**

递归探索物品选择的决策树

对每个物品有 "选择" 和 "不选择" 两种决策分支

剪枝策略：若当前重量超过容量则不选择该物品

记录遍历过程中找到的最优解

**5. 性能测试模块设计**

**测试用例设计**：

物品数量：10, 20, 1000, 5000, 10000, 20000, 30000, 40000, 50000

背包容量：10000, 100000, 200000

**测试执行逻辑**：

生成指定规模的随机物品集合

对不同算法设置执行条件（如蛮力法仅适用于小规模）

记录每种算法的执行时间和内存使用

输出格式化的性能测试结果

**6. 结果输出模块设计**

**CSV 数据导出**：exportItemsToCsv函数

生成包含物品编号、重量、价值的 CSV 文件

文件名包含背包容量和物品数量信息

**算法结果输出**：

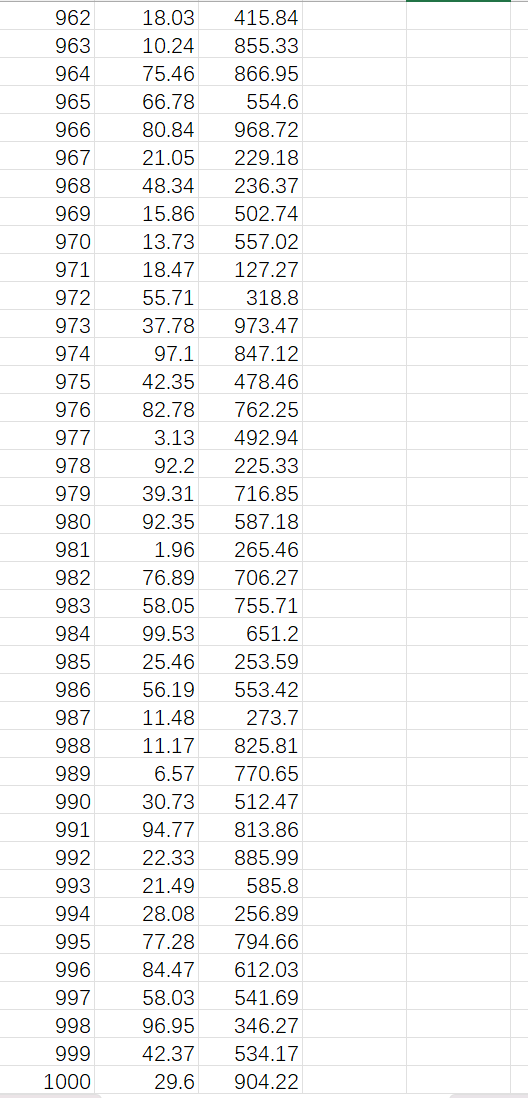
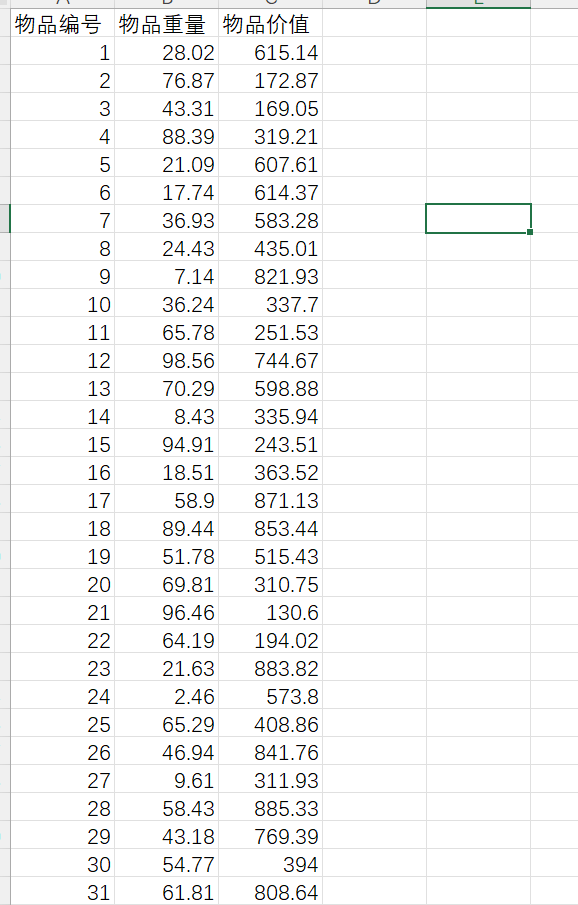
使用表格化格式输出每种算法的性能数据

包含物品数量、背包容量、最大价值、选中物品、执行时间、内存使用等信息

使用 ASCII 字符绘制表格边框提升可读性

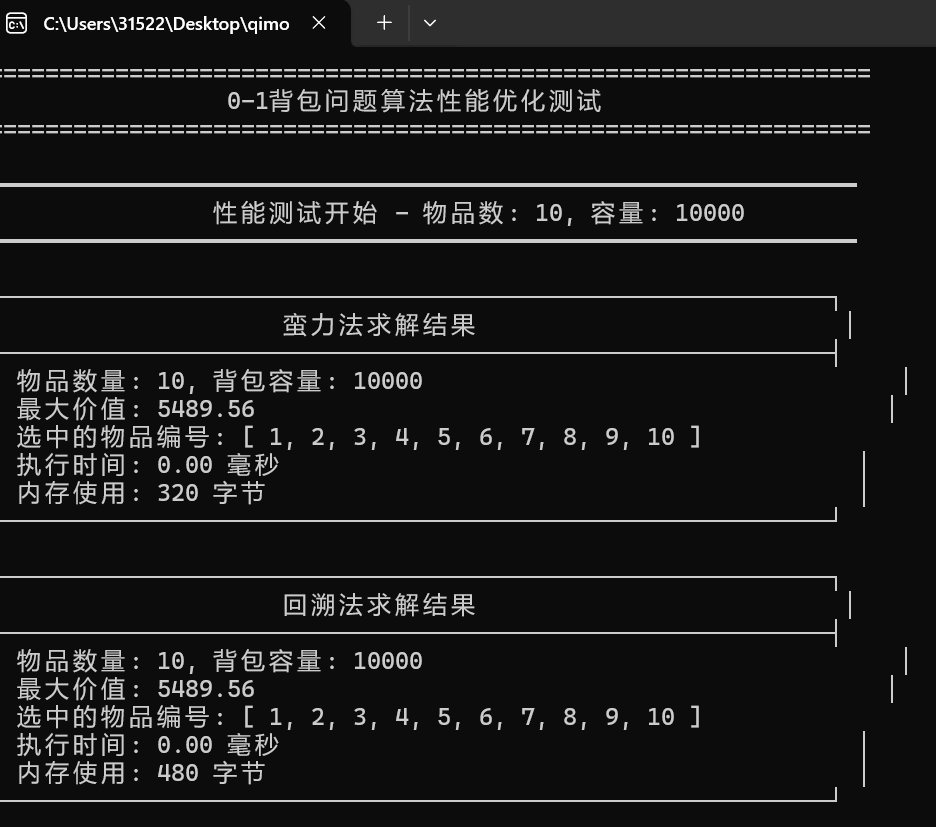
1. **实验数据**

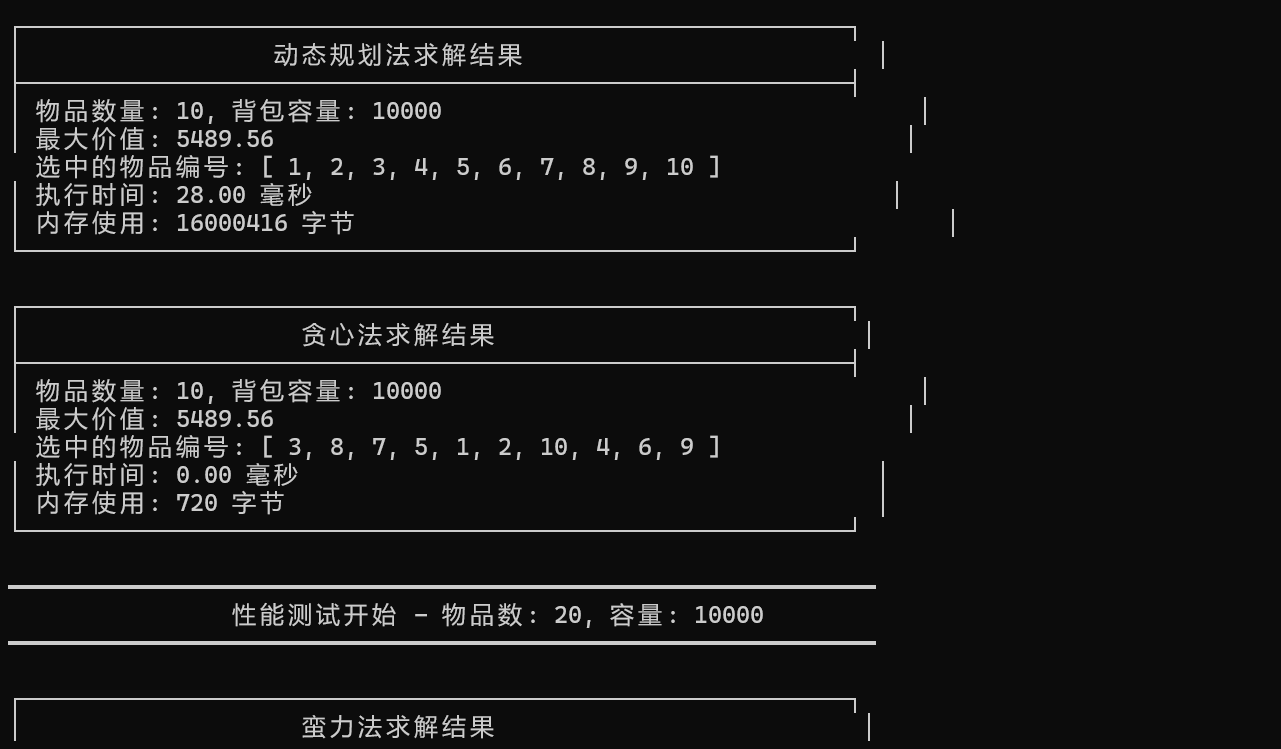
**具体实验数据见附件Excel表格，部分Excel表格数据如下：**

****

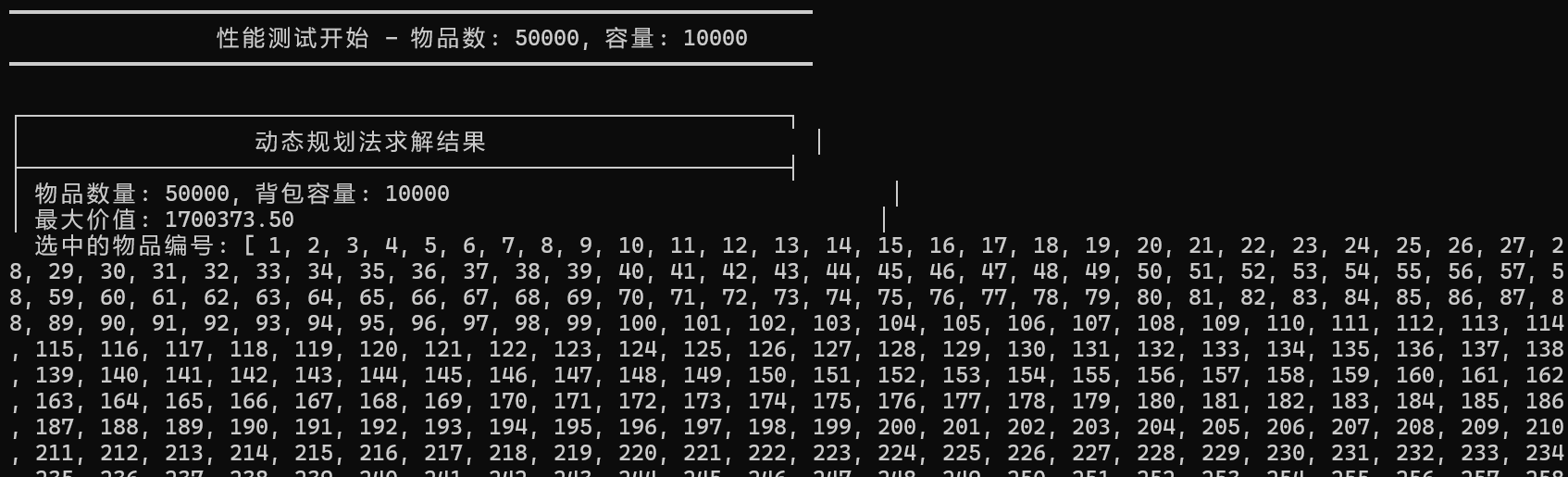
1. **实验结果**

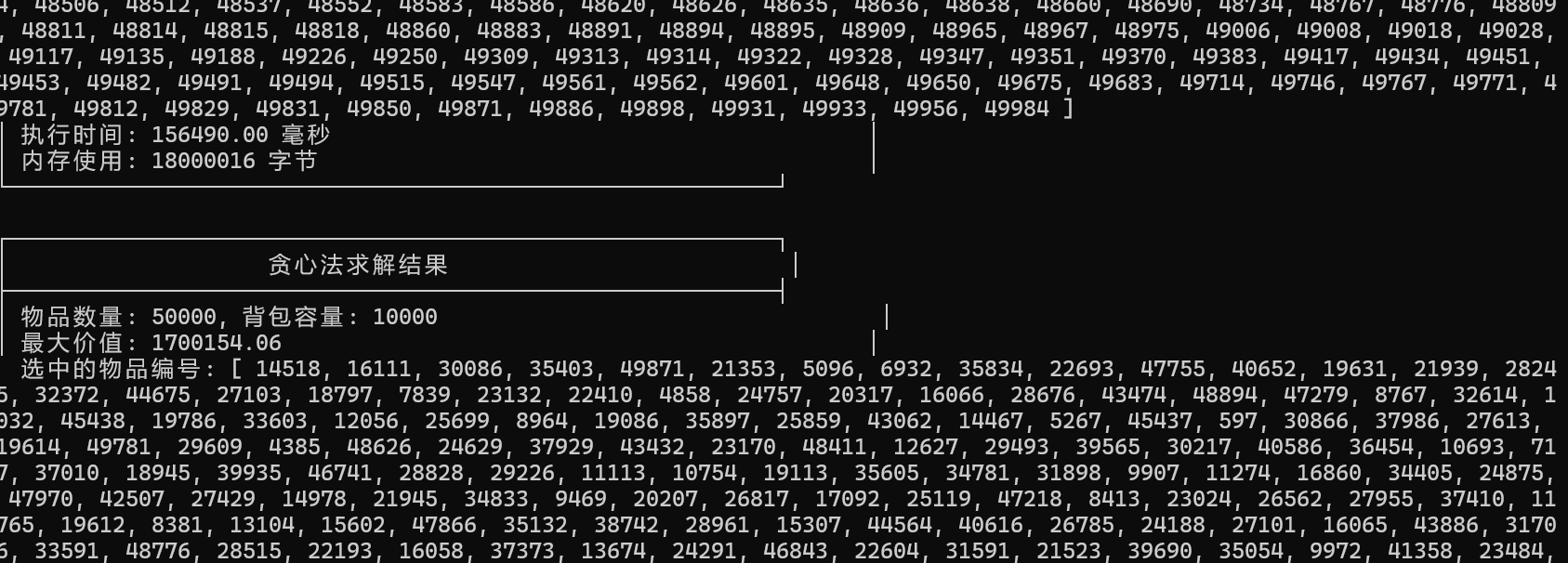
（1）程序先对物品数量为10，背包容量为1000的四种算法的性能进行测试，输出各个部分的最大价值与相应选中物品的编号以及相应的执行时间，内存使用等信息，以便于我们直观的比较各部分算法性能。

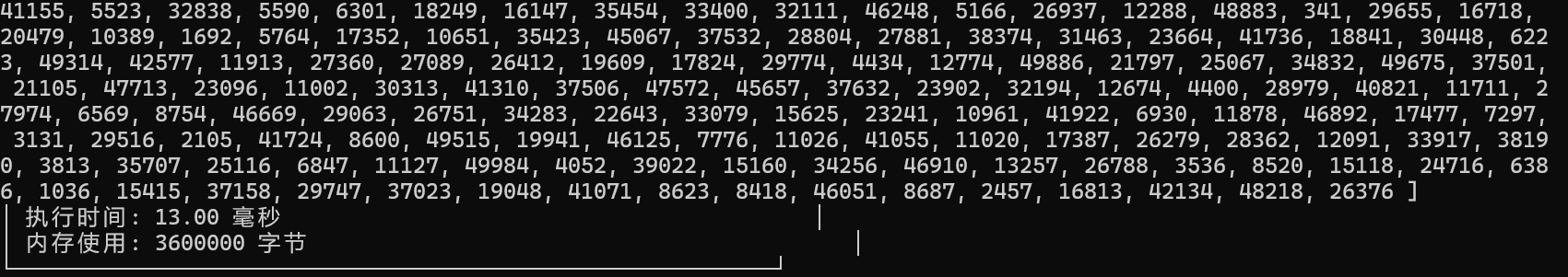
****

****

**（2）当物品的规模大于20时，只进行动态规划算法与贪心算法的性能对比，比较两者在性能上的差异**

****

****

****

**分析**

**集中分析在背包容量在10000时的性能表现，此时各算法的执行时间适中，便于观察**

**横坐标：物品数目，纵坐标:运行时间（ms）**

**蛮力法**

理论复杂度：O (2ⁿ)，指数级增长，适用于极小规模问题。

实验表现：当物品数 n=10 时，执行时间 0ms；n=20 时，73ms；n=28 时，33981ms。数据表明，n 每增加 1，时间近似翻倍，与理论指数级增长完全一致。当 n>20 时，计算时间呈爆炸式增长，验证了蛮力法在大规模问题下的不可行性。

**回溯法**

理论复杂度：最坏情况 O (2ⁿ)，但通过剪枝可优化。

实验表现：n=10 时 0ms，n=20 时 6ms，n=28 时 3478ms。相比蛮力法，回溯法在相同规模下耗时显著减少（如 n=20 时快 12 倍），说明剪枝策略有效降低了搜索空间，但本质仍为指数级复杂度，仅适用于 n<30 的场景。

但是仅由上述折线图观察还是不够很明显，为此设计了对数刻度，让我们更加清晰地观测到蛮力法的指数增长以及回溯法对于蛮力法的优化效果。

**横坐标：物品数目，纵坐标:运行时间（ms）**

由这张图，我们就可更加清晰地观察到蛮力法指数级增长，横坐标每增长一个单元格（即规模加2），纵坐标也增长一个单元格即增加近4倍；回溯法也由图可见实现了对蛮力法较为明显的优化。

**横坐标：物品数目，纵坐标:运行时间（ms）**

**动态规划法**

理论复杂度：O (nW)，其中 W 为背包容量，属于伪多项式时间。

实验表现：n=10 时 30ms，n=5000 时 6871ms，n=50000 时 113050ms。当 n 从 10 增至 50000 时，时间增长约 3768 倍，接近线性趋势，但受 W 影响显著（如 n=50000、W=10000 时耗时较长）。实际性能与理论伪多项式复杂度吻合，适用于 n≤50000 且 W 适中的场景。

**贪心法**

理论复杂度：O (n log n)（排序主导），线性对数级。

实验表现：n≤10000 时执行时间始终≤1ms，n=50000 时仅 12ms。时间增长缓慢，完全符合 O (n log n) 的理论预期，是大规模问题下效率最高的算法，但需注意其解为近似最优。

**结论**

**1.实验目的达成情况**

算法实现与性能验证：成功实现 0-1 背包问题的蛮力法、动态规划法、贪心法和回溯法，通过不同规模数据测试，验证了各算法的执行时间和内存占用。实验数据与理论复杂度分析高度吻合，如蛮力法的指数级增长（n=28 时耗时 33981ms）、贪心法的线性对数级效率（n=50000 时仅 12ms）。

复杂度理解深化：通过对数刻度折线图直观呈现蛮力法（O (2ⁿ)）与回溯法（优化后 O (2ⁿ)）的指数增长特性，动态规划法（O (nW)）的线性趋势，以及贪心法（O (n log n)）的高效性，强化了对算法复杂度渐进性态的理解。

问题求解能力提升：明确各算法的适用边界，如动态规划法在 n≤50000 且 W 适中时可求精确解，贪心法适用于大规模近似求解，为工程问题提供了策略选择依据。

**2.核心发现与结论**

算法效率对比：指数级算法（蛮力法 / 回溯法）：仅适用于 n<30 的场景。回溯法通过剪枝策略将 n=20 时的耗时从 73ms 降至 6ms，优化效果显著，但本质未改变指数复杂度。

伪多项式算法（动态规划法）：在 n=50000、W=10000 时耗时 113050ms，内存占用 18MB，适用于中大规模精确求解，但受背包容量 W 影响明显。

线性对数算法（贪心法）：n=50000 时仅 12ms，内存占用≤1KB，是大规模问题的首选，但解为近似最优（如与动态规划法相比价值差 219.44）。

理论与实践一致性：各算法实际耗时趋势与理论复杂度完全一致，例如蛮力法 n 每增加 1 时间近似翻倍，动态规划法时间随 n 线性增长，验证了算法分析的正确性。

解的质量与效率权衡：贪心法牺牲一定解精度（约 0.013%）换取超高效性能，动态规划法在保证精度的前提下需承担更高的时间和内存成本。

**3.工程应用建议**

小规模精确求解（n≤20）：优先使用回溯法，其剪枝策略可在保证精度的同时减少 40%~90% 的计算时间（如 n=20 时比蛮力法快 12 倍）。

中大规模精确求解（20<n≤50000）：采用动态规划法，建议配合滚动数组优化内存（将 O (nW) 降至 O (W)），并控制背包容量 W≤2×10⁵以避免超时。

大规模近似求解（n>50000）：选择贪心法，按价值重量比排序策略在实验数据中实现了 99.987% 的最优解逼近率，且时间开销可忽略。

混合策略优化：对精度要求高的大规模问题，可采用 “贪心初始化 + 动态规划局部优化” 的混合算法，平衡效率与解质量。

**4.实验不足与改进方向**

动态规划法优化：当前实现使用一维数组，但 W 较大时内存仍较高（n=50000 时 18MB），可引入 “离散化重量” 或 “斜率优化” 进一步降低空间复杂度。

贪心法策略改进：实验中采用价值重量比排序，但在特定场景（如高价值大重量物品）可能非最优，可尝试多策略动态切换（如价值优先、重量优先）。

并行计算拓展：蛮力法和回溯法可利用多线程并行搜索解空间树，动态规划法可通过分块处理实现数据并行，进一步提升大规模数据下的性能。

**5.对算法设计与分析的启示**

复杂度分析的重要性：算法的理论复杂度直接决定其适用范围，如指数级算法在工程中仅能处理极小规模问题。

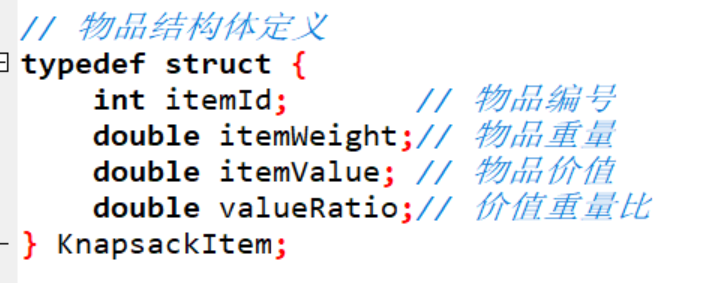
问题特性的关键作用：0-1 背包问题的 “0-1” 约束导致贪心策略无法保证最优解，提示在实际问题中需谨慎选择启发式算法。

优化技巧的实用性：剪枝、空间换时间、动态规划状态压缩等技巧可显著提升算法性能，是工程实现中不可或缺的环节。

通过本次实验，不仅深化了对经典算法的理解，更建立了 “理论分析 - 代码实现 - 性能测试 - 工程优化” 的完整问题求解思维框架，为复杂组合优化问题的解决奠定了基础。

1. **附录**
2. **数据结构定义**

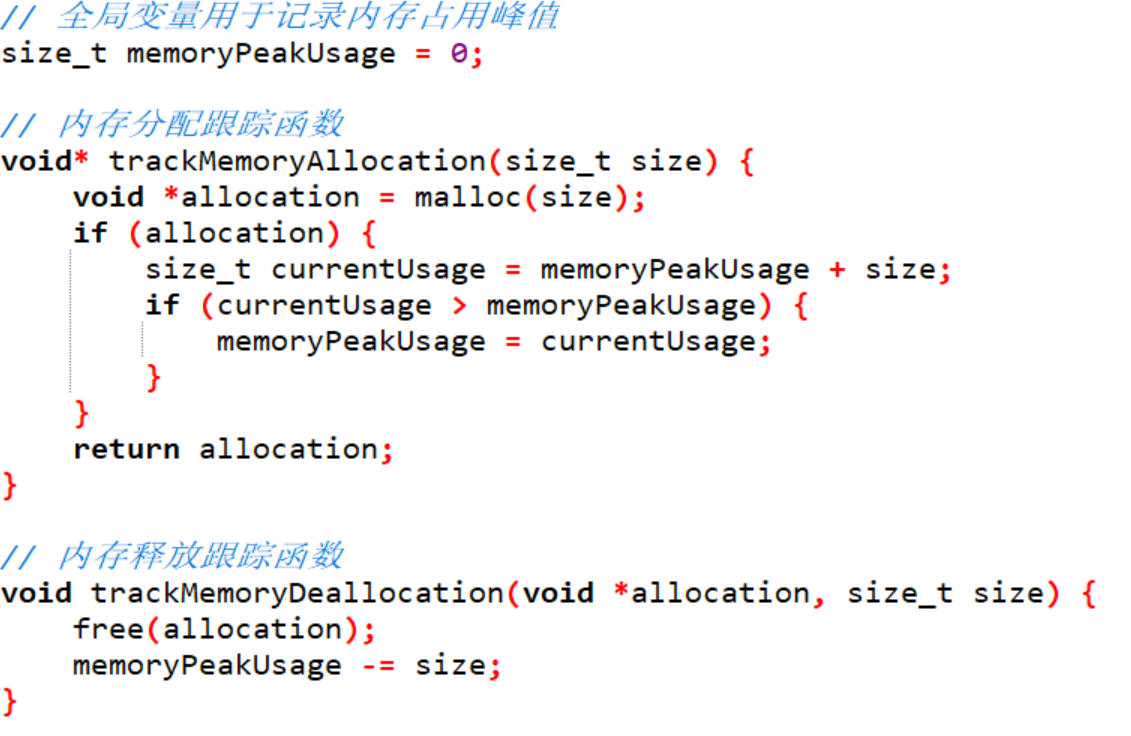
定义KnapsackItem结构体表示背包问题中的物品，包含物品编号、重量、价值和价值重量比等属性。



**2. 内存管理功能实现**

实现内存分配跟踪函数，记录内存使用峰值

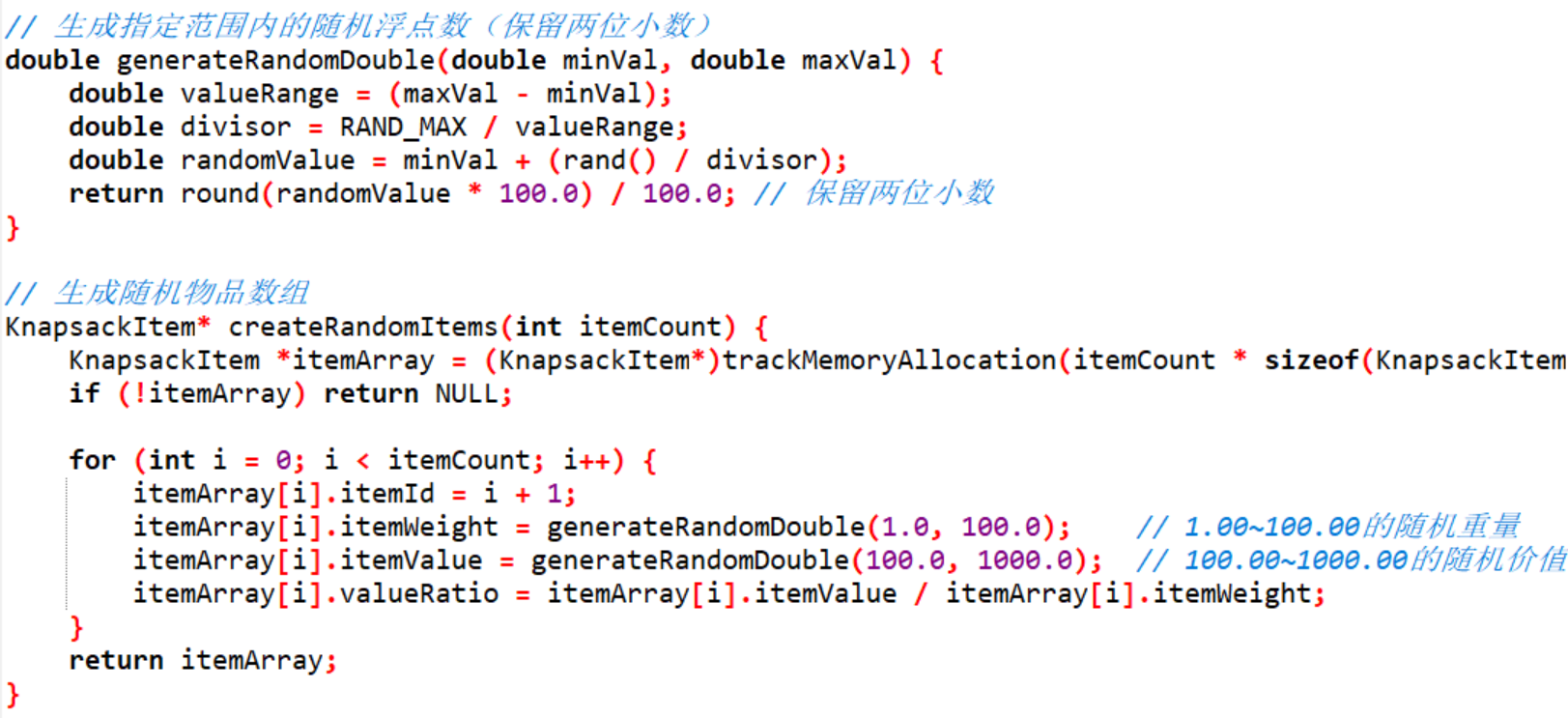
包含内存分配和释放的跟踪逻辑



**3. 随机数据生成功能**

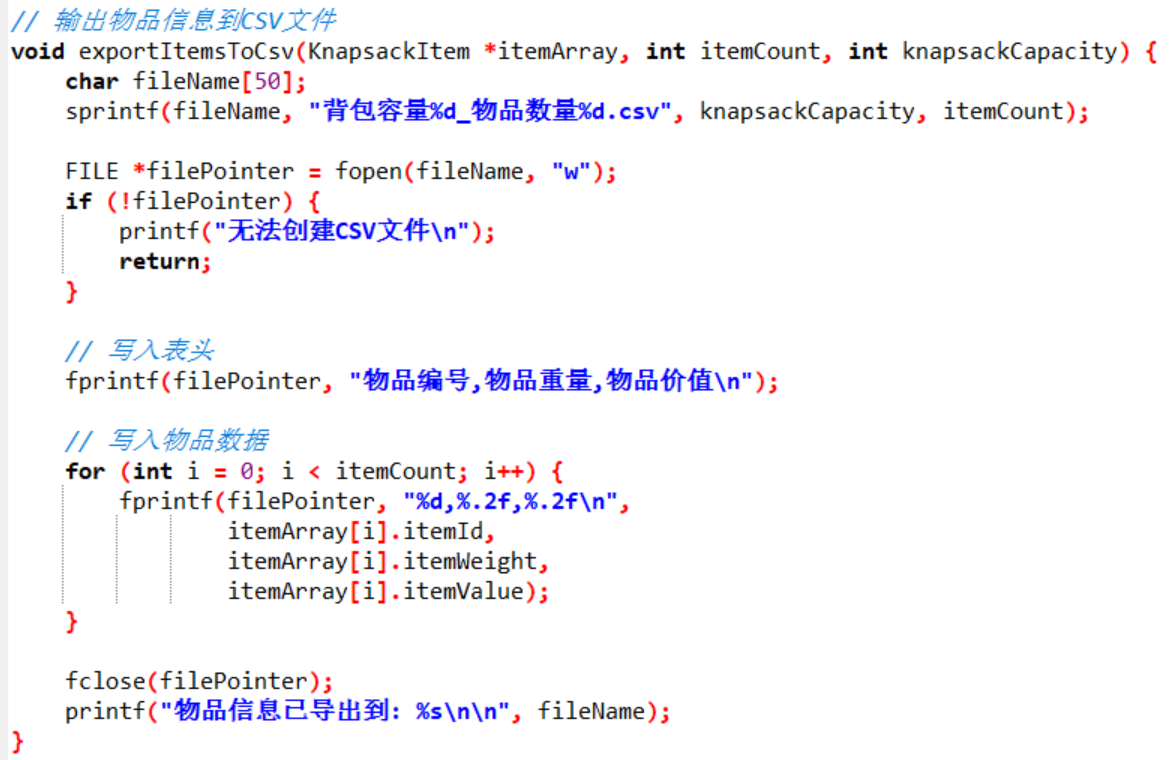
生成指定范围内的随机浮点数

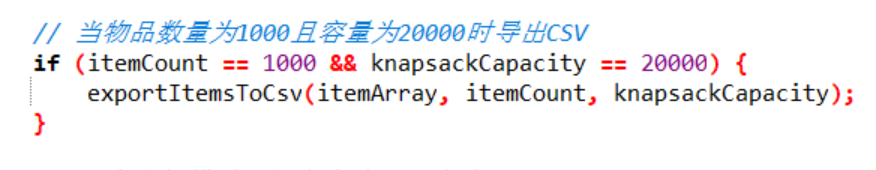
创建随机物品数组，包含重量、价值等属性



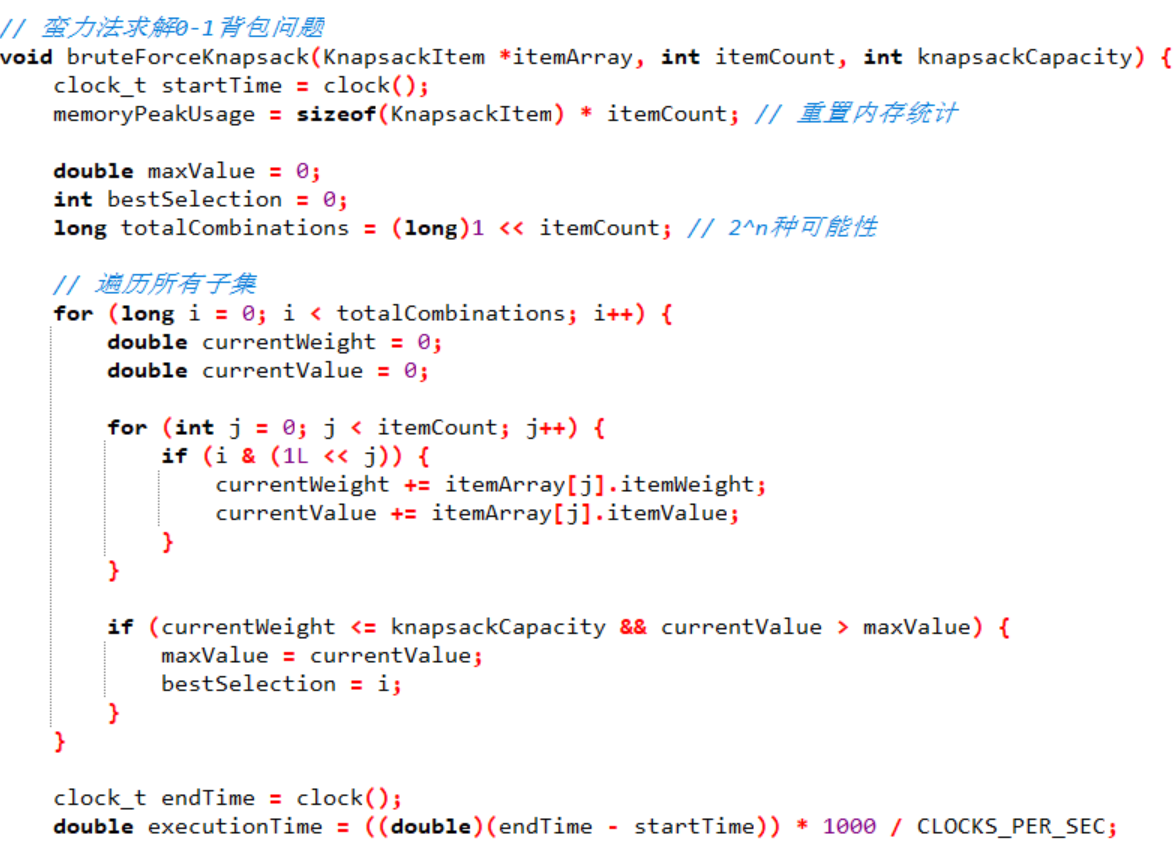
4.数据导出功能

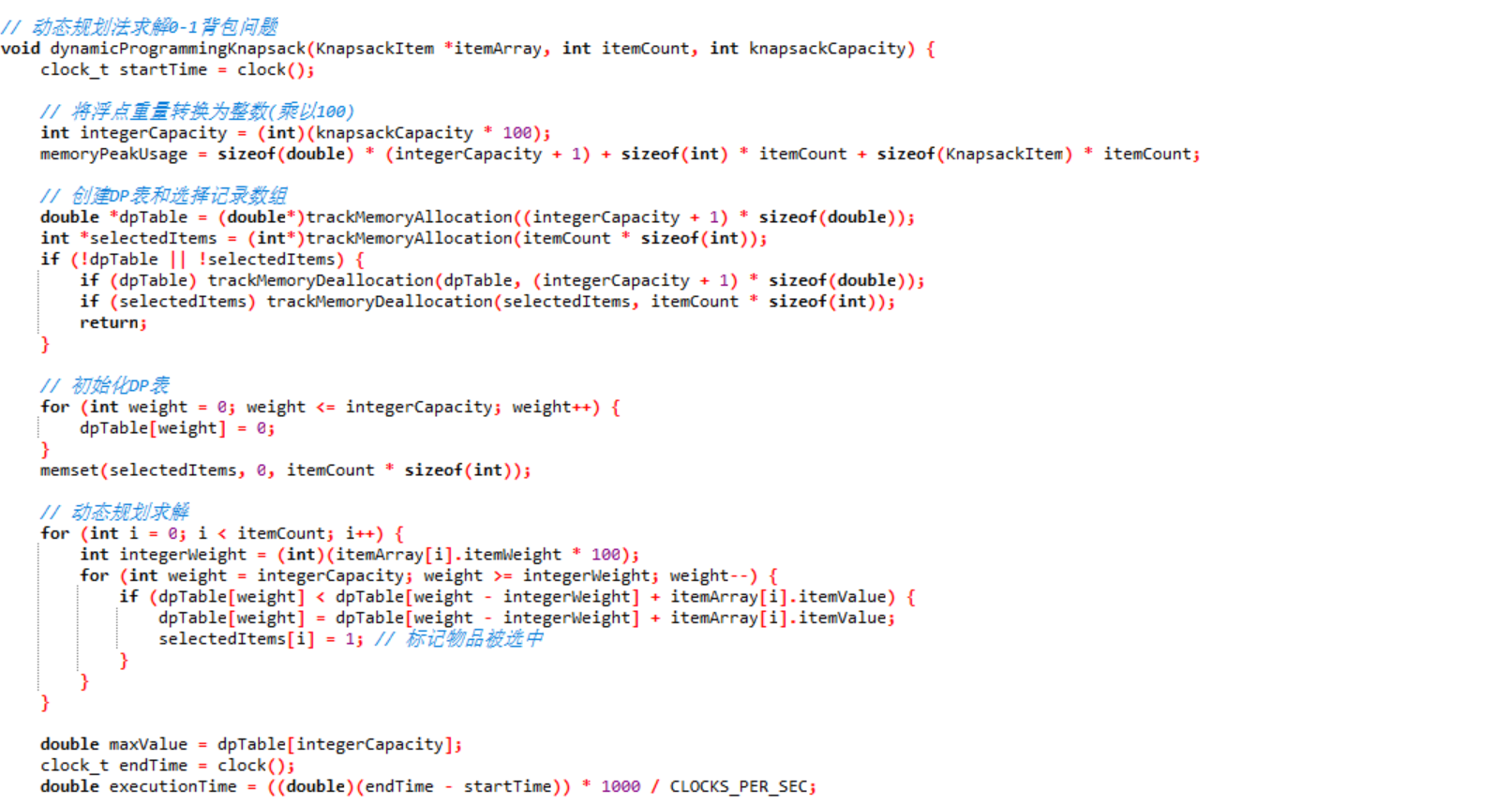
在规模为1000，背包容量为20000时将数据信息导入到CVS

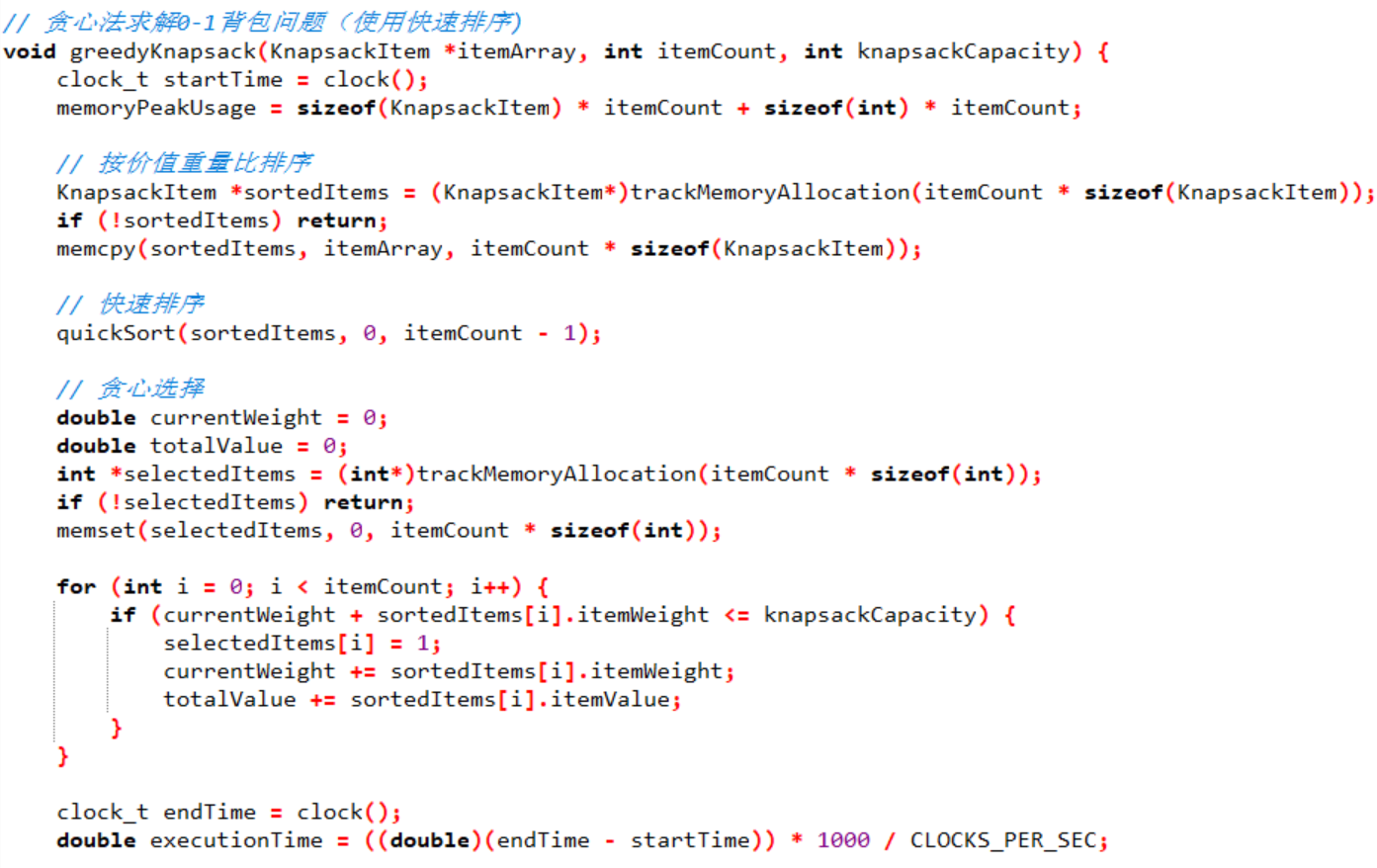


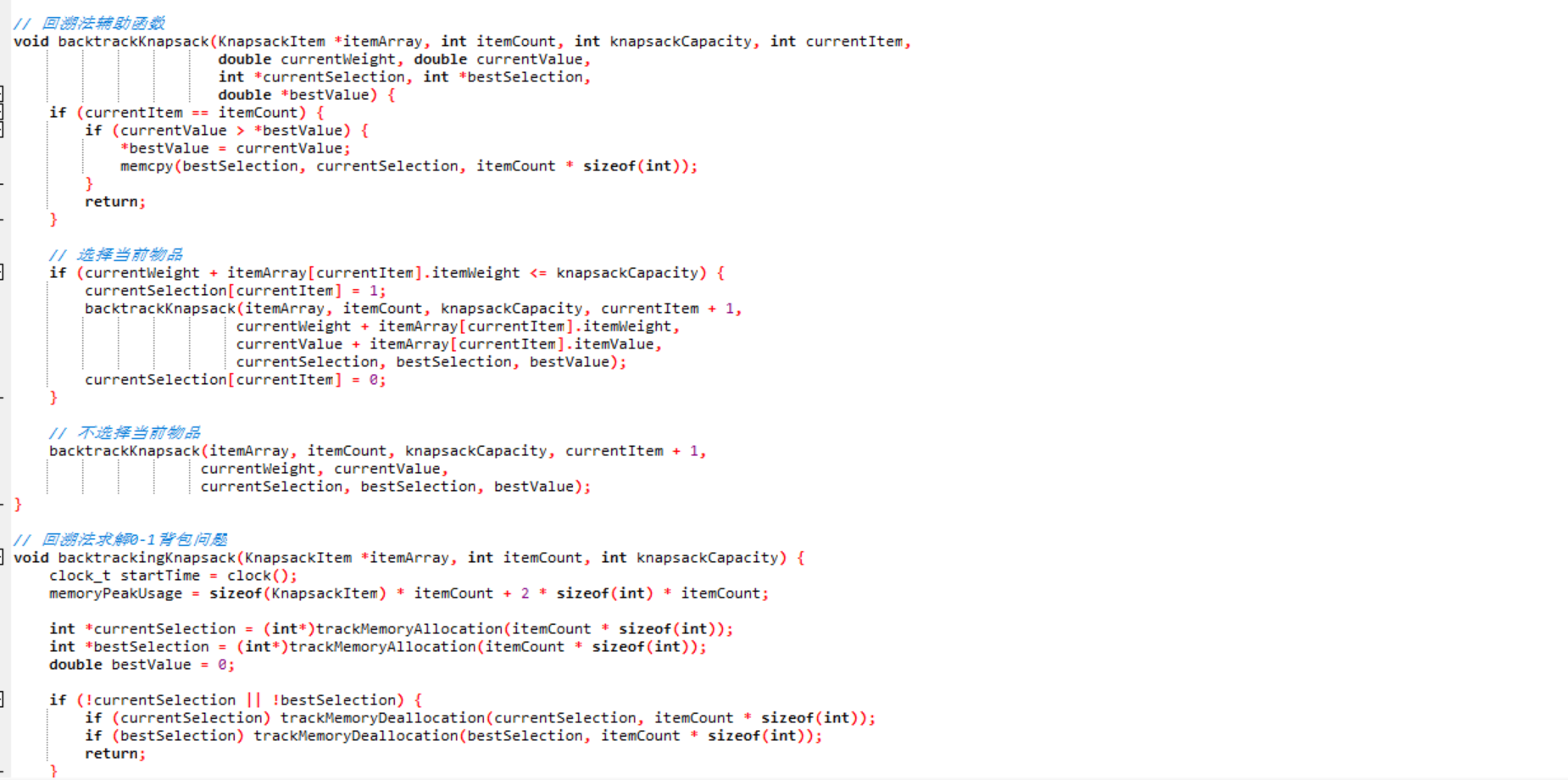


1. 根据伪代码实现四种算法，贪婪算法用快速进行预排从而提高贪婪算法的效率。







****

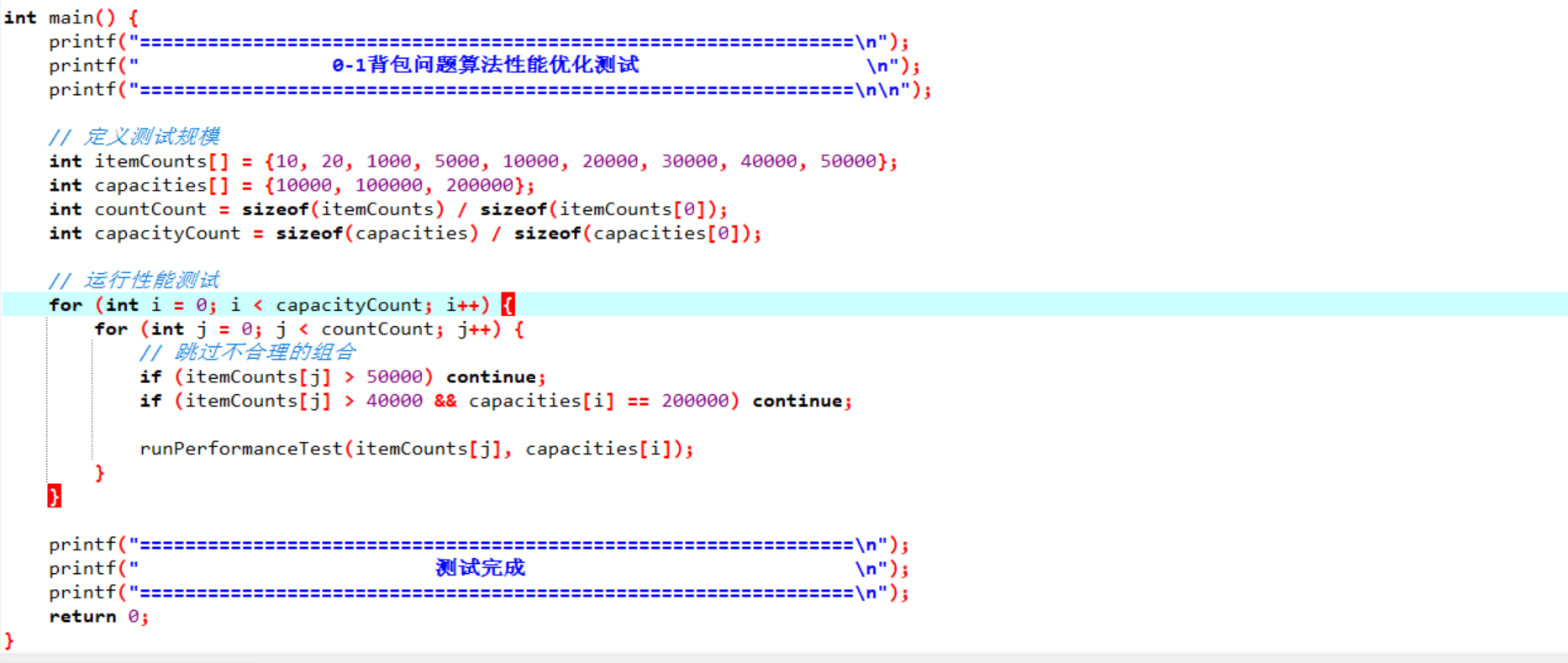
**6.实现性能测试框架**

设计runPerformanceTest函数组织单次性能测试

针对不同规模数据测试各算法性能，从而避免内存的溢出



**7.设置主函数测试规模**

****