

=====

文件夹: class051_KnapsackProblem

=====

[Markdown 文件]

=====

文件: README.md

=====

Class073: 01 背包问题深度解析

概述

01 背包问题是动态规划中的经典问题，也是算法面试和竞赛中的高频考点。本目录收集了来自各大算法平台的 01 背包相关题目，并提供了 Java、C++、Python 三种语言的详细实现。

题目列表

经典 01 背包模板题

1. **洛谷 P1048 [NOIP2005 普及组] 采药** - 经典 01 背包模板题
 - 文件: Code01_01Knapsack.java/Code01_01Knapsack.py
 - 链接: <https://www.luogu.com.cn/problem/P1048>

新补充题目（来自各大算法平台）

LeetCode 题目

2. **LeetCode 279. 完全平方数** - 完全背包问题
 - 文件: Code44_PerfectSquares.java/Code44_PerfectSquares.py/Code44_PerfectSquares.cpp
 - 链接: <https://leetcode.cn/problems/perfect-squares/>
 - 类型: 完全背包问题，求最少完全平方数个数
3. **LeetCode 377. 组合总和 IV** - 完全背包问题（排列数）
 - 文件: Code45_CombinationSumIV.java/Code45_CombinationSumIV.py/Code45_CombinationSumIV.cpp
 - 链接: <https://leetcode.cn/problems/combination-sum-iv/>
 - 类型: 完全背包问题，计算排列数
4. **LeetCode 518. 零钱兑换 II** - 完全背包问题（组合数）
 - 文件: Code46_CoinChangeII.java/Code46_CoinChangeII.py/Code46_CoinChangeII.cpp
 - 链接: <https://leetcode.cn/problems/coin-change-ii/>
 - 类型: 完全背包问题，计算组合数
5. **LeetCode 416. 分割等和子集** - 经典 01 背包应用
 - 文件: Code01_01Knapsack.java/Code01_01Knapsack.py
 - 链接: <https://leetcode.cn/problems/partition-equal-subset-sum/>

3. **LeetCode 494. 目标和** - 01 背包变形题
 - 文件: Code03_TargetSum. java/Code03_TargetSum. py
 - 链接: <https://leetcode.cn/problems/target-sum/>
4. **LeetCode 474. 一和零** - 二维费用 01 背包
 - 文件: Code07_OnesAndZeros. java/Code07_OnesAndZeros. py/Code07_OnesAndZeros. cpp
 - 链接: <https://leetcode.cn/problems/ones-and-zeroes/>
5. **LeetCode 879. 盈利计划** - 三维费用 01 背包
 - 文件: Code08_ProfitableSchemes. java/Code08_ProfitableSchemes. py/Code08_ProfitableSchemes. cpp
 - 链接: <https://leetcode.cn/problems/profitable-schemes/>
6. **LeetCode 322. 零钱兑换** - 完全背包变形题
 - 链接: <https://leetcode.cn/problems/coin-change/>
7. **LeetCode 518. 零钱兑换 II** - 完全背包变形题
 - 链接: <https://leetcode.cn/problems/coin-change-ii/>
8. **LeetCode 40. 组合总和 II** - 01 背包变形题
 - 链接: <https://leetcode.cn/problems/combination-sum-ii/>
9. **LeetCode 1049. 最后一块石头的重量 II** - 01 背包变形题
 - 文件: Code04_LastStoneWeightII. java/Code04_LastStoneWeightII. py
 - 链接: <https://leetcode.cn/problems/last-stone-weight-ii/>

洛谷题目

10. **洛谷 P1049 [NOIP2001 普及组] 装箱问题** - 01 背包变形题
 - 文件: Code10_PackingProblem. java/Code10_PackingProblem. py/Code10_PackingProblem. cpp
 - 链接: <https://www.luogu.com.cn/problem/P1049>
11. **洛谷 P1060 [NOIP2006 普及组] 开心的金明** - 经典 01 背包应用
 - 文件: Code11_HappyJinming. java/Code11_HappyJinming. py/Code11_HappyJinming. cpp
 - 链接: <https://www.luogu.com.cn/problem/P1060>
12. **洛谷 P2347 [NOIP2011 普及组] 碍码称重** - 01 背包变形题
 - 链接: <https://www.luogu.com.cn/problem/P2347>

AtCoder 题目

13. **AtCoder Educational DP Contest D - Knapsack 1** - 经典 01 背包
 - 文件: Code09_Knapsack1. java/Code09_Knapsack1. py/Code09_Knapsack1. cpp
 - 链接: https://atcoder.jp/contests/dp/tasks/dp_d

24. **AtCoder DP Contest E - Knapsack 2** - 大容量 01 背包
- 文件: Code49_Knapsack2. java/Code49_Knapsack2. py/Code49_Knapsack2. cpp
 - 链接: https://atcoder.jp/contests/dp/tasks/dp_e
 - 类型: 价值维度 DP, 处理大容量背包

POJ 题目

14. **POJ 1837 Balance** - 01 背包变形题 (力矩平衡)
- 文件: Code12_Balance. java/Code12_Balance. py/Code12_Balance. cpp
 - 链接: <http://poj.org/problem?id=1837>
15. **POJ 1276 Cash Machine** - 多重背包转 01 背包
- 文件: Code13_CashMachine. java/Code13_CashMachine. py/Code13_CashMachine. cpp
 - 链接: <http://poj.org/problem?id=1276>
16. **POJ 2184 Cow Exhibition** - 二维费用 01 背包
- 文件: Code47_CowExhibition. java/Code47_CowExhibition. py/Code47_CowExhibition. cpp
 - 链接: <http://poj.org/problem?id=2184>
 - 类型: 二维费用 01 背包, 处理负数坐标

CodeForces 题目

17. **Codeforces 546D Soldier and Number Game** - 数论+背包问题
- 链接: <https://codeforces.com/problemset/problem/546/D>
18. **Codeforces 1132E Knapsack** - 混合背包问题
- 链接: <https://codeforces.com/problemset/problem/1132/E>

HackerRank 题目

19. **HackerRank The Coin Change Problem** - 完全背包变形题
- 链接: <https://www.hackerrank.com/challenges/coin-change/problem>

牛客网题目

20. **牛客 NC15411 硬币** - 多重背包问题
- 链接: <https://ac.nowcoder.com/acm/problem/15411>

UVA 题目

21. **UVA 10616 Divisible Group Sums** - 分组背包+模数运算
- 文件:
 - 链接:
https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=1557
 - 类型: 分组背包+模数运算, 选择 M 个数字能被 D 整除

HDU 题目

22. **HDU 2955 Robberies** - 概率背包问题

- 文件: Code48_Robberies.java/Code48_Robberies.py/Code48_Robberies.cpp
- 链接: <https://acm.hdu.edu.cn/showproblem.php?pid=2955>
- 类型: 概率背包问题, 安全概率计算

23. **HDU 3535 AreYouBusy** - 分组背包问题

- 链接: <https://acm.hdu.edu.cn/showproblem.php?pid=3535>

解题思路总结

01 背包基本模型

- **问题描述**: 有 N 个物品和一个容量为 W 的背包, 每个物品有重量 $w[i]$ 和价值 $v[i]$, 每个物品只能使用一次, 求能装入背包的最大价值。
- **状态定义**: $dp[i][j]$ 表示前 i 个物品, 背包容量为 j 时能获得的最大价值
- **状态转移方程**:
```  
 $dp[i][j] = \max(dp[i-1][j], dp[i-1][j-w[i]] + v[i]) \quad (\text{当 } j \geq w[i] \text{ 时})$   
 $dp[i][j] = dp[i-1][j] \quad (\text{当 } j < w[i] \text{ 时})$   
```

- **空间优化**: 使用滚动数组, 倒序遍历背包容量

新补充题目的解题技巧

完全背包问题 (LeetCode 279, 377, 518)

- **特点**: 每个物品可以无限次使用
- **状态转移**: 正序遍历背包容量
- **关键区别**: 01 背包倒序遍历, 完全背包正序遍历

二维费用背包 (POJ 2184, LeetCode 474)

- **特点**: 每个物品有两个限制条件
- **状态定义**: $dp[j][k]$ 表示第一个限制为 j , 第二个限制为 k 时的最优解
- **遍历顺序**: 双重循环, 从大到小遍历

概率背包 (HDU 2955)

- **特点**: 将概率转化为安全概率 ($1-p$)
- **状态定义**: $dp[j]$ 表示抢劫金额为 j 时的最大安全概率
- **关键点**: 概率相乘转化为安全概率相乘

模数运算背包 (UVA 10616)

- **特点**: 涉及模数运算和组合计数
- **状态定义**: $dp[j][k]$ 表示选择 j 个数字, 和对 D 取模为 k 的方案数
- **关键技巧**: 正确处理负数取模

大容量背包 (AtCoder Knapsack 2)

- **特点**: 背包容量极大 (10^9), 但物品价值较小
- **解法**: 转换维度, 以价值为状态进行 DP
- **状态定义**: $dp[j]$ 表示获得价值 j 所需的最小重量

常见变形题型

1. **目标和问题**: 将问题转化为 01 背包, 选择一些数使其和为特定值
2. **分割等和子集**: 判断是否能将数组分割成两个和相等的子集
3. **二维费用背包**: 每个物品有两个限制条件 (如 0 和 1 的个数)
4. **三维费用背包**: 每个物品有三个限制条件 (如人数、利润等)
5. **多重背包**: 每个物品有指定数量, 可转化为 01 背包求解
6. **完全背包**: 每个物品可以无限次使用
7. **分组背包**: 物品分组, 每组只能选一个
8. **依赖背包**: 物品间存在依赖关系

复杂度分析

基础 01 背包

- **时间复杂度**: $O(N * W)$, 其中 N 是物品数量, W 是背包容量
- **空间复杂度**: $O(W)$, 使用滚动数组优化后

新补充题目的复杂度

1. **完全背包问题**: $O(N * W)$
2. **二维费用背包**: $O(N * W1 * W2)$, 其中 $W1, W2$ 是两个维度的限制
3. **三维费用背包**: $O(N * W1 * W2 * W3)$
4. **模数运算背包**: $O(N * M * D)$, 其中 M 是选择数量, D 是除数
5. **大容量背包**: $O(N * V)$, 其中 V 是总价值, 适用于 W 很大但 V 较小的情况

空间优化技巧

1. **滚动数组**: 将二维 DP 优化为一维
2. **状态压缩**: 使用位运算或模数运算减少状态空间
3. **维度转换**: 当容量过大时, 转换为价值维度 DP

工程化考虑

1. 异常处理与边界条件

- **输入验证**: 检查数组为空、容量为负等异常情况
- **边界处理**: 处理 $M=0, D=0, W=0$ 等特殊情况
- **溢出防护**: 处理大数运算时的溢出问题

2. 性能优化策略

- **空间优化**: 优先使用滚动数组减少内存占用

- **时间优化**: 避免不必要的循环和计算
- **缓存友好**: 优化内存访问模式, 提高缓存命中率

3. 可配置性与扩展性

- **参数化设计**: 将背包容量、物品数量等作为参数
- **模块化实现**: 分离 DP 逻辑和业务逻辑
- **接口设计**: 提供统一的解题接口, 支持多种输入格式

4. 测试覆盖与质量保证

- **单元测试**: 覆盖正常情况、边界情况、异常情况
- **性能测试**: 测试大规模数据的处理能力
- **回归测试**: 确保修改不会破坏现有功能

面试要点

1. 基础理解深度

- **本质理解**: 01 背包是选择问题, 每个物品选或不选
- **状态转移**: 能够推导状态转移方程
- **空间优化**: 理解滚动数组的原理和实现

2. 变种识别能力

- **题型识别**: 快速识别各种背包问题的变种
- **转化技巧**: 将复杂问题转化为标准背包问题
- **维度分析**: 正确分析问题的维度和约束条件

3. 复杂度分析能力

- **时间复杂度**: 准确分析算法的时间复杂度
- **空间复杂度**: 分析空间使用情况
- **优化潜力**: 识别算法的优化空间

4. 工程实践能力

- **代码质量**: 编写清晰、可读、可维护的代码
- **异常处理**: 正确处理各种边界和异常情况
- **测试思维**: 设计全面的测试用例

实战技巧

1. 调试与问题定位

- **打印调试**: 使用 `System.out.println` 打印关键变量
- **断点思维**: 在关键位置添加调试输出
- **逐步验证**: 从小规模数据开始验证算法正确性

2. 性能优化策略

- **常数优化**: 减少不必要的计算和函数调用
- **内存优化**: 合理使用数据结构，减少内存占用
- **算法选择**: 根据数据规模选择合适的算法

3. 笔试面试技巧

- **模板准备**: 准备常用算法的代码模板
- **时间管理**: 合理分配解题时间
- **沟通表达**: 清晰表达解题思路和复杂度分析

扩展应用

1. 机器学习与背包问题

- **特征选择**: 背包问题可用于特征选择优化
- **资源分配**: 在资源受限的机器学习任务中的应用
- **模型压缩**: 神经网络剪枝中的背包问题应用

2. 大数据场景优化

- **分布式计算**: 大规模背包问题的分布式求解
- **近似算法**: 处理超大规模数据的近似解法
- **流式处理**: 数据流场景下的背包问题求解

3. 跨语言实现差异

- **Java**: 注重面向对象设计和异常处理
- **C++**: 强调性能优化和内存管理
- **Python**: 关注代码简洁性和可读性

通过系统学习本目录的所有题目和代码，您将全面掌握 01 背包问题及其各种变种，具备解决复杂动态规划问题的能力。

文件: SUMMARY.md

01 背包问题深度总结

一、基本概念

01 背包问题是动态规划中的经典问题，描述如下：

- 有 N 个物品和一个容量为 W 的背包
- 每个物品有两个属性：重量 $w[i]$ 和价值 $v[i]$
- 每个物品只能选择一次（0 表示不选，1 表示选）
- 目标是在不超过背包容量的前提下，使得装入背包的物品价值总和最大

二、解题思路

1. 状态定义

- $dp[i][j]$ 表示前 i 个物品，在背包容量为 j 时能获得的最大价值

2. 状态转移方程

```

$dp[i][j] = \max(dp[i-1][j], dp[i-1][j-w[i]] + v[i])$  (当  $j \geq w[i]$  时)

$dp[i][j] = dp[i-1][j]$  (当  $j < w[i]$  时)

```

3. 初始状态

- $dp[0][j] = 0$ (没有物品时价值为 0)

- $dp[i][0] = 0$ (背包容量为 0 时价值为 0)

4. 空间优化

使用滚动数组，倒序遍历背包容量：

```

```
for (int i = 1; i <= n; i++) {
 for (int j = W; j >= w[i]; j--) {
 dp[j] = max(dp[j], dp[j - w[i]] + v[i]);
 }
}
```

## ## 三、常见题型及解法

### #### 1. 经典 01 背包

**\*\*特征\*\*:** 直接给出物品重量和价值，求最大价值

**\*\*解法\*\*:** 标准 01 背包模板

**\*\*典型题目\*\*:** 洛谷 P1048 采药、AtCoder DP Contest D

### #### 2. 目标和问题

**\*\*特征\*\*:** 给定数组和目标值，通过添加+/-符号使表达式等于目标值

**\*\*解法\*\*:** 转化为 01 背包，选择一些数使其和为特定值

**\*\*公式推导\*\*:** 设正数和为  $P$ ，负数和为  $N$ ，则  $sum = P + N$ ,  $target = P - N$ , 解得  $P = (sum + target) / 2$ ，问题转化为求有多少种方式选出元素和为  $P$

**\*\*典型题目\*\*:** LeetCode 494. 目标和

### #### 3. 分割等和子集

**\*\*特征\*\*:** 判断是否能将数组分割成两个和相等的子集

**\*\*解法\*\*:** 转化为 01 背包，判断是否能装满容量为  $sum/2$  的背包

**\*\*注意事项\*\*:** 需要先判断总和是否为偶数

**\*\*典型题目\*\*:** LeetCode 416. 分割等和子集

- **解题思路**: 将问题转化为 01 背包问题, 判断是否存在一个子集, 其和为数组总和的一半
- **核心代码**: 使用一维 DP 数组,  $dp[j]$  表示是否可以选择一些元素使其和为  $j$
- **优化点**: 提前剪枝 (判断总和奇偶性、最大值是否超过 target)

#### #### 4. 二维费用背包

**特征**: 每个物品有两个限制条件 (如 0 和 1 的个数)

**解法**: 使用二维 dp 数组,  $dp[i][j][k]$  表示前  $i$  个物品, 使用  $j$  个 0 和  $k$  个 1 的最大价值

**空间优化**: 可以使用滚动数组优化到  $O(W_1 * W_2)$

**典型题目**: LeetCode 474. 一和零

#### #### 5. 三维费用背包

**特征**: 每个物品有三个限制条件 (如人数、利润等)

**解法**: 使用三维 dp 数组, 或根据具体问题进行状态设计

**典型题目**: LeetCode 879. 盈利计划

#### #### 6. 多重背包

**特征**: 每个物品有指定数量

**解法**: 二进制优化转化为 01 背包

**二进制优化原理**: 将数量  $s$  分解为  $2^0, 2^1, \dots, 2^k, s - 2^{k+1} + 1$ , 每个二进制项代表一个物品

**典型题目**: POJ 1276 Cash Machine、牛客 NC15411

#### #### 7. 分组背包

**特征**: 物品分组, 每组只能选一个物品

**解法**: 每组内遍历物品, 组间使用 01 背包

**注意遍历顺序**: 先遍历组, 再遍历容量 (逆序), 最后遍历组内物品

**典型题目**: HDU 3535 AreYouBusy

#### #### 8. 完全背包

**特征**: 每个物品可以选无限次

**解法**: 正序遍历背包容量

**与 01 背包的区别**: 01 背包逆序遍历, 完全背包正序遍历

**典型题目**: LeetCode 322. 零钱兑换、LeetCode 518. 零钱兑换 II

#### #### 9. 背包变形题

**最后一块石头的重量**: 转化为最小化两堆石头的重量差

**砝码称重**: 转化为正负背包问题

**力矩平衡**: 转化为中心对称的背包问题

**概率背包**: 使用概率作为价值

**模数背包**: 结合模运算的特殊背包问题

#### #### 10. 混合背包

**特征**: 同时包含 01 背包、完全背包、多重背包的元素

**\*\*解法\*\*:** 根据不同类型的物品采用不同的处理方式

**\*\*典型题目\*\*:** Codeforces 1132E Knapsack

## ## 四、解题技巧

### #### 1. 识别 01 背包特征

- 有“选”或“不选”的决策
- 有容量或资源限制
- 求最优值（最大/最小）
- 每个物品只能使用一次

### #### 2. 状态设计技巧

- **一维 DP**: 适用于单约束条件的背包问题
- **二维 DP**: 适用于双约束条件的背包问题
- **三维 DP**: 适用于三约束条件的背包问题
- **布尔型 DP**: 适用于判断可行性问题
- **计数型 DP**: 适用于求方案数问题

### #### 3. 转化技巧

- **目标和转背包**: 通过数学推导将目标和转化为 01 背包问题
- **分割等和子集**: 转化为容量为  $\text{sum}/2$  的装满问题
- **最小差值**: 转化为容量为  $\text{sum}/2$  的最接近问题
- **多维限制**: 使用多维 DP 数组处理多个限制条件

### #### 4. 优化技巧

- **滚动数组**: 将二维 DP 优化为一维 DP，节省空间
- **二进制拆分**: 将多重背包转化为 01 背包
- **坐标平移**: 处理负数状态，如砝码称重问题
- **初始状态优化**: 根据问题特点设置合理的初始状态
- **剪枝**: 对不可能达到的状态提前终止计算

### #### 5. 遍历顺序的重要性

- **01 背包**: 物品正序，容量逆序
- **完全背包**: 物品正序，容量正序
- **分组背包**: 组正序，容量逆序，组内物品正序
- **混合背包**: 根据物品类型调整容量遍历顺序

### #### 6. 数学模型建立

- **状态定义**: 明确 dp 数组的含义
- **转移方程**: 根据选或不选的决策建立转移关系
- **边界条件**: 正确设置初始状态
- **结果提取**: 从最终状态中获取答案

## ## 五、复杂度分析

### #### 时间复杂度

- \*\*标准 01 背包\*\*:  $O(N * W)$ , 其中 N 是物品数量, W 是背包容量
- \*\*二维费用背包\*\*:  $O(N * W1 * W2)$ , W1 和 W2 是两个费用维度
- \*\*三维费用背包\*\*:  $O(N * W1 * W2 * W3)$ , W1、W2 和 W3 是三个费用维度
- \*\*多重背包（普通）\*\*:  $O(N * W * s)$ , s 是物品数量上限
- \*\*多重背包（二进制优化）\*\*:  $O(N * W * \log s)$
- \*\*分组背包\*\*:  $O(G * W * K)$ , G 是组数, K 是每组的物品数

### #### 空间复杂度

- \*\*未优化二维 DP\*\*:  $O(N * W)$
- \*\*滚动数组优化一维 DP\*\*:  $O(W)$
- \*\*二维费用背包\*\*:  $O(W1 * W2)$
- \*\*三维费用背包\*\*:  $O(W1 * W2 * W3)$

### #### 时间复杂度优化技巧

1. \*\*剪枝优化\*\*: 提前过滤不可能达到的状态
2. \*\*状态压缩\*\*: 使用位运算等方式优化状态表示
3. \*\*数学优化\*\*: 利用问题的数学性质减少计算量
4. \*\*预处理优化\*\*: 提前计算某些中间结果

### #### 空间复杂度优化技巧

1. \*\*滚动数组\*\*: 只保留必要的状态
2. \*\*状态压缩\*\*: 使用位掩码等方式减少空间占用
3. \*\*原地更新\*\*: 在某些情况下可以原地更新 DP 数组
4. \*\*稀疏矩阵\*\*: 对于稀疏状态, 可以使用哈希表等数据结构

## ## 六、边界场景处理

### #### 1. 空输入处理

- \*\*空数组\*\*: 当输入数组为空时, 返回合理的默认值 (如 0 或 false)
- \*\*零容量\*\*: 当背包容量为 0 时, 只能选择 0 个物品
- \*\*零物品\*\*: 当没有物品时, 能获得的价值为 0

### #### 2. 极端值处理

- \*\*超大容量\*\*: 当背包容量远大于物品总重量时, 可以直接返回所有物品的价值和
- \*\*物品重量超过容量\*\*: 需要跳过无法放入的物品
- \*\*负数价值/重量\*\*: 根据问题描述判断如何处理负数输入
- \*\*零价值/重量\*\*: 需要正确处理零值物品

### #### 3. 特殊数据分布

- \*\*有序数据\*\*: 验证算法结果不受数据顺序影响

- **重复数据**: 确保算法能正确处理重复的物品
- **极端分布**: 如物品重量全部相同、价值全部相同等情况

#### #### 4. 边界条件测试用例

- **最小输入**:  $n=1, w=1, v=1, capacity=1$
- **临界情况**: 总和正好为奇数 (分割等和子集问题)
- **目标无法达到**: 如目标和问题中 target 无法通过数组元素组合得到
- **全部选中**: 所有物品都能放入背包的情况
- **全部无法选中**: 所有物品都无法放入背包的情况

### ## 七、工程化考虑

#### #### 1. 异常处理

- **输入验证**: 检查参数合法性, 如非负数、空指针等
- **异常抛出**: 对非法输入明确抛出异常并提供详细错误信息
- **防御性编程**: 使用 try-catch 或条件检查避免程序崩溃

#### #### 2. 性能优化

- **内存优化**: 使用滚动数组减少内存占用
- **计算优化**: 避免重复计算, 使用预处理技术
- **数据结构选择**: 根据实际问题选择合适的数据结构
- **并行计算**: 对于大规模数据, 考虑并行优化的可能性

#### #### 3. 可配置性

- **参数化设计**: 将关键参数提取为可配置项
- **接口设计**: 提供清晰的 API, 支持灵活调用
- **扩展性**: 设计时考虑未来可能的扩展需求

#### #### 4. 测试覆盖

- **单元测试**: 为核心函数编写全面的单元测试
- **集成测试**: 测试完整的调用流程
- **边界测试**: 覆盖各种边界情况
- **性能测试**: 评估在大数据量下的性能表现

#### #### 5. 代码质量

- **可读性**: 使用清晰的命名和注释
- **模块化**: 将复杂问题分解为可管理的模块
- **代码复用**: 提取公共功能为可复用组件
- **文档完善**: 提供详细的使用说明和 API 文档

### ## 八、面试要点

#### #### 1. 算法本质理解

- **选择问题本质**: 01 背包是典型的选择问题，每个物品有选或不选两种状态
- **动态规划思想**: 通过状态定义和转移方程，避免重复计算
- **贪心 vs 动态规划**: 解释为什么贪心算法不适用（物品不可分割的情况下）

#### #### 2. 状态转移分析

- **状态定义的思考过程**: 如何想到定义  $dp[i][j]$  表示前  $i$  个物品容量为  $j$  时的最大价值
- **转移方程推导**: 基于选或不选的决策推导出转移方程
- **状态优化思路**: 从二维到一维的优化过程和数学证明

#### #### 3. 空间优化原理

- **倒序遍历的必要性**: 为什么 01 背包需要逆序遍历容量
- **滚动数组的工作原理**: 如何复用一维数组来存储状态
- **不同背包问题的遍历顺序差异**: 01 背包 vs 完全背包的遍历顺序区别

#### #### 4. 变种问题转化能力

- **模型抽象能力**: 如何将实际问题抽象为背包模型
- **多维度扩展**: 从一维到多维费用的扩展思路
- **目标函数转化**: 最大化、最小化、计数、判断可行性等不同目标的处理

#### #### 5. 复杂度分析深度

- **时间复杂度详细计算**: 为什么是  $O(N \times W)$ ，常数项的影响
- **空间复杂度优化路径**: 从  $O(N \times W)$  到  $O(W)$  的优化过程
- **大数据量处理思路**: 当  $W$  非常大时的替代算法考虑

#### #### 6. 代码实现细节

- **边界条件处理**: 如何正确初始化  $dp$  数组
- **数组越界防护**: 如何避免数组访问越界
- **效率优化技巧**: 如预处理、剪枝等优化手段

#### #### 7. 问题迁移能力

- **相似问题识别**: 能够快速识别背包问题的变形题
- **算法迁移应用**: 将背包思想应用到新的问题场景
- **跨语言实现差异**: 不同编程语言实现时的注意事项

## ## 九、常见误区

#### #### 1. 遍历顺序错误

- **01 背包逆序遍历**: 容量必须逆序遍历，否则会导致物品被重复选择
- **完全背包正序遍历**: 容量需要正序遍历，允许物品被多次选择
- **分组背包顺序**: 组、容量、组内物品的三重循环顺序不能随意调换

#### #### 2. 状态转移方程错误

- **边界条件遗漏**: 未考虑物品重量超过当前容量的情况

- **\*\*初始化错误\*\*:** 未正确初始化 dp 数组, 如可行性问题初始化为 false
- **\*\*转移逻辑错误\*\*:** 混淆最大/最小/计数等不同类型问题的转移方程

#### #### 3. 空间优化错误

- **\*\*滚动数组使用不当\*\*:** 在多维背包问题中未正确应用滚动数组
- **\*\*越界访问\*\*:** 未检查数组索引的有效性
- **\*\*状态覆盖\*\*:** 在需要保留之前状态时错误地进行了覆盖

#### #### 4. 问题转化错误

- **\*\*数学推导错误\*\*:** 如目标和问题中的公式推导错误
- **\*\*条件判断遗漏\*\*:** 如分割等和子集问题中未检查总和是否为偶数
- **\*\*模型抽象偏差\*\*:** 错误地将问题抽象为不适合的背包模型

#### #### 5. 性能优化误区

- **\*\*过度优化\*\*:** 在不必要的的情况下过度追求优化
- **\*\*忽略实际约束\*\*:** 未考虑问题的实际约束条件
- **\*\*优化方向错误\*\*:** 选择了错误的优化方向, 如时间换空间还是空间换时间

### ## 十、扩展应用

#### #### 1. 与机器学习的联系

- **\*\*特征选择\*\*:** 在特征选择中, 选择重要特征可以建模为 01 背包问题
- **\*\*模型压缩\*\*:** 神经网络剪枝可以看作选择重要神经元的背包问题
- **\*\*资源分配\*\*:** 在强化学习中, 资源分配问题可以用背包模型解决
- **\*\*集成学习\*\*:** 选择基模型的问题可以转化为背包问题

#### #### 2. 与图像处理的联系

- **\*\*图像分割\*\*:** 能量最小化分割问题可以用背包模型近似
- **\*\*特征匹配\*\*:** 在资源限制下选择最优匹配点的问题
- **\*\*压缩感知\*\*:** 信号重构中的采样点选择问题

#### #### 3. 与自然语言处理的联系

- **\*\*文本摘要\*\*:** 选择重要句子生成摘要的问题
- **\*\*关键词提取\*\*:** 在预算限制下选择最重要的关键词
- **\*\*机器翻译\*\*:** 词汇选择和资源分配问题

#### #### 4. 工程实践应用

- **\*\*项目管理\*\*:** 在时间和资源限制下选择最优项目组合
- **\*\*投资组合优化\*\*:** 在风险约束下选择最优投资组合
- **\*\*资源调度\*\*:** 云计算中的资源分配问题
- **\*\*供应链优化\*\*:** 库存管理和订单选择问题

#### #### 5. 与其他算法的结合

- **\*\*背包+图论\*\*:** 如最短路径中的资源限制问题
- **\*\*背包+数论\*\*:** 模数约束下的背包问题
- **\*\*背包+贪心\*\*:** 混合策略解决复杂优化问题
- **\*\*背包+搜索\*\*:** 在组合优化中的应用

通过系统学习和练习这些 01 背包问题，可以深入理解动态规划的思想，提高算法设计和问题解决能力。

## ## 十一、具体问题实现

### #### 15. LeetCode 416. 分割等和子集

**\*\*核心思路\*\*:** 将问题转化为「01 背包」问题，判断是否能从数组中选择一些数字，使其和恰好等于整个数组和的一半。

**\*\*实现要点\*\*:** 使用 1D DP 数组优化空间，逆序遍历容量避免重复计算。

**\*\*时间复杂度\*\*:**  $O(n * \text{target})$ ，其中  $n$  是数组长度， $\text{target}$  是数组和的一半。

**\*\*空间复杂度\*\*:**  $O(\text{target})$ ，使用 1D DP 数组。

### #### 16. LeetCode 494. 目标和

**\*\*核心思路\*\*:** 将问题转化为「01 背包」问题，通过数学推导，找到和为特定值的子集数目。

**\*\*实现要点\*\*:** 利用  $\text{sum\_pos} = (\text{sum} + \text{target}) / 2$  将问题转化为统计满足特定和的子集数目。

**\*\*时间复杂度\*\*:**  $O(n * \text{target})$ ，其中  $n$  是数组长度， $\text{target}$  是转换后的目标和。

**\*\*空间复杂度\*\*:**  $O(\text{target})$ ，使用 1D DP 数组。

### #### 17. LeetCode 474. 一和零

**\*\*核心思路\*\*:** 将问题转化为「二维费用 01 背包」问题，每个字符串的 0 和 1 数量作为两个维度的费用。

**\*\*实现要点\*\*:** 使用二维 DP 数组， $dp[i][j]$  表示最多使用  $i$  个 0 和  $j$  个 1 时能组成的大子集长度。

**\*\*时间复杂度\*\*:**  $O(1 * m * n)$ ，其中 1 是字符串数组的长度， $m$  和  $n$  是背包的两个容量维度。

**\*\*空间复杂度\*\*:**  $O(m * n)$ ，使用二维 DP 数组。

### #### 18. LeetCode 879. 盈利计划

**\*\*核心思路\*\*:** 将问题转化为「三维费用 01 背包」问题，分别考虑员工数量、利润要求和工作数量三个维度。

**\*\*实现要点\*\*:** 使用  $dp[j][k]$  表示使用  $j$  个员工，获得至少  $k$  的利润的方案数，通过空间优化减少维度。

**\*\*时间复杂度\*\*:**  $O(N * \text{minProfit} * n)$ ，其中  $N$  是工作数量， $n$  是员工人数， $\text{minProfit}$  是最小利润要求。

**\*\*空间复杂度\*\*:**  $O(n * \text{minProfit})$ ，使用二维 DP 数组。

### #### 19. Code43 零钱兑换 (Coin Change)

**\*\*题目链接\*\*:** [LeetCode 322. 零钱兑换] (<https://leetcode.cn/problems/coin-change/>)

**\*\*题目难度\*\*:** Medium

**\*\*核心算法\*\*:** 动态规划，完全背包问题

**\*\*实现语言\*\*:** Java, C++, Python (均已完成)

**\*\*题目大意\*\*:** 给你一个整数数组  $\text{coins}$ ，表示不同面额的硬币；以及一个整数  $\text{amount}$ ，表示总金额。计算并返回可以凑成总金额所需的最少的硬币个数。如果没有任何一种硬币组合能组成总金额，返回 -1。

**\*\*核心思路\*\*:** 这是一个典型的完全背包问题，每种硬币可以无限次使用，目标是找出凑成总金额所需的最少硬币个数。

**\*\*实现要点\*\*:** 使用  $dp[i]$  表示凑成金额  $i$  所需的最少硬币个数，状态转移方程为  $dp[i] = \min(dp[i], dp[i - coin] + 1)$ ，需要正序遍历金额以允许重复使用硬币。

**\*\*时间复杂度\*\*:**  $O(amount * n)$ ，其中  $amount$  是总金额， $n$  是硬币种类数。

**\*\*空间复杂度\*\*:**  $O(amount)$ ，使用一维 DP 数组。

**\*\*扩展方法\*\*:**

1. **基础 DP 方法**: 标准的完全背包问题解法
2. **替代遍历顺序**: 先遍历金额，再遍历硬币
3. **广度优先搜索 (BFS)**: 将问题视为图的最短路径问题
4. **贪心+回溯**: 通过贪心策略优化回溯过程
5. **记忆化搜索**: 优化的回溯方法
6. **优化策略**: 提前过滤掉大于  $amount$  的硬币，提高效率

#### ### 20. LeetCode 518. 零钱兑换 II

**\*\*核心思路\*\*:** 这是一个完全背包问题的计数变种，目标是计算凑成总金额的硬币组合数，每种硬币可以无限次使用。

**\*\*实现要点\*\*:** 使用  $dp[i]$  表示凑成金额  $i$  的硬币组合数，通过先遍历硬币再遍历金额来确保计算的是组合数而非排列数。

**\*\*时间复杂度\*\*:**  $O(amount * n)$ ，其中  $amount$  是总金额， $n$  是硬币种类数。

**\*\*空间复杂度\*\*:**  $O(amount)$ ，使用一维 DP 数组。

#### ### 21. LeetCode 416. 分割等和子集

**\*\*核心思路\*\*:** 将问题转化为 01 背包问题，判断是否存在一个子集，其和等于数组总和的一半。

**\*\*实现要点\*\*:** 使用  $dp[i]$  表示是否可以组成和为  $i$  的子集，逆序遍历容量以确保每个元素只使用一次。

**\*\*时间复杂度\*\*:**  $O(n * target)$ ，其中  $n$  是数组长度， $target$  是数组总和的一半。

**\*\*空间复杂度\*\*:**  $O(target)$ ，使用一维 DP 数组。

#### ### 22. LeetCode 1049. 最后一块石头的重量 II

**\*\*核心思路\*\*:** 将石头分成两堆，使两堆重量尽可能接近，转化为 01 背包问题，寻找不超过总重量一半的最大子集和。

**\*\*实现要点\*\*:** 使用  $dp[i]$  表示是否可以组成和为  $i$  的石头堆，最终结果为总重量减去两倍的最大可达到和。

**\*\*时间复杂度\*\*:**  $O(n * target)$ ，其中  $n$  是石头数量， $target$  是总重量的一半。

**\*\*空间复杂度\*\*:**  $O(target)$ ，使用一维 DP 数组。

#### ### 23. LeetCode 474. 一和零

**\*\*核心思路\*\*:** 这是一个多维背包问题，我们需要同时考虑两种资源限制：0 的数量和 1 的数量。每个字符串相当于一个物品，占用的空间是它包含的 0 和 1 的数量，价值为 1。

**\*\*实现要点\*\*:** 使用二维  $dp$  数组  $dp[i][j]$  表示使用  $i$  个 0 和  $j$  个 1 时可以选择的最大字符串数量，通过逆序遍历两个维度避免重复选择同一物品。

**\*\*时间复杂度\*\*:**  $O(l * m * n)$ ，其中  $l$  是字符串数组的长度， $m$  和  $n$  是给定的整数。

**\*\*空间复杂度\*\*:**  $O(m * n)$ ，使用二维 DP 数组。

#### #### 24. LeetCode 377. 组合总和 IV

**\*\*核心思路\*\*:** 这是一个完全背包问题的排列变种，需要计算总和为目标值的所有排列数，顺序不同的序列视为不同的组合。

**\*\*实现要点\*\*:** 为了计算排列数而非组合数，需要先遍历容量 (target) 再遍历物品 (nums 数组)，使用  $dp[i]$  表示总和为  $i$  的元素组合个数。

**\*\*时间复杂度\*\*:**  $O(target * n)$ ，其中  $n$  是  $nums$  数组的长度。

**\*\*空间复杂度\*\*:**  $O(target)$ ，使用一维 DP 数组。

#### #### 25. 单词拆分 (Word Break)

**\*\*核心思路\*\*:** 这是一个完全背包问题的变种，判断是否能用字典中的单词（可重复使用）拼接成目标字符串。

**\*\*实现要点\*\*:** 使用  $dp[i]$  表示字符串前  $i$  个字符是否可以被拆分，对于每个位置  $i$ ，检查所有  $j < i$ ，如果  $dp[j]$  为 true 且子串  $s[j:i]$  在字典中，则  $dp[i] = \text{true}$ 。

**\*\*时间复杂度\*\*:**  $O(n^3)$ ，其中  $n$  是字符串长度，因为每次检查子串需要  $O(n)$  时间。

**\*\*空间复杂度\*\*:**  $O(n)$ ，使用一维 DP 数组。

**\*\*优化技巧\*\*:** 将字典转换为哈希集合以提高查找效率，可以记录字典中单词的最大长度减少不必要的子串检查。

**\*\*其他实现方法\*\*:** DFS+记忆化、BFS、Trie 树优化

#### #### 26. 零钱兑换 II (Coin Change 2)

**\*\*核心思路\*\*:** 这是一个完全背包问题，计算凑成总金额的硬币组合数，每种硬币可以使用无限次。

**\*\*实现要点\*\*:** 使用  $dp[i]$  表示凑成总金额  $i$  的硬币组合数，状态转移方程为  $dp[i] += dp[i - \text{coin}]$  (当  $i \geq \text{coin}$  时)。为了计算组合数而非排列数，需要先遍历硬币再遍历金额。

**\*\*时间复杂度\*\*:**  $O(amount * n)$ ，其中  $amount$  是总金额， $n$  是硬币种类数。

**\*\*空间复杂度\*\*:**  $O(amount)$ ，使用一维 DP 数组。

**\*\*关键技巧\*\*:** 完全背包问题中，正序遍历容量允许物品被重复使用；先遍历物品再遍历容量确保计算的是组合数而非排列数。

#### #### 27. 目标和 (Target Sum)

**\*\*核心思路\*\*:** 这是一个 0-1 背包问题的变种，通过添加正负号将问题转化为找一个子集和，使得该子集和与其余元素和的差等于目标值。

**\*\*实现要点\*\*:** 将问题转化为求子集和为 ( $\text{subsetSum}$ ) 的数目，其中  $\text{subsetSum} = (\text{sum} + \text{target}) / 2$ ，使用  $dp[i]$  表示和为  $i$  的子集数目。

**\*\*时间复杂度\*\*:**  $O(n * \text{subsetSum})$ ，其中  $n$  是数组长度， $\text{subsetSum}$  是转化后的目标子集和。

**\*\*空间复杂度\*\*:**  $O(\text{subsetSum})$ ，使用一维 DP 数组。

**\*\*关键技巧\*\*:** 问题转化是关键，需要判断  $\text{sum} + \text{target}$  是否为非负偶数，否则无解。

#### #### 28. 分割等和子集 (Partition Equal Subset Sum)

**\*\*核心思路\*\*:** 这是一个 0-1 背包问题的经典应用，将原问题转化为是否存在一个子集，其和等于数组总和的一半。

**\*\*实现要点\*\*:** 首先检查数组总和是否为偶数，然后使用  $dp[i]$  表示是否可以组成和为  $i$  的子集，状态转移方程为  $dp[i] = dp[i] || dp[i - \text{num}]$ 。

**\*\*时间复杂度\*\*:**  $O(n * \text{target})$ , 其中  $n$  是数组长度,  $\text{target}$  是数组和的一半。

**\*\*空间复杂度\*\*:**  $O(\text{target})$ , 使用一维 DP 数组。

**\*\*关键技巧\*\*:**

1. 使用逆序遍历容量来避免重复使用同一个元素
2. 提前剪枝: 如果数组总和是奇数或最大元素大于目标和, 可以直接返回 false
3. 可以使用位运算进一步优化, 每个二进制位表示是否可以组成对应索引的和

#### #### 29. 最后一块石头的重量 II (Last Stone Weight II)

**\*\*核心思路\*\*:** 这是一个 0-1 背包问题的变种, 将石头分成两组, 使得两组的重量差最小, 这样最后剩下的石头重量也会最小。

**\*\*实现要点\*\*:** 将问题转化为找到一组石头, 使其和尽可能接近总重量的一半, 使用  $dp[i]$  表示是否可以组成和为  $i$  的子集。

**\*\*时间复杂度\*\*:**  $O(n * \text{target})$ , 其中  $n$  是石头数量,  $\text{target}$  是总重量的一半。

**\*\*空间复杂度\*\*:**  $O(\text{target})$ , 使用一维 DP 数组。

**\*\*关键技巧\*\*:**

1. 最终结果是总重量减去两倍的最大可达子集和
2. 逆序遍历容量确保每个石头只能使用一次
3. 位运算优化可以更高效地表示所有可能的子集和

#### #### 30. 一和零 (Ones and Zeroes)

**\*\*核心思路\*\*:** 这是一个二维费用的 0-1 背包问题, 每个字符串作为物品, 其 0 和 1 的个数作为两个维度的重量,  $m$  和  $n$  作为两个维度的背包容量。

**\*\*实现要点\*\*:** 使用  $dp[i][j]$  表示最多使用  $i$  个 0 和  $j$  个 1 时, 可以组成的大子集长度, 状态转移方程为  $dp[i][j] = \max(dp[i][j], dp[i-zeroes][j-ones] + 1)$ 。

**\*\*时间复杂度\*\*:**  $O(1 * m * n)$ , 其中 1 是字符串数组的长度,  $m$  和  $n$  是给定的两个整数。

**\*\*空间复杂度\*\*:**  $O(m * n)$ , 使用二维 DP 数组。

**\*\*关键技巧\*\*:**

1. 需要逆序遍历两个维度, 以确保每个字符串只被选择一次
2. 可以预处理每个字符串的 0 和 1 的个数, 提高效率
3. 可以提前剪枝, 过滤掉那些 0 或 1 的个数超过限制的字符串

#### #### 31. 零钱兑换 (Coin Change)

**\*\*核心思路\*\*:** 这是一个典型的完全背包问题, 硬币可以重复使用, 目标是找到凑成总金额所需的最少硬币个数。

**\*\*实现要点\*\*:** 使用  $dp[i]$  表示凑成金额  $i$  所需的最少硬币个数, 状态转移方程为  $dp[i] = \min(dp[i], dp[i-coin] + 1)$ , 需要正序遍历金额以允许重复使用硬币。

**\*\*时间复杂度\*\*:**  $O(\text{amount} * n)$ , 其中  $n$  是硬币的种类数。

**\*\*空间复杂度\*\*:**  $O(\text{amount})$ , 使用一维 DP 数组。

**\*\*关键技巧\*\*:**

1. 正序遍历金额 (完全背包的特点)
2. 初始化为一个不可能达到的值 (如  $\text{amount} + 1$ )
3. 除了动态规划, 还可以使用 BFS 或贪心+DFS 剪枝的方法解决
4. 可以提前排序硬币, 在某些情况下进行剪枝优化

#### #### 32. 零钱兑换 II (Coin Change II)

**\*\*核心思路\*\*:** 这是一个完全背包问题的变种，目标不是求最少硬币个数，而是求凑成总金额的不同组合数。

**\*\*实现要点\*\*:** 使用  $dp[i]$  表示凑成金额  $i$  的不同组合数，状态转移方程为  $dp[i] += dp[i-coin]$ ，必须将硬币循环放在外层，金额循环放在内层，以避免计算不同顺序的重复组合。

**\*\*时间复杂度\*\*:**  $O(amount * n)$ ，其中  $n$  是硬币的种类数。

**\*\*空间复杂度\*\*:**  $O(amount)$ ，使用一维 DP 数组。

**\*\*关键技巧\*\*:**

1. 必须将硬币放在外层循环，金额放在内层循环，以避免计算重复的组合
2. 初始状态  $dp[0] = 1$ ，表示凑成金额 0 只有一种方式（不使用任何硬币）
3. 注意整数溢出问题，在 C++ 等语言中可能需要使用 long long 类型
4. 递归实现需要进行排序并使用剪枝策略，避免重复计算

#### #### 33. 组合总和 IV (Combination Sum IV)

**\*\*核心思路\*\*:** 这是一个与完全背包相关但关注排列而非组合的问题，顺序不同的序列被视为不同的组合。

**\*\*实现要点\*\*:** 使用  $dp[i]$  表示凑成目标值  $i$  的不同排列数，状态转移方程为  $dp[i] += dp[i-num]$ ，与零钱兑换 II 不同的是，这里需要将目标值循环放在外层，数组元素循环放在内层，以考虑不同顺序的排列。

**\*\*时间复杂度\*\*:**  $O(target * n)$ ，其中  $n$  是数组  $nums$  的长度。

**\*\*空间复杂度\*\*:**  $O(target)$ ，使用一维 DP 数组。

**\*\*关键技巧\*\*:**

1. 必须将目标值放在外层循环，数组元素放在内层循环，以计算所有可能的排列
2. 初始状态  $dp[0] = 1$ ，表示凑成目标值 0 只有一种方式（不选择任何数字）
3. 对于较大的 target，需要注意整数溢出问题
4. 可以通过排序数组进行剪枝优化，当当前元素大于剩余目标值时提前退出内层循环

#### #### 34. 掷骰子的 N 种方法 (Number of Dice Rolls With Target Sum)

**\*\*核心思路\*\*:** 这是一个典型的分组背包问题，每个骰子可以看作一组，每组有  $k$  个选项（1 到  $k$  的点数），我们需要从每组中选择一个选项，使得它们的总和等于 target。

**\*\*实现要点\*\*:** 使用  $dp[i][j]$  表示使用  $i$  个骰子能得到点数和为  $j$  的方案数，状态转移方程为  $dp[i][j] = \sum(dp[i-1][j-m])$ ，其中  $m$  从 1 到  $k$  且  $j-m \geq i-1$ 。可以通过滚动数组优化空间复杂度。

**\*\*时间复杂度\*\*:**  $O(n * k * target)$ ，其中  $n$  是骰子数量， $k$  是每个骰子的面数。

**\*\*空间复杂度\*\*:**  $O(n * target)$ ，优化后为  $O(target)$ 。

**\*\*关键技巧\*\*:**

1. 注意边界条件：当 target 小于  $n$  或大于  $n*k$  时，直接返回 0
2. 初始状态  $dp[0][0] = 1$ ，表示使用 0 个骰子得到点数和为 0 只有一种方式
3. 计算时需要对结果取模  $10^9+7$ ，避免整数溢出
4. 使用记忆化搜索也可以解决此问题，但动态规划方法更高效

#### #### 35. 最后一块石头的重量 II (Last Stone Weight II)

**\*\*核心思路\*\*:** 这是一个 0-1 背包问题的变种，目标是将石头分成两组，使得两组的重量差最小。可以转化为寻找一个子集，其总重量尽可能接近总重量的一半。

**\*\*实现要点\*\*:** 使用  $dp[j]$  表示是否能组成重量为  $j$  的子集，状态转移方程为  $dp[j] = dp[j] || dp[j - stones[i]]$ 。最后找到最大的  $j$ ，使得  $dp[j]$  为 true 且  $j$  不超过总重量的一半。

**\*\*时间复杂度\*\*:**  $O(n * target)$ , 其中  $n$  是石头数量,  $target$  是总重量的一半。

**\*\*空间复杂度\*\*:**  $O(target)$ , 使用一维 DP 数组。

**\*\*关键技巧\*\*:**

1. 初始状态  $dp[0] = \text{true}$ , 表示空子集的重量为 0 是可以组成的
2. 在更新 DP 数组时需要逆序遍历重量, 避免重复使用同一块石头
3. 最终结果为总重量减去两倍的最大子集和
4. 对于较大的输入, 可以使用位集合来优化空间, 但在实际实现中一维布尔数组更为直观

### ### 36. 一和零 (Ones and Zeroes)

**\*\*核心思路\*\*:** 这是一个二维费用的 0-1 背包问题。每个字符串可以看作是一个物品, 它有两个费用 ( $0$  的数量和  $1$  的数量), 我们的背包有两个容量限制 (最多可以使用  $m$  个  $0$  和  $n$  个  $1$ ), 目标是选择尽可能多的物品。

**\*\*实现要点\*\*:** 使用  $dp[i][j]$  表示使用  $i$  个  $0$  和  $j$  个  $1$  时, 最多可以选择的字符串数量。状态转移方程为  $dp[i][j] = \max(dp[i][j], dp[i - \text{zeros}][j - \text{ones}] + 1)$ , 其中  $\text{zeros}$  和  $\text{ones}$  是当前字符串的  $0$  和  $1$  的数量。

**\*\*时间复杂度\*\*:**  $O(1 * m * n)$ , 其中  $1$  是字符串数组的长度。

**\*\*空间复杂度\*\*:**  $O(m * n)$ , 使用二维 DP 数组。

**\*\*关键技巧\*\*:**

1. 在更新 DP 数组时需要逆序遍历两个维度, 避免重复使用同一个字符串
2. 可以预先统计所有字符串中  $0$  和  $1$  的数量, 避免重复计算
3. 初始状态时所有  $dp[i][j]$  都为  $0$ , 不需要特别初始化
4. 也可以使用三维 DP 数组 ( $dp[k][i][j]$ ) 表示前  $k$  个字符串的状态, 更容易理解但空间效率较低

### ### 37. 分割等和子集 (Partition Equal Subset Sum)

**\*\*核心思路\*\*:** 这是一个 0-1 背包问题的变种。问题可以转化为是否存在一个子集, 其和等于整个数组和的一半。

**\*\*实现要点\*\*:** 使用  $dp[j]$  表示是否能组成和为  $j$  的子集。状态转移方程为  $dp[j] = dp[j] || dp[j - \text{nums}[i]]$ 。初始状态  $dp[0] = \text{true}$ , 表示空子集的和为  $0$  是可以组成的。

**\*\*时间复杂度\*\*:**  $O(n * target)$ , 其中  $n$  是数组长度,  $target$  是数组和的一半。

**\*\*空间复杂度\*\*:**  $O(target)$ , 使用一维 DP 数组。

**\*\*关键技巧\*\*:**

1. 如果数组总和是奇数, 直接返回  $\text{false}$ , 因为无法分成两个和相等的子集
2. 如果数组中最大的数字大于总和的一半, 也直接返回  $\text{false}$
3. 在更新 DP 数组时需要逆序遍历, 避免重复使用同一个元素
4. 可以使用位操作优化, 对于元素值不大的情况效率更高

### ### 38. 单词拆分 (Word Break)

**\*\*核心思路\*\*:** 这是一个完全背包问题的变种。我们可以将字符串  $s$  看作是背包, 将字典中的单词看作是物品。问题转化为: 是否可以从字典中选择一些单词 (可以重复选择), 使得它们的拼接恰好等于字符串  $s$ 。

**\*\*实现要点\*\*:** 使用  $dp[i]$  表示字符串  $s$  的前  $i$  个字符是否可以被拆分成字典中的单词。状态转移方程为对于每个  $i$ , 我们检查所有  $j < i$ , 如果  $dp[j]$  为  $\text{true}$  且  $s.\text{substring}(j, i)$  在字典中, 则  $dp[i]$  为  $\text{true}$ 。初始状态  $dp[0] = \text{true}$ , 表示空字符串可以被拆分。

**\*\*时间复杂度\*\*:**  $O(n^3)$ , 其中  $n$  是字符串  $s$  的长度。

**\*\*空间复杂度\*\*:**  $O(n + m)$ , 其中  $m$  是字典中所有单词的字符总数。

## \*\*关键技巧\*\*:

1. 将字典转换为哈希集合，提高查找效率
2. 限制  $j$  的范围为最大单词长度，避免不必要的检查
3. 可以使用动态规划、递归+记忆化、BFS 或前缀树来实现
4. 在动态规划中，只要找到一个可行的拆分方式就可以提前结束内层循环

### ### 39. 单词拆分 II (Word Break II)

**\*\*核心思路\*\*:** 这是单词拆分的进阶版本，要求返回所有可能的拆分方案。这是一个完全背包问题的变种，同时也是一个组合问题。

**\*\*实现要点\*\*:** 使用递归+记忆化搜索来找出所有可能的拆分方案。对于每个位置  $i$ ，我们尝试所有可能的单词，如果  $s.substring(i, j)$  在字典中，我们递归处理剩余部分，然后将当前单词与剩余部分的结果组合。

**\*\*时间复杂度\*\*:**  $O(n^2 * 2^n)$ ，其中  $n$  是字符串  $s$  的长度。在最坏情况下，每个字符之间都可以拆分，会产生  $2^{(n-1)}$  种拆分方式。

**\*\*空间复杂度\*\*:**  $O(n^2)$ ，用于存储记忆化缓存。

## \*\*关键技巧\*\*:

1. 先使用动态规划检查字符串是否可以拆分，如果不能拆分直接返回空列表，避免不必要的递归
2. 使用记忆化缓存避免重复计算
3. 限制搜索范围为最大单词长度，避免不必要的检查
4. 可以使用回溯算法或动态规划来实现，存储所有可能的拆分方案

### ### 40. 完全平方数 (Perfect Squares)

**\*\*核心思路\*\*:** 这是一个完全背包问题。我们可以将问题转化为：使用最少数量的物品（每个物品是一个完全平方数），恰好装满容量为  $n$  的背包。

**\*\*实现要点\*\*:** 使用  $dp[i]$  表示和为  $i$  的完全平方数的最少数量。状态转移方程为  $dp[i] = \min(dp[i], dp[i - j^2] + 1)$ ，其中  $j^2 \leq i$ 。初始状态  $dp[0] = 0$ ，表示和为 0 的完全平方数的最少数量为 0。

**\*\*时间复杂度\*\*:**  $O(n * \sqrt{n})$ ，其中  $n$  是给定的整数。

**\*\*空间复杂度\*\*:**  $O(n)$ ，使用一维 DP 数组。

## \*\*关键技巧\*\*:

1. 预先生成所有可能的完全平方数，避免重复计算
2. 可以使用广度优先搜索 (BFS) 将问题转化为最短路径问题
3. 利用数学定理（拉格朗日四平方定理）进行优化，最多只需 4 个平方数
4. 对于 DP 实现，可以从目标数或完全平方数两个角度考虑状态转移

### ### 41. 零钱兑换 II (Coin Change II)

**\*\*核心思路\*\*:** 这是一个完全背包问题的变种。我们需要计算使用不同面额的硬币（可以重复使用）恰好凑出总金额的方式数。

**\*\*实现要点\*\*:** 使用  $dp[j]$  表示凑成总金额  $j$  的硬币组合数。状态转移方程为  $dp[j] += dp[j - coin]$ ，其中  $coin$  是当前硬币的面额。初始状态  $dp[0] = 1$ ，表示凑成总金额 0 的方式有一种（不使用任何硬币）。

**\*\*时间复杂度\*\*:**  $O(amount * n)$ ，其中  $amount$  是总金额， $n$  是硬币的种类数。

**\*\*空间复杂度\*\*:**  $O(amount)$ ，使用一维 DP 数组。

## \*\*关键技巧\*\*:

1. 先遍历硬币，再遍历金额，确保计算的是组合数而不是排列数
2. 提前过滤掉大于  $amount$  的硬币，优化计算

3. 可以通过回溯或动态规划方法打印出所有可能的组合
4. 在实际应用中，可以使用模运算避免整数溢出

#### #### 42. 目标和 (Target Sum)

**\*\*核心思路\*\*:** 这是一个 0-1 背包问题的变种。将问题转化为找到一个子集 P，使得  $\text{sum}(P) - \text{sum}(N) = \text{target}$ ，其中 N 是数组中不在 P 中的元素。通过数学推导，可以将问题转化为子集和问题。

**\*\*实现要点\*\*:** 计算目标和:  $\text{sum}(P) = (\text{total\_sum} + \text{target}) / 2$ ，然后使用动态规划计算有多少个子集的和等于这个目标值。使用  $\text{dp}[j]$  表示和为 j 的子集数目。

**\*\*时间复杂度\*\*:**  $O(n * \text{sum})$ ，其中 n 是数组的长度，sum 是数组元素的和。

**\*\*空间复杂度\*\*:**  $O(\text{sum})$ ，使用一维 DP 数组。

**\*\*关键技巧\*\*:**

1. 使用数学推导将问题转化为标准的子集和问题
2. 注意处理特殊情况（无解的条件判断）
3. 对于包含 0 的数组可以进行优化
4. 可以使用回溯法、记忆化递归或 BFS 等多种方法实现

=====

[代码文件]

=====

文件: Code01\_01Knapsack.cpp

=====

```
// 01 背包问题 (模板)
//
// 问题描述:
// 给定一个正数 t，表示背包的容量
// 有 m 个货物，每个货物可以选择一次
// 每个货物有自己的体积 costs[i] 和价值 values[i]
// 返回在不超过总容量的情况下，怎么挑选货物能达到价值最大
// 返回最大的价值
//
// 解题思路:
// 01 背包问题是动态规划中的经典问题，每个物品只能选择一次。
// 状态定义: dp[i][j] 表示前 i 个物品，背包容量为 j 时能获得的最大价值
// 状态转移方程:
// - 不选择第 i 个物品: dp[i][j] = dp[i-1][j]
// - 选择第 i 个物品: dp[i][j] = dp[i-1][j-cost[i]] + val[i] (前提是 j >= cost[i])
// - dp[i][j] = max(dp[i-1][j], dp[i-1][j-cost[i]] + val[i])
//
// 优化: 通过观察状态转移方程，发现 dp[i][j] 只依赖于 dp[i-1] 这一行，
// 所以可以用一维数组优化空间复杂度，但需要倒序遍历背包容量以确保每个物品只使用一次。
//
// 时间复杂度: O(n * t)，其中 n 是物品数量，t 是背包容量
```

```

// 空间复杂度: O(t)
//
// 测试链接 : https://www.luogu.com.cn/problem/P1048

// 全局常量
const int MAXM = 101;
const int MAXT = 1001;

// 全局变量
int cost[MAXM] = {0};
int val[MAXM] = {0};
int dp[MAXT] = {0};

int t = 0;
int n = 0;

// 严格位置依赖的动态规划
// n 个物品编号 1~n, 第 i 号物品的花费 cost[i]、价值 val[i]
// cost、val 数组是全局变量, 已经把数据读入了
// 时间复杂度: O(n * t), 空间复杂度: O(n * t)
int compute1() {
 // 创建二维 dp 数组
 // dp[i][j] 表示前 i 个物品, 背包容量为 j 时能获得的最大价值
 int dp_table[101][1001] = {0};

 // 遍历每个物品
 for (int i = 1; i <= n; i++) {
 // 遍历每个背包容量
 for (int j = 0; j <= t; j++) {
 // 不选择第 i 个物品
 dp_table[i][j] = dp_table[i - 1][j];

 // 选择第 i 个物品 (前提是背包容量足够)
 if (j - cost[i] >= 0) {
 // 要 i 号物品
 dp_table[i][j] = (dp_table[i][j] > dp_table[i - 1][j - cost[i]] + val[i]) ?
 dp_table[i][j] : dp_table[i - 1][j - cost[i]] + val[i];
 }
 }
 }

 // 返回前 n 个物品, 背包容量为 t 时能获得的最大价值
 return dp_table[n][t];
}

```

```
}
```

```
// 空间压缩版本
// 通过观察状态转移方程，发现 dp[i][j] 只依赖于 dp[i-1] 这一行
// 所以可以用一维数组优化空间复杂度
// 时间复杂度：O(n * t)，空间复杂度：O(t)
int compute2() {
 // 初始化 dp 数组
 for (int i = 0; i <= t; i++) {
 dp[i] = 0;
 }

 // 遍历物品
 for (int i = 1; i <= n; i++) {
 // 倒序遍历背包容量，确保每个物品只使用一次
 // 如果正序遍历，前面的状态会被更新，导致一个物品被多次使用
 for (int j = t; j >= cost[i]; j--) {
 // 状态转移方程：
 // dp[j] = max(不选择当前物品, 选择当前物品)
 // 不选择当前物品: dp[j] (保持原值)
 // 选择当前物品: dp[j - cost[i]] + val[i]
 int select = dp[j - cost[i]] + val[i];
 dp[j] = (dp[j] > select) ? dp[j] : select;
 }
 }

 // 返回背包容量为 t 时能获得的最大价值
 return dp[t];
}
```

```
// LeetCode 416. 分割等和子集 (Partition Equal Subset Sum)
// 题目描述：给定一个只包含正整数的非空数组，判断是否可以将这个数组分割成两个子集，
// 使得两个子集的元素和相等。
// 链接：https://leetcode.cn/problems/partition-equal-subset-sum/
// 解题思路：
// 1. 如果数组总和为奇数，则无法分割成两个相等的子集，返回 false
// 2. 如果数组总和为偶数，则问题转化为 01 背包问题：
// - 背包容量为 sum/2
// - 每个数字既是重量也是价值
// - 判断是否能装满背包
// 时间复杂度：O(n * sum)，其中 n 是数组长度，sum 是数组元素和
// 空间复杂度：O(sum)
bool canPartition(int* nums, int numsSize) {
```

```

// 计算数组总和
int sum = 0;
for (int i = 0; i < numsSize; i++) {
 sum += nums[i];
}

// 如果总和为奇数，无法分割成两个相等子集
if (sum % 2 == 1) {
 return false;
}

// 目标和为总和的一半
int target = sum / 2;

// dp[j] 表示是否能组成和为 j 的子集
bool dp_table[10001] = {false};
// 初始状态：和为 0 的子集总是存在（空集）
dp_table[0] = true;

// 遍历每个数字
for (int i = 0; i < numsSize; i++) {
 int num = nums[i];
 // 01 背包需要倒序遍历，确保每个物品只使用一次
 for (int j = target; j >= num; j--) {
 // 状态转移方程：
 // dp[j] = dp[j] || dp[j - num]
 // dp[j] 表示不选择当前数字能否组成和为 j 的子集
 // dp[j - num] 表示选择当前数字能否组成和为 j - num 的子集
 dp_table[j] = dp_table[j] || dp_table[j - num];
 }
}

// 返回是否能组成和为 target 的子集
return dp_table[target];
}

/*
 * 示例：
 * 输入：nums = [1, 5, 11, 5]
 * 输出：true
 * 解释：数组可以分割成 [1, 5, 5] 和 [11]。
 *
 * 输入：nums = [1, 2, 3, 5]

```

```
* 输出: false
* 解释: 数组不能分割成两个元素和相等的子集。
*
* 时间复杂度: O(n * sum), 其中 n 是数组长度, sum 是数组元素和
* 空间复杂度: O(sum)
*/
=====
```

文件: Code01\_01Knapsack.java

```
=====
package class073;

/**
 * 01 背包问题 (模板)
 *
 * 问题描述:
 * 给定一个正数 t, 表示背包的容量
 * 有 m 个货物, 每个货物可以选择一次
 * 每个货物有自己的体积 costs[i] 和价值 values[i]
 * 返回在不超过总容量的情况下, 怎么挑选货物能达到价值最大
 * 返回最大的价值
 *
 * 解题思路:
 * 01 背包问题是动态规划中的经典问题, 每个物品只能选择一次。
 * 状态定义: dp[i][j] 表示前 i 个物品, 背包容量为 j 时能获得的最大价值
 * 状态转移方程:
 * - 不选择第 i 个物品: dp[i][j] = dp[i-1][j]
 * - 选择第 i 个物品: dp[i][j] = dp[i-1][j-cost[i]] + val[i] (前提是 j >= cost[i])
 * - dp[i][j] = max(dp[i-1][j], dp[i-1][j-cost[i]] + val[i])
 *
 * 优化: 通过观察状态转移方程, 发现 dp[i][j] 只依赖于 dp[i-1] 这一行,
 * 所以可以用一维数组优化空间复杂度, 但需要倒序遍历背包容量以确保每个物品只使用一次。
 *
 * 时间复杂度: O(n * t), 其中 n 是物品数量, t 是背包容量
 * 空间复杂度: O(t)
 *
 * 测试链接 : https://www.luogu.com.cn/problem/P1048
 * 请同学们务必参考如下代码中关于输入、输出的处理
 * 这是输入输出处理效率很高的写法
 * 提交以下的所有代码, 并把主类名改成"Main", 可以直接通过
*/
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 Code01_01Knapsack {

 public static int MAXM = 101;

 public static int MAXT = 1001;

 // 物品的体积（成本）
 public static int[] cost = new int[MAXM];

 // 物品的价值
 public static int[] val = new int[MAXM];

 // DP 数组，用于空间优化版本
 public static int[] dp = new int[MAXT];

 public static int t, n;

 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 (in.nextToken() != StreamTokenizer.TT_EOF) {
 t = (int) in.nval;
 in.nextToken();
 n = (int) in.nval;
 for (int i = 1; i <= n; i++) {
 in.nextToken();
 cost[i] = (int) in.nval;
 in.nextToken();
 val[i] = (int) in.nval;
 }
 out.println(compute2());
 }
 out.flush();
 out.close();
 br.close();
 }

 private static int compute2() {
 if (t == 0 || n == 0) return 0;
 if (dp[t] != -1) return dp[t];
 int res = 0;
 for (int i = 1; i <= n; i++) {
 if (cost[i] > t) break;
 res = Math.max(res, val[i] + compute2(t - cost[i]));
 }
 dp[t] = res;
 return res;
 }
}
```

```

}

/**
 * 严格位置依赖的动态规划解法
 *
 * 解题思路:
 * 使用二维 DP 数组, dp[i][j] 表示前 i 个物品, 背包容量为 j 时能获得的最大价值
 * 状态转移方程:
 * - 不选择第 i 个物品: dp[i][j] = dp[i-1][j]
 * - 选择第 i 个物品: dp[i][j] = dp[i-1][j-cost[i]] + val[i] (前提是 j >= cost[i])
 * - dp[i][j] = max(dp[i-1][j], dp[i-1][j-cost[i]] + val[i])
 *
 * 时间复杂度: O(n * t)
 * 空间复杂度: O(n * t)
 *
 * @return 最大价值
 */
public static int compute1() {
 // dp[i][j] 表示前 i 个物品, 背包容量为 j 时能获得的最大价值
 int[][] dp = new int[n + 1][t + 1];

 // 遍历每个物品
 for (int i = 1; i <= n; i++) {
 // 遍历每个背包容量
 for (int j = 0; j <= t; j++) {
 // 不选择第 i 个物品
 dp[i][j] = dp[i - 1][j];

 // 选择第 i 个物品 (前提是背包容量足够)
 if (j - cost[i] >= 0) {
 // 要 i 号物品
 dp[i][j] = Math.max(dp[i][j], dp[i - 1][j - cost[i]] + val[i]);
 }
 }
 }

 // 返回前 n 个物品, 背包容量为 t 时能获得的最大价值
 return dp[n][t];
}

/**
 * 空间压缩版本
 *

```

```

* 解题思路:
* 通过观察状态转移方程, 发现 dp[i][j] 只依赖于 dp[i-1] 这一行
* 所以可以用一维数组优化空间复杂度
* 但需要倒序遍历背包容量以确保每个物品只使用一次
*
* 时间复杂度: O(n * t)
* 空间复杂度: O(t)
*
* @return 最大价值
*/
public static int compute2() {
 // 初始化 dp 数组
 Arrays.fill(dp, 0, t + 1, 0);

 // 遍历物品
 for (int i = 1; i <= n; i++) {
 // 倒序遍历背包容量, 确保每个物品只使用一次
 // 如果正序遍历, 前面的状态会被更新, 导致一个物品被多次使用
 for (int j = t; j >= cost[i]; j--) {
 // 状态转移方程:
 // dp[j] = max(不选择当前物品, 选择当前物品)
 // 不选择当前物品: dp[j] (保持原值)
 // 选择当前物品: dp[j - cost[i]] + val[i]
 dp[j] = Math.max(dp[j], dp[j - cost[i]] + val[i]);
 }
 }

 // 返回背包容量为 t 时能获得的最大价值
 return dp[t];
}

/**
* LeetCode 416. 分割等和子集 (Partition Equal Subset Sum)
*
* 题目描述: 给定一个只包含正整数的非空数组, 判断是否可以将这个数组分割成两个子集,
* 使得两个子集的元素和相等。
*
* 解题思路:
* 1. 如果数组总和为奇数, 则无法分割成两个相等的子集, 返回 false
* 2. 如果数组总和为偶数, 则问题转化为 01 背包问题:
* - 背包容量为 sum/2
* - 每个数字既是重量也是价值
* - 判断是否能装满背包

```

```

*
* 时间复杂度: O(n * sum), 其中 n 是数组长度, sum 是数组元素和
* 空间复杂度: O(sum)
*
* @param nums 正整数数组
* @return 是否可以分割成两个和相等的子集
*/
public static boolean canPartition(int[] nums) {
 // 计算数组总和
 int sum = 0;
 for (int num : nums) {
 sum += num;
 }

 // 如果总和为奇数, 无法分割成两个相等子集
 if ((sum & 1) == 1) {
 return false;
 }

 // 目标和为总和的一半
 int target = sum >> 1;

 // dp[j] 表示是否能组成和为 j 的子集
 boolean[] dp = new boolean[target + 1];
 // 初始状态: 和为 0 的子集总是存在 (空集)
 dp[0] = true;

 // 遍历每个数字
 for (int num : nums) {
 // 01 背包需要倒序遍历, 确保每个物品只使用一次
 for (int j = target; j >= num; j--) {
 // 状态转移方程:
 // dp[j] = dp[j] || dp[j - num]
 // dp[j] 表示不选择当前数字能否组成和为 j 的子集
 // dp[j - num] 表示选择当前数字能否组成和为 j-num 的子集
 dp[j] = dp[j] || dp[j - num];
 }
 }

 // 返回是否能组成和为 target 的子集
 return dp[target];
}

```

```
/*
 * 示例:
 * 输入: nums = [1, 5, 11, 5]
 * 输出: true
 * 解释: 数组可以分割成 [1, 5, 5] 和 [11]。
 *
 * 输入: nums = [1, 2, 3, 5]
 * 输出: false
 * 解释: 数组不能分割成两个元素和相等的子集。
 *
 * 时间复杂度: O(n * sum), 其中 n 是数组长度, sum 是数组元素和
 * 空间复杂度: O(sum)
 */

}
```

```
=====
文件: Code01_01Knapsack.py
=====

01 背包问题 (模板)
#
问题描述:
给定一个正数 t, 表示背包的容量
有 m 个货物, 每个货物可以选择一次
每个货物有自己的体积 costs[i] 和价值 values[i]
返回在不超过总容量的情况下, 怎么挑选货物能达到价值最大
返回最大的价值
#
解题思路:
01 背包问题是动态规划中的经典问题, 每个物品只能选择一次。
状态定义: dp[i][j] 表示前 i 个物品, 背包容量为 j 时能获得的最大价值
状态转移方程:
- 不选择第 i 个物品: dp[i][j] = dp[i-1][j]
- 选择第 i 个物品: dp[i][j] = dp[i-1][j-cost[i]] + val[i] (前提是 j >= cost[i])
- dp[i][j] = max(dp[i-1][j], dp[i-1][j-cost[i]] + val[i])
#
优化: 通过观察状态转移方程, 发现 dp[i][j] 只依赖于 dp[i-1] 这一行,
所以可以用一维数组优化空间复杂度, 但需要倒序遍历背包容量以确保每个物品只使用一次。
#
时间复杂度: O(n * t), 其中 n 是物品数量, t 是背包容量
空间复杂度: O(t)
```

```
测试链接 : https://www.luogu.com.cn/problem/P1048
```

```
import sys
```

```
全局变量
```

```
MAXM = 101
```

```
MAXT = 1001
```

```
物品的体积（成本）
```

```
cost = [0] * MAXM
```

```
物品的价值
```

```
val = [0] * MAXM
```

```
DP 数组，用于空间优化版本
```

```
dp = [0] * MAXT
```

```
t = 0
```

```
n = 0
```

```
严格位置依赖的动态规划
```

```
n 个物品编号 1~n，第 i 号物品的花费 cost[i]、价值 val[i]
```

```
cost、val 数组是全局变量，已经把数据读入了
```

```
时间复杂度: O(n * t)，空间复杂度: O(n * t)
```

```
def compute1():
```

```
 """
```

```
 严格位置依赖的动态规划解法
```

解题思路:

使用二维 DP 数组， $dp[i][j]$  表示前  $i$  个物品，背包容量为  $j$  时能获得的最大价值

状态转移方程:

- 不选择第  $i$  个物品:  $dp[i][j] = dp[i-1][j]$
- 选择第  $i$  个物品:  $dp[i][j] = dp[i-1][j-cost[i]] + val[i]$  (前提是  $j \geq cost[i]$ )
- $dp[i][j] = \max(dp[i-1][j], dp[i-1][j-cost[i]] + val[i])$

时间复杂度:  $O(n * t)$

空间复杂度:  $O(n * t)$

Returns:

最大价值

```
"""
```

```
创建二维 dp 数组
```

```
$dp[i][j]$ 表示前 i 个物品，背包容量为 j 时能获得的最大价值
```

```

dp = [[0 for _ in range(t + 1)] for _ in range(n + 1)]

遍历每个物品
for i in range(1, n + 1):
 # 遍历每个背包容量
 for j in range(t + 1):
 # 不选择第 i 个物品
 dp[i][j] = dp[i - 1][j]

 # 选择第 i 个物品 (前提是背包容量足够)
 if j - cost[i] >= 0:
 # 要 i 号物品
 dp[i][j] = max(dp[i][j], dp[i - 1][j - cost[i]] + val[i])

返回前 n 个物品，背包容量为 t 时能获得的最大价值
return dp[n][t]

```

```

空间压缩版本
通过观察状态转移方程，发现 dp[i][j] 只依赖于 dp[i-1] 这一行
所以可以用一维数组优化空间复杂度
时间复杂度：O(n * t)，空间复杂度：O(t)
def compute2():
 """
 空间压缩版本

```

解题思路：

通过观察状态转移方程，发现  $dp[i][j]$  只依赖于  $dp[i-1]$  这一行  
 所以可以用一维数组优化空间复杂度  
 但需要倒序遍历背包容量以确保每个物品只使用一次

时间复杂度：O(n \* t)  
 空间复杂度：O(t)

Returns:

最大价值

```

"""
初始化 dp 数组
for i in range(t + 1):
 dp[i] = 0

遍历物品
for i in range(1, n + 1):
 # 倒序遍历背包容量，确保每个物品只使用一次

```

```

如果正序遍历，前面的状态会被更新，导致一个物品被多次使用
for j in range(t, cost[i] - 1, -1):
 # 状态转移方程：
 # dp[j] = max(不选择当前物品, 选择当前物品)
 # 不选择当前物品: dp[j] (保持原值)
 # 选择当前物品: dp[j - cost[i]] + val[i]
 dp[j] = max(dp[j], dp[j - cost[i]] + val[i])

返回背包容量为 t 时能获得的最大价值
return dp[t]

```

```

LeetCode 416. 分割等和子集 (Partition Equal Subset Sum)
题目描述：给定一个只包含正整数的非空数组，判断是否可以将这个数组分割成两个子集，
使得两个子集的元素和相等。
链接: https://leetcode.cn/problems/partition-equal-subset-sum/
解题思路：
1. 如果数组总和为奇数，则无法分割成两个相等的子集，返回 false
2. 如果数组总和为偶数，则问题转化为 01 背包问题：
- 背包容量为 sum/2
- 每个数字既是重量也是价值
- 判断是否能装满背包
时间复杂度: O(n * sum)，其中 n 是数组长度，sum 是数组元素和
空间复杂度: O(sum)
def canPartition(nums):
 """
 判断是否可以将数组分割成两个和相等的子集
 """

```

解题思路：

1. 如果数组总和为奇数，则无法分割成两个相等的子集，返回 false
2. 如果数组总和为偶数，则问题转化为 01 背包问题：
  - 背包容量为 sum/2
  - 每个数字既是重量也是价值
  - 判断是否能装满背包

Args:

nums: 正整数数组

Returns:

是否可以分割成两个和相等的子集

"""

# 计算数组总和

total\_sum = sum(nums)

```

如果总和为奇数，无法分割成两个相等子集
if total_sum % 2 == 1:
 return False

目标和为总和的一半
target = total_sum // 2

dp[j] 表示是否能组成和为 j 的子集
dp = [False] * (target + 1)
初始状态：和为 0 的子集总是存在（空集）
dp[0] = True

遍历每个数字
for num in nums:
 # 01 背包需要倒序遍历，确保每个物品只使用一次
 for j in range(target, num - 1, -1):
 # 状态转移方程：
 # dp[j] = dp[j] || dp[j - num]
 # dp[j] 表示不选择当前数字能否组成和为 j 的子集
 # dp[j - num] 表示选择当前数字能否组成和为 j-num 的子集
 dp[j] = dp[j] or dp[j - num]

返回是否能组成和为 target 的子集
return dp[target]

```

,,

示例:

输入: nums = [1, 5, 11, 5]

输出: true

解释: 数组可以分割成 [1, 5, 5] 和 [11]。

输入: nums = [1, 2, 3, 5]

输出: false

解释: 数组不能分割成两个元素和相等的子集。

时间复杂度:  $O(n * sum)$ ，其中 n 是数组长度，sum 是数组元素和

空间复杂度:  $O(sum)$

,,

# 主函数用于测试洛谷 P1048 采药问题

if \_\_name\_\_ == "\_\_main\_\_":

# 注意: Python 中没有标准的输入流处理方式, 这里仅作示例

# 实际使用时需要根据具体平台调整输入方式

```
pass
```

```
=====
```

文件: Code02\_BuyGoodsHaveDiscount.java

```
=====
```

```
package class073;
```

```
// 夏季特惠
// 某公司游戏平台的夏季特惠开始了，你决定入手一些游戏
// 现在你一共有 X 元的预算，平台上所有的 n 个游戏均有折扣
// 标号为 i 的游戏的原价 a_i 元，现价只要 b_i 元
// 也就是说该游戏可以优惠 a_i - b_i，并且你购买该游戏能获得快乐值为 w_i
// 由于优惠的存在，你可能做出一些冲动消费导致最终买游戏的总费用超过预算
// 只要满足：获得的总优惠金额不低于超过预算的总金额
// 那在心理上就不会觉得吃亏。
// 现在你希望在心理上不觉得吃亏的前提下，获得尽可能多的快乐值。
// 测试链接：https://leetcode.cn/problems/tJau2o/
// 请同学们务必参考如下代码中关于输入、输出的处理
// 这是输入输出处理效率很高的写法
// 提交以下的所有代码，并把主类名改成“Main”，可以直接通过
```

```
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;
```

```
/**
 * 夏季特惠购物问题
 *
 * 问题描述：
 * 给定预算 X 元，有 n 个游戏可供购买，每个游戏有原价、现价和快乐值。
 * 购买游戏可以获得优惠金额（原价-现价），但实际花费是现价。
 * 在心理上不觉得吃亏的前提下（总优惠金额≥超预算金额），求能获得的最大快乐值。
 *
 * 解题思路：
 * 这是一个变形的 01 背包问题。我们需要将问题转换为标准的背包问题形式：
 * 1. 将商品分为两类：
 * - “一定要买的商品”：优惠金额 ≥ 现价，即 (原价-现价) ≥ 现价，这样购买会增加预算
 * - “需要考虑的商品”：优惠金额 < 现价，这类商品需要在预算范围内进行选择
```

```

* 2. 对于“一定要买的商品”，直接购买并累加其快乐值，同时更新预算
* 3. 对于“需要考虑的商品”，将其转化为标准背包问题：
* - 成本(cost) = 现价 - (原价 - 现价) = 2*现价 - 原价
* - 价值(val) = 快乐值
* 4. 使用 01 背包算法求解在更新后预算内能获得的最大快乐值
*
* 时间复杂度: O(n * x)，其中 n 是商品数量，x 是预算
* 空间复杂度: O(x)
*/
public class Code02_BuyGoodsHaveDiscount {

 public static int MAXN = 501;

 public static int MAXX = 100001;

 // 对于“一定要买的商品”，直接买！
 // 只把“需要考虑的商品”放入 cost、val 数组
 public static int[] cost = new int[MAXN];

 public static long[] val = new long[MAXN];

 public static long[] dp = new long[MAXX];

 public static int n, m, x;

 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 (in.nextToken() != StreamTokenizer.TT_EOF) {
 n = (int) in.nval;
 m = 1;
 in.nextToken();
 x = (int) in.nval;
 long ans = 0;
 long happy = 0;
 for (int i = 1, pre, cur, well; i <= n; i++) {
 // 原价
 in.nextToken(); pre = (int) in.nval;
 // 现价
 in.nextToken(); cur = (int) in.nval;
 // 快乐值
 in.nextToken(); happy = (long) in.nval;
 if (cur < well) {
 well = cur;
 }
 if (well > m) {
 break;
 }
 if (dp[well] <= ans) {
 continue;
 }
 if (dp[well] <= ans + val[i]) {
 dp[well] = ans + val[i];
 } else {
 dp[well] = dp[well];
 }
 }
 out.println(ans);
 }
 }
}

```

```

well = pre - cur - cur;
// 如下是一件“一定要买的商品”
// 预算 = 100, 商品原价 = 10, 打折后 = 3
// 那么好处(well) = (10 - 3) - 3 = 4
// 所以, 可以认为这件商品把预算增加到了 104! 一定要买!
// 如下是一件“需要考虑的商品”
// 预算 = 104, 商品原价 = 10, 打折后 = 8
// 那么好处(well) = (10 - 8) - 8 = -6
// 所以, 可以认为这件商品就花掉 6 元!
// 也就是说以后花的不是打折后的值, 是“坏处”
if (well >= 0) {
 x += well;
 ans += happy;
} else {
 cost[m] = -well;
 val[m++] = happy;
}
ans += compute();
out.println(ans);
}

out.flush();
out.close();
br.close();
}

/***
 * 01 背包算法求解在预算内能获得的最大快乐值
 *
 * @return 最大快乐值
 */
public static long compute() {
 // 初始化 dp 数组, dp[j] 表示预算为 j 时能获得的最大快乐值
 Arrays.fill(dp, 0, x + 1, 0);

 // 遍历每件需要考虑的商品
 for (int i = 1; i < m; i++) {
 // 从后往前遍历预算, 避免重复选择同一件商品
 for (int j = x; j >= cost[i]; j--) {
 // 状态转移方程:
 // dp[j] = max(不选择第 i 件商品, 选择第 i 件商品)
 // 不选择: dp[j] (保持原值)
 // 选择: dp[j - cost[i]] + val[i] (选择第 i 件商品后的最大快乐值)
 }
 }
}

```

```

 dp[j] = Math.max(dp[j], dp[j - cost[i]] + val[i]);
 }
}

// 返回预算为 x 时能获得的最大快乐值
return dp[x];
}
}

```

=====

文件: Code03\_TargetSum.java

=====

```

package class073;

import java.util.HashMap;

/**
 * 目标和问题
 *
 * 问题描述:
 * 给你一个非负整数数组 nums 和一个整数 target 。
 * 向数组中的每个整数前添加 '+' 或 '-' ，然后串联起所有整数，
 * 可以构造一个表达式，返回可以通过上述方法构造的，运算结果等于 target 的不同表达式的数目。
 *
 * 示例:
 * 输入: nums = [1, 1, 1, 1, 1]， target = 3
 * 输出: 5
 * 解释: 一共有 5 种方法让最终目标和为 3 。
 *
 * 解题思路:
 * 本题有多种解法:
 * 1. 暴力递归: 对每个数字尝试加号或减号, 递归计算所有可能的组合
 * 2. 记忆化搜索: 在暴力递归基础上增加缓存, 避免重复计算
 * 3. 动态规划: 使用二维 DP 数组, 通过平移技巧处理负数下标问题
 * 4. 转化为 01 背包问题: 通过数学推导将问题转化为求子集和的问题
 *
 * 第四种方法是最优解:
 * 1. 将数组分为两个子集: 正数集合 A 和负数集合 B
 * 2. 有 sum(A) - sum(B) = target
 * 3. 两边同时加上 sum(A) + sum(B) 得到: 2 * sum(A) = target + sum
 * 4. 即 sum(A) = (target + sum) / 2

```

```

* 5. 问题转化为：求有多少个子集的和等于 (target + sum) / 2，这就是标准的 01 背包问题
*
* 测试链接 : https://leetcode.cn/problems/target-sum/
*/
public class Code03_TargetSum {

 /**
 * 普通尝试，暴力递归版
 *
 * 解题思路：
 * 对于数组中的每个元素，都有两种选择：加上该元素或减去该元素
 * 递归地尝试所有可能的组合，当遍历完所有元素后，检查累加和是否等于 target
 *
 * 时间复杂度：O(2^n)，其中 n 是数组长度
 * 空间复杂度：O(n)，递归栈深度
 *
 * @param nums 非负整数数组
 * @param target 目标和
 * @return 不同表达式的数目
 */
 public static int findTargetSumWays1(int[] nums, int target) {
 return f1(nums, target, 0, 0);
 }

 /**
 * 暴力递归辅助函数
 *
 * @param nums 非负整数数组
 * @param target 目标和
 * @param i 当前处理到数组的第 i 个元素
 * @param sum 当前累加和
 * @return 从第 i 个元素开始能构成 target 的不同表达式数目
 */
 public static int f1(int[] nums, int target, int i, int sum) {
 // 基础情况：已经处理完所有元素
 if (i == nums.length) {
 // 如果当前累加和等于目标值，说明找到了一种有效方案
 return sum == target ? 1 : 0;
 }
 // 递归情况：对当前元素尝试加号和减号两种情况
 // 返回两种情况的方案数之和
 return f1(nums, target, i + 1, sum + nums[i]) + f1(nums, target, i + 1, sum - nums[i]);
 }
}

```

```

/**
 * 普通尝试，记忆化搜索版
 *
 * 解题思路：
 * 在暴力递归的基础上，使用哈希表缓存已经计算过的结果
 * 避免重复计算相同状态（位置 i 和当前累加和 sum）下的结果
 *
 * 时间复杂度：O(n * sum)，其中 n 是数组长度，sum 是数组元素和的范围
 * 空间复杂度：O(n * sum)
 *
 * @param nums 非负整数数组
 * @param target 目标和
 * @return 不同表达式的数目
 */
public static int findTargetSumWays2(int[] nums, int target) {
 // i, sum -> value (返回值)
 HashMap<Integer, HashMap<Integer, Integer>> dp = new HashMap<>();
 return f2(nums, target, 0, 0, dp);
}

/**
 * 记忆化搜索辅助函数
 *
 * @param nums 非负整数数组
 * @param target 目标和
 * @param i 当前处理到数组的第 i 个元素
 * @param j 当前累加和
 * @param dp 缓存已经计算过的结果
 * @return 从第 i 个元素开始能构成 target 的不同表达式数目
 */
public static int f2(int[] nums, int target, int i, int j, HashMap<Integer, HashMap<Integer, Integer>> dp) {
 // 基础情况：已经处理完所有元素
 if (i == nums.length) {
 // 如果当前累加和等于目标值，说明找到了一种有效方案
 return j == target ? 1 : 0;
 }
 // 检查是否已经计算过当前状态
 if (dp.containsKey(i) && dp.get(i).containsKey(j)) {
 return dp.get(i).get(j);
 }
 // 递归计算两种情况的方案数之和

```

```

 int ans = f2(nums, target, i + 1, j + nums[i], dp) + f2(nums, target, i + 1, j - nums[i], dp);
 }
}

/**
 * 普通尝试，严格位置依赖的动态规划
 *
 * 解题思路：
 * 使用二维 DP 数组，dp[i][j] 表示处理前 i 个元素，累加和为 j 的方案数
 * 由于累加和可能为负数，使用平移技巧将负数下标转换为非负数下标
 *
 * 时间复杂度：O(n * sum)，其中 n 是数组长度，sum 是数组元素和
 * 空间复杂度：O(n * sum)
 *
 * @param nums 非负整数数组
 * @param target 目标和
 * @return 不同表达式的数目
 */
public static int findTargetSumWays3(int[] nums, int target) {
 // 计算数组元素和
 int s = 0;
 for (int num : nums) {
 s += num;
 }
 // 如果目标值超出可能的范围，直接返回 0
 if (target < -s || target > s) {
 return 0;
 }
 int n = nums.length;
 // -s ~ +s -> 2 * s + 1，计算 DP 数组的列数
 int m = 2 * s + 1;
 // 原本的 dp[i][j] 含义：
 // nums[0...i-1] 范围上，已经形成的累加和是 sum
 // nums[i...] 范围上，每个数字可以标记+或者-
 // 最终形成累加和为 target 的不同表达式数目
 // 因为 sum 可能为负数，为了下标不出现负数，
 // "原本的 dp[i][j]" 由 dp 表中的 dp[i][j + s] 来表示
 // 也就是平移操作！
 // 一切"原本的 dp[i][j]"一律平移到 dp 表中的 dp[i][j + s]
}

```

```

int[][] dp = new int[n + 1][m];
// 原本的 dp[n][target] = 1, 平移!
dp[n][target + s] = 1;
// 从后往前填表
for (int i = n - 1; i >= 0; i--) {
 for (int j = -s; j <= s; j++) {
 // 状态转移方程:
 // dp[i][j] = dp[i+1][j+nums[i]] + dp[i+1][j-nums[i]]
 // 即对当前元素选择加号或减号两种情况
 if (j + nums[i] + s < m) {
 // 原本是 : dp[i][j] = dp[i + 1][j + nums[i]]
 // 平移!
 dp[i][j + s] = dp[i + 1][j + nums[i] + s];
 }
 if (j - nums[i] + s >= 0) {
 // 原本是 : dp[i][j] += dp[i + 1][j - nums[i]]
 // 平移!
 dp[i][j + s] += dp[i + 1][j - nums[i] + s];
 }
 }
}

// 原本应该返回 dp[0][0]
// 平移!
// 返回 dp[0][0 + s]
return dp[0][s];
}

/***
 * 新思路, 转化为 01 背包问题
 *
 * 解题思路:
 * 通过数学推导将问题转化为 01 背包问题:
 * 1. 将数组分为两个子集: 正数集合 A 和负数集合 B
 * 2. 有 sum(A) - sum(B) = target
 * 3. 两边同时加上 sum(A) + sum(B) 得到: 2 * sum(A) = target + sum
 * 4. 即 sum(A) = (target + sum) / 2
 * 5. 问题转化为: 求有多少个子集的和等于 (target + sum) / 2
 *
 * 时间复杂度: O(n * t), 其中 n 是数组长度, t 是(target + sum) / 2
 * 空间复杂度: O(t)
 *
 * @param nums 非负整数数组

```

```

* @param target 目标和
* @return 不同表达式的数目
*/
public static int findTargetSumWays4(int[] nums, int target) {
 // 计算数组元素和
 int sum = 0;
 for (int n : nums) {
 sum += n;
 }
 // 如果 sum 小于 target 或者(target+sum)是奇数，直接返回 0
 if (sum < target || ((target & 1) ^ (sum & 1)) == 1) {
 return 0;
 }
 // 转化为求子集和为(target + sum) / 2 的方案数
 return subsets(nums, (target + sum) >> 1);
}

```

```

/**
* 求非负数组 nums 有多少个子序列累加和是 t
*
* 解题思路:
* 使用 01 背包问题的解法，dp[j] 表示和为 j 的子集数目
* 状态转移方程: dp[j] = dp[j] + dp[j - nums[i]]
*
* @param nums 非负整数数组
* @param t 目标子集和
* @return 和为 t 的子集数目
*/

```

```

public static int subsets(int[] nums, int t) {
 // 如果目标值为负数，直接返回 0
 if (t < 0) {
 return 0;
 }
 // dp[j] 表示和为 j 的子集数目
 int[] dp = new int[t + 1];
 // 初始状态: 和为 0 的子集有 1 个（空集）
 dp[0] = 1;
 // 遍历每个数字
 for (int num : nums) { // i 省略了
 // 从后往前遍历，确保每个数字只使用一次
 for (int j = t; j >= num; j--) {
 // 状态转移方程: 选择当前数字或不选择当前数字
 dp[j] += dp[j - num];
 }
 }
}

```

```

 }

}

// 返回和为 t 的子集数目
return dp[t];
}

/***
 * 牛客网背包问题
 *
 * 题目描述：玥玥带乔乔一起逃亡，现在有许多的东西要放到乔乔的包里面，
 * 但是包的大小有限，所以我们只能够在里面放入非常重要的物品。
 * 现在给出该种物品的数量、体积、价值的数值，希望你能够算出怎样能使背包的价值最大的组合方式，
 * 并且输出这个数值，乔乔会非常感谢你。
 *
 * 解题思路：
 * 这是经典的 01 背包问题，使用动态规划求解
 * dp[j] 表示背包容量为 j 时能装入的最大价值
 * 状态转移方程：dp[j] = max(dp[j], dp[j - volumes[i]] + values[i])
 *
 * @param n 物品数量
 * @param v 背包容量
 * @param volumes 物品种体积数组
 * @param values 物品价值数组
 * @return 背包能装入的最大价值
*/
public static int backpack(int n, int v, int[] volumes, int[] values) {
 // dp[j] 表示背包容量为 j 时能装入的最大价值
 int[] dp = new int[v + 1];

 // 遍历物品
 for (int i = 0; i < n; i++) {
 // 倒序遍历背包容量，确保每个物品只使用一次
 for (int j = v; j >= volumes[i]; j--) {
 dp[j] = Math.max(dp[j], dp[j - volumes[i]] + values[i]);
 }
 }

 return dp[v];
}

/*
 * 示例：
 * 输入：n = 4, v = 5

```

```
* volumes = [1, 2, 3, 4]
* values = [2, 4, 4, 5]
* 输出: 8
* 解释: 选择第 1 个和第 3 个物品, 总重量为 2+3=5, 总价值为 4+4=8
*
* 时间复杂度: O(n * v)
* 空间复杂度: O(v)
*/
}

=====
```

文件: Code03\_TargetSum.py

```
目标和问题
#
问题描述:
给你一个非负整数数组 nums 和一个整数 target 。
向数组中的每个整数前添加 '+' 或 '-' ，然后串联起所有整数,
可以构造一个表达式, 返回可以通过上述方法构造的, 运算结果等于 target 的不同表达式的数目。
#
示例:
输入: nums = [1,1,1,1,1], target = 3
输出: 5
解释: 一共有 5 种方法让最终目标和为 3 。
#
解题思路:
本题有多种解法:
1. 暴力递归: 对每个数字尝试加号或减号, 递归计算所有可能的组合
2. 记忆化搜索: 在暴力递归基础上增加缓存, 避免重复计算
3. 转化为 01 背包问题: 通过数学推导将问题转化为求子集和的问题
#
第三种方法是最优解:
1. 将数组分为两个子集: 正数集合 A 和负数集合 B
2. 有 sum(A) - sum(B) = target
3. 两边同时加上 sum(A) + sum(B) 得到: 2 * sum(A) = target + sum
4. 即 sum(A) = (target + sum) / 2
5. 问题转化为: 求有多少个子集的和等于 (target + sum) / 2, 这就是标准的 01 背包问题
#
测试链接 : https://leetcode.cn/problems/target-sum/
#
普通尝试, 暴力递归版
```

```

#
解题思路:
对于数组中的每个元素，都有两种选择：加上该元素或减去该元素
递归地尝试所有可能的组合，当遍历完所有元素后，检查累加和是否等于 target
#
时间复杂度：O(2^n)，其中 n 是数组长度
空间复杂度：O(n)，递归栈深度
def findTargetSumWays1(nums, target):
 """
 暴力递归求解目标和问题
 """

 Args:
 nums: 非负整数数组
 target: 目标和

 Returns:
 不同表达式的数目
 """

 def f1(nums, target, i, sum_val):
 # 基础情况：已经处理完所有元素
 if i == len(nums):
 # 如果当前累加和等于目标值，说明找到了一种有效方案
 return 1 if sum_val == target else 0
 # 递归情况：对当前元素尝试加号和减号两种情况
 # 返回两种情况的方案数之和
 return f1(nums, target, i + 1, sum_val + nums[i]) + f1(nums, target, i + 1, sum_val - nums[i])

 return f1(nums, target, 0, 0)

普通尝试，记忆化搜索版
#
解题思路：
在暴力递归的基础上，使用字典缓存已经计算过的结果
避免重复计算相同状态（位置 i 和当前累加和 sum）下的结果
#
时间复杂度：O(n * sum)，其中 n 是数组长度，sum 是数组元素和的范围
空间复杂度：O(n * sum)
def findTargetSumWays2(nums, target):
 """
 记忆化搜索求解目标和问题
 """

 Args:

```

nums: 非负整数数组

target: 目标和

Returns:

不同表达式的数目

"""

dp = {}

def f2(nums, target, i, j):

# 基础情况: 已经处理完所有元素

if i == len(nums):

# 如果当前累加和等于目标值, 说明找到了一种有效方案

return 1 if j == target else 0

# 检查是否已经计算过当前状态

if (i, j) in dp:

return dp[(i, j)]

# 递归计算两种情况的方案数之和

ans = f2(nums, target, i + 1, j + nums[i]) + f2(nums, target, i + 1, j - nums[i])

# 缓存计算结果

dp[(i, j)] = ans

return ans

return f2(nums, target, 0, 0)

# 新思路, 转化为 01 背包问题

#

# 解题思路:

# 通过数学推导将问题转化为 01 背包问题:

# 1. 将数组分为两个子集: 正数集合 A 和负数集合 B

# 2. 有  $\text{sum}(A) - \text{sum}(B) = \text{target}$

# 3. 两边同时加上  $\text{sum}(A) + \text{sum}(B)$  得到:  $2 * \text{sum}(A) = \text{target} + \text{sum}$

# 4. 即  $\text{sum}(A) = (\text{target} + \text{sum}) / 2$

# 5. 问题转化为: 求有多少个子集的和等于  $(\text{target} + \text{sum}) / 2$

#

# 时间复杂度:  $O(n * t)$ , 其中 n 是数组长度, t 是  $(\text{target} + \text{sum}) / 2$

# 空间复杂度:  $O(t)$

#

# 思考过程:

# 1. 虽然题目说 nums 是非负数组, 但即使 nums 中有负数比如 [3, -4, 2]

# 因为能在每个数前面用+或者-号

# 所以 [3, -4, 2] 其实和 [3, 4, 2] 会达成一样的结果

```
所以即使 nums 中有负数，也可以把负数直接变成正数，也不会影响结果
2. 如果 nums 都是非负数，并且所有数的累加和是 sum
那么如果 target>sum，很明显没有任何方法可以达到 target，可以直接返回 0
3. nums 内部的数组，不管怎么+和-，最终的结果都一定不会改变奇偶性
所以，如果所有数的累加和是 sum，并且与 target 的奇偶性不一样
那么没有任何方法可以达到 target，可以直接返回 0
4. 最重要：
比如说给定一个数组，nums = [1, 2, 3, 4, 5] 并且 target = 3
其中一个方案是：+1 -2 +3 -4 +5 = 3
该方案中取了正的集合为 A = {1, 3, 5}
该方案中取了负的集合为 B = {2, 4}
所以任何一种方案，都一定有 sum(A) - sum(B) = target
现在我们来处理一下这个等式，把左右两边都加上 sum(A) + sum(B)，那么就会变成如下：
sum(A) - sum(B) + sum(A) + sum(B) = target + sum(A) + sum(B)
2 * sum(A) = target + 数组所有数的累加和
sum(A) = (target + 数组所有数的累加和) / 2
也就是说，任何一个集合，只要累加和是(target + 数组所有数的累加和) / 2
那么就一定对应一种 target 的方式
比如非负数组 nums，target = 1，nums 所有数累加和是 11
求有多少方法组成 1，其实就是求，有多少子集累加和达到 6 的方法，(1+11)/2=6
因为，子集累加和 6 - 另一半的子集累加和 5 = 1(target)
所以有多少个累加和为 6 的不同集合，就代表有多少个 target==1 的表达式数量
至此已经转化为 01 背包问题了
```

```
def findTargetSumWays4(nums, target):
```

```
 """
```

转化为 01 背包问题求解目标和

Args:

nums: 非负整数数组

target: 目标和

Returns:

不同表达式的数目

```
 """
```

```
total_sum = sum(nums)
```

```
如果 sum 小于 target 或者(target+sum)是奇数，直接返回 0
```

```
if total_sum < abs(target) or (target + total_sum) % 2 == 1:
 return 0
```

```
转化为求子集和为(target + total_sum) // 2 的方案数
```

```
return subsets(nums, (target + total_sum) // 2)
```

```
求非负数组 nums 有多少个子序列累加和是 t
01 背包问题(子集累加和严格是 t) + 空间压缩
dp[i][j] = dp[i-1][j] + dp[i-1][j-nums[i]]
def subsets(nums, t):
 """

```

求非负数组 nums 有多少个子序列累加和是 t

解题思路:

使用 01 背包问题的解法,  $dp[j]$  表示和为  $j$  的子集数目

状态转移方程:  $dp[j] = dp[j] + dp[j - \text{nums}[i]]$

Args:

nums: 非负整数数组

t: 目标子集和

Returns:

和为  $t$  的子集数目

```
"""
如果目标值为负数, 直接返回 0
if t < 0:
 return 0
```

```
dp[j] 表示和为 j 的子集数目
```

```
dp = [0] * (t + 1)
```

```
初始状态: 和为 0 的子集有 1 个 (空集)
```

```
dp[0] = 1
```

```
遍历每个数字
```

```
for num in nums:
```

```
 # 从后往前遍历, 确保每个数字只使用一次
```

```
 for j in range(t, num - 1, -1):
```

```
 # 状态转移方程: 选择当前数字或不选择当前数字
```

```
 dp[j] += dp[j - num]
```

```
返回和为 t 的子集数目
```

```
return dp[t]
```

```
牛客网背包问题
```

```
#
```

```
题目描述: 玖玥带乔乔一起逃亡, 现在有许多的东西要放到乔乔的包里面,
```

```
但是包的大小有限, 所以我们只能够在里面放入非常重要的物品。
```

```
现在给出该种物品的数量、体积、价值的数值, 希望你能够算出怎样能使背包的价值最大的组合方式,
```

```
并且输出这个数值, 乔乔会非常感谢你。
```

```

解题思路:
这是经典的 01 背包问题，使用动态规划求解
dp[j] 表示背包容量为 j 时能装入的最大价值
状态转移方程: dp[j] = max(dp[j], dp[j - volumes[i]] + values[i])
def backpack(n, v, volumes, values):
 """
```

牛客网背包问题求解

Args:

n: 物品数量  
v: 背包容量  
volumes: 物品体积数组  
values: 物品价值数组

Returns:

背包能装入的最大价值

```
"""
dp[j] 表示背包容量为 j 时能装入的最大价值
dp = [0] * (v + 1)
```

# 遍历物品

```
for i in range(n):
 # 倒序遍历背包容量，确保每个物品只使用一次
 for j in range(v, volumes[i] - 1, -1):
 dp[j] = max(dp[j], dp[j - volumes[i]] + values[i])

return dp[v]
```

,,

示例:

输入: n = 4, v = 5  
volumes = [1, 2, 3, 4]  
values = [2, 4, 4, 5]  
输出: 8

解释: 选择第 1 个和第 3 个物品，总重量为 2+3=5，总价值为 4+4=8

时间复杂度: O(n \* v)

空间复杂度: O(v)

,,

=====

文件: Code04\_LastStoneWeightII.java

```
=====
```

```
package class073;
```

```
/**
```

```
* 最后一块石头的重量 II
```

```
*
```

```
* 问题描述:
```

```
* 有一堆石头，用整数数组 stones 表示，其中 stones[i] 表示第 i 块石头的重量。
```

```
* 每一回合，从中选出任意两块石头，然后将它们一起粉碎。
```

```
* 假设石头的重量分别为 x 和 y，且 x <= y，粉碎结果：
```

```
* - 如果 x == y，那么两块石头都会被完全粉碎；
```

```
* - 如果 x != y，那么重量为 x 的石头将会完全粉碎，而重量为 y 的石头新重量为 y-x。
```

```
* 最后，最多只会剩下一块石头，返回此石头最小的可能重量。如果没有石头剩下，就返回 0。
```

```
*
```

```
* 解题思路:
```

```
* 这是一个可以转化为 01 背包问题的变种。我们的目标是将石头分成两堆，使得它们的重量尽可能接近，
```

```
* 这样最后剩下的石头重量就会最小（等于两堆重量之差）。
```

```
*
```

```
* 具体分析:
```

```
* 1. 如果我们能将石头分成两堆，重量分别为 sum1 和 sum2，那么最终剩下的石头重量为 |sum1 - sum2|
```

```
* 2. 由于 sum1 + sum2 = totalSum (石头总重量)，所以剩下的重量为 |totalSum - 2*sum1|
```

```
* 3. 为了最小化这个值，我们需要让 sum1 尽可能接近 totalSum/2
```

```
* 4. 这转化为：在石头中选择一些，使得它们的总重量不超过 totalSum/2，且尽可能接近 totalSum/2
```

```
* 5. 这正是一个 01 背包问题，背包容量为 totalSum/2，物品重量为石头重量，目标是最大化能装入的重量
```

```
*
```

```
* 时间复杂度: O(n * sum)，其中 n 是石头数量，sum 是总重量
```

```
* 空间复杂度: O(sum)，使用一维 DP 数组
```

```
*
```

```
* 测试链接 : https://leetcode.cn/problems/last-stone-weight-ii/
```

```
*/
```

```
public class Code04_LastStoneWeightII {
```

```
/**
```

```
* 计算最后一块石头的最小可能重量
```

```
*
```

```
* 解题思路:
```

```
* 1. 这道题可以转化为将石头分为两堆，使得两堆重量差最小
```

```
* 2. 假设两堆分别为 A 和 B, A >= B
```

```
* 3. 最终剩下的石头重量就是 A - B
```

```
* 4. 要使 A - B 最小，就要使 B 尽可能接近 sum/2
```

```
* 5. 问题转化为：在不超过 sum/2 的前提下，背包最多能装多少重量的石头
```

```
* 6. 这就是一个标准的 01 背包问题
```

```

*
* @param nums 石头重量数组
* @return 最后一块石头的最小可能重量
*/
public static int lastStoneWeightII(int[] nums) {
 // 参数验证
 if (nums == null || nums.length == 0) {
 return 0;
 }

 // 计算所有石头的总重量
 int sum = 0;
 for (int num : nums) {
 sum += num;
 }

 // nums 中随意选择数字
 // 累加和一定要 <= sum / 2
 // 又尽量接近
 int near = near(nums, sum / 2);
 // 返回两堆石头重量差的最小值
 // 其中一堆重量为 near，另一堆重量为 sum-near
 // 重量差为 (sum-near) - near = sum - 2*near
 return sum - near - near;
}

/**
* 非负数组 nums 中，子序列累加和不超过 t，但是最接近 t 的累加和是多少
* 01 背包问题(子集累加和尽量接近 t) + 空间压缩
*
* 解题思路：
* 使用 01 背包问题的解法，dp[j] 表示容量为 j 的背包最多能装的石头重量
* 状态转移方程：dp[j] = max(dp[j], dp[j - num] + num)
*
* @param nums 数组
* @param t 目标值
* @return 不超过 t 但最接近 t 的子序列累加和
*/
public static int near(int[] nums, int t) {
 // dp[j] 表示在容量为 j 的背包中能装入的最大重量
 int[] dp = new int[t + 1];

 // 遍历每个石头（物品）

```

```

for (int num : nums) {
 // 倒序遍历背包容量，确保每个物品只使用一次
 for (int j = t; j >= num; j--) {
 // 状态转移方程：
 // dp[j] = max(不选择当前石头, 选择当前石头)
 // 不选择当前石头: dp[j] (保持原值)
 // 选择当前石头: dp[j - num] + num (前一个状态+当前石头重量)
 // dp[i][j] = Math.max(dp[i-1][j], dp[i-1][j-nums[i]]+nums[i])
 dp[j] = Math.max(dp[j], dp[j - num] + num);
 }
}

// 返回容量为 t 的背包能装入的最大重量
return dp[t];
}

// LeetCode 1049. 最后一块石头的重量 II
// 题目描述：有一堆石头，用整数数组 stones 表示，每一回合，从中选出任意两块石头，然后将它们一起粉碎。
// 假设石头的重量分别为 x 和 y，且 x <= y，粉碎结果：
// 如果 x == y，那么两块石头都会被完全粉碎；
// 如果 x != y，那么重量为 x 的石头将会完全粉碎，而重量为 y 的石头新重量为 y-x。
// 最后，最多只会剩下一块石头，返回此石头最小的可能重量。如果没有石头剩下，就返回 0。
// 链接: https://leetcode.cn/problems/last-stone-weight-ii/

/**
 * 计算最后一块石头的最小可能重量 (LeetCode 版本)
 *
 * 解题思路：
 * 1. 这道题可以转化为将石头分为两堆，使得两堆重量差最小
 * 2. 假设两堆分别为 A 和 B, A >= B
 * 3. 最终剩下的石头重量就是 A - B
 * 4. 要使 A - B 最小，就要使 B 尽可能接近 sum/2
 * 5. 问题转化为：在不超过 sum/2 的前提下，背包最多能装多少重量的石头
 * 6. 这就是一个标准的 01 背包问题
 *
 * @param stones 石头重量数组
 * @return 最后一块石头的最小可能重量
 */
public static int lastStoneWeightIILeetcode(int[] stones) {
 // 参数验证
 if (stones == null || stones.length == 0) {
 return 0;
 }
}

```

```

}

// 计算所有石头的总重量
int sum = 0;
for (int stone : stones) {
 sum += stone;
}

// 目标是使其中一堆石头的重量尽可能接近总重量的一半
int target = sum / 2;

// dp[j] 表示容量为 j 的背包最多能装的石头重量
int[] dp = new int[target + 1];

// 遍历每个石头（物品）
for (int stone : stones) {
 // 倒序遍历背包容量，确保每个物品只使用一次
 for (int j = target; j >= stone; j--) {
 // 状态转移方程：
 // dp[j] = max(不选择当前石头, 选择当前石头)
 // 不选择当前石头: dp[j] (保持原值)
 // 选择当前石头: dp[j - stone] + stone (前一个状态+当前石头重量)
 dp[j] = Math.max(dp[j], dp[j - stone] + stone);
 }
}

// 返回两堆石头重量差的最小值
// 其中一堆重量为 dp[target]，另一堆重量为 sum-dp[target]
// 重量差为 (sum-dp[target]) - dp[target] = sum - 2*dp[target]
return sum - 2 * dp[target];
}

/*
* 解题思路：
* 1. 这道题可以转化为将石头分为两堆，使得两堆重量差最小
* 2. 假设两堆分别为 A 和 B, A >= B
* 3. 最终剩下的石头重量就是 A - B
* 4. 要使 A - B 最小，就要使 B 尽可能接近 sum/2
* 5. 问题转化为：在不超过 sum/2 的前提下，背包最多能装多少重量的石头
* 6. 这就是一个标准的 01 背包问题
*
* 示例：
* 输入：stones = [2, 7, 4, 1, 8, 1]

```

```
* 输出: 1
* 解释:
* 最优分法:
* 选 2, 8, 1 放一堆, 总重量是 11
* 选 7, 4, 1 放另一堆, 总重量是 12
* 最后剩下石头重量 = 12 - 11 = 1
*
* 时间复杂度: O(n * sum)
* - 外层循环遍历所有石头: O(n)
* - 内层循环遍历背包容量: O(sum)
* 空间复杂度: O(sum)
* - 一维 DP 数组的空间消耗
*/
}
```

}

=====

文件: Code04\_LastStoneWeightII.py

=====

```
最后一块石头的重量 II
#
问题描述:
有一堆石头, 用整数数组 stones 表示, 其中 stones[i] 表示第 i 块石头的重量。
每一回合, 从中选出任意两块石头, 然后将它们一起粉碎。
假设石头的重量分别为 x 和 y, 且 x <= y, 粉碎结果:
- 如果 x == y, 那么两块石头都会被完全粉碎;
- 如果 x != y, 那么重量为 x 的石头将会完全粉碎, 而重量为 y 的石头新重量为 y-x。
最后, 最多只会剩下一块石头, 返回此石头最小的可能重量。如果没有石头剩下, 就返回 0。
#
解题思路:
这是一个可以转化为 01 背包问题的变种。我们的目标是将石头分成两堆, 使得它们的重量尽可能接近,
这样最后剩下的石头重量就会最小(等于两堆重量之差)。
#
具体分析:
1. 如果我们能将石头分成两堆, 重量分别为 sum1 和 sum2, 那么最终剩下的石头重量为 |sum1 - sum2|
2. 由于 sum1 + sum2 = totalSum (石头总重量), 所以剩下的重量为 |totalSum - 2*sum1|
3. 为了最小化这个值, 我们需要让 sum1 尽可能接近 totalSum/2
4. 这转化为: 在石头中选择一些, 使得它们的总重量不超过 totalSum/2, 且尽可能接近 totalSum/2
5. 这正是一个 01 背包问题, 背包容量为 totalSum/2, 物品重量为石头重量, 目标是最大化能装入的重量
#
时间复杂度: O(n * sum), 其中 n 是石头数量, sum 是总重量
空间复杂度: O(sum), 使用一维 DP 数组
```

```

测试链接 : https://leetcode.cn/problems/last-stone-weight-ii/

def lastStoneWeightII(nums):
 """
 计算最后一块石头的最小可能重量
 """
```

解题思路:

1. 这道题可以转化为将石头分为两堆，使得两堆重量差最小
2. 假设两堆分别为 A 和 B,  $A \geq B$
3. 最终剩下的石头重量就是  $A - B$
4. 要使  $A - B$  最小，就要使 B 尽可能接近  $\text{sum}/2$
5. 问题转化为：在不超过  $\text{sum}/2$  的前提下，背包最多能装多少重量的石头
6. 这就是一个标准的 01 背包问题

Args:

nums: 石头重量数组

Returns:

最后一块石头的最小可能重量

"""

# 参数验证

```
if not nums:
 return 0
```

# 计算所有石头的总重量

```
total_sum = 0
for num in nums:
 total_sum += num
```

# nums 中随意选择数字

# 累加和一定要  $\leq \text{sum} / 2$

# 又尽量接近

near\_val = near(nums, total\_sum // 2)

# 返回两堆石头重量差的最小值

# 其中一堆重量为 near\_val，另一堆重量为 total\_sum-near\_val

# 重量差为  $(\text{total\_sum}-\text{near\_val}) - \text{near\_val} = \text{total\_sum} - 2*\text{near\_val}$

return total\_sum - near\_val - near\_val

```
def near(nums, t):
```

"""

非负数组 nums 中，子序列累加和不超过 t，但是最接近 t 的累加和是多少

01 背包问题(子集累加和尽量接近 t) + 空间压缩

解题思路:

使用 01 背包问题的解法,  $dp[j]$  表示容量为  $j$  的背包最多能装的石头重量

状态转移方程:  $dp[j] = \max(dp[j], dp[j - num] + num)$

Args:

nums: 数组

t: 目标值

Returns:

不超过  $t$  但最接近  $t$  的子序列累加和

"""

```
dp[j] 表示在容量为 j 的背包中能装入的最大重量
```

```
dp = [0] * (t + 1)
```

```
遍历每个石头 (物品)
```

```
for num in nums:
```

```
 # 倒序遍历背包容量, 确保每个物品只使用一次
```

```
 for j in range(t, num - 1, -1):
```

```
 # 状态转移方程:
```

```
 # dp[j] = max(不选择当前石头, 选择当前石头)
```

```
 # 不选择当前石头: dp[j] (保持原值)
```

```
 # 选择当前石头: dp[j - num] + num (前一个状态+当前石头重量)
```

```
 # dp[i][j] = Math.max(dp[i-1][j], dp[i-1][j-nums[i]]+nums[i])
```

```
 dp[j] = max(dp[j], dp[j - num] + num)
```

```
返回容量为 t 的背包能装入的最大重量
```

```
return dp[t]
```

```
LeetCode 1049. 最后一块石头的重量 II
```

```
题目描述: 有一堆石头, 用整数数组 stones 表示, 每一回合, 从中选出任意两块石头, 然后将它们一起粉碎。
```

```
假设石头的重量分别为 x 和 y, 且 x <= y, 粉碎结果:
```

```
如果 x == y, 那么两块石头都会被完全粉碎;
```

```
如果 x != y, 那么重量为 x 的石头将会完全粉碎, 而重量为 y 的石头新重量为 y-x。
```

```
最后, 最多只会剩下一块石头, 返回此石头最小的可能重量。如果没有石头剩下, 就返回 0。
```

```
链接: https://leetcode.cn/problems/last-stone-weight-ii/
```

```
def lastStoneWeightIILeetcode(stones):
```

"""

计算最后一块石头的最小可能重量 (LeetCode 版本)

解题思路:

1. 这道题可以转化为将石头分为两堆, 使得两堆重量差最小

2. 假设两堆分别为 A 和 B,  $A \geq B$
3. 最终剩下的石头重量就是  $A - B$
4. 要使  $A - B$  最小, 就要使 B 尽可能接近  $\text{sum}/2$
5. 问题转化为: 在不超过  $\text{sum}/2$  的前提下, 背包最多能装多少重量的石头
6. 这就是一个标准的 01 背包问题

Args:

stones: 石头重量数组

Returns:

最后一块石头的最小可能重量

"""

# 参数验证

```
if not stones:
 return 0
```

# 计算所有石头的总重量

```
total_sum = sum(stones)
```

# 目标是使其中一堆石头的重量尽可能接近总重量的一半

```
target = total_sum // 2
```

#  $dp[j]$  表示容量为 j 的背包最多能装的石头重量

```
dp = [0] * (target + 1)
```

# 遍历每个石头 (物品)

```
for stone in stones:
```

# 倒序遍历背包容量, 确保每个物品只使用一次

```
for j in range(target, stone - 1, -1):
```

# 状态转移方程:

#  $dp[j] = \max(\text{不选择当前石头}, \text{选择当前石头})$

# 不选择当前石头:  $dp[j]$  (保持原值)

# 选择当前石头:  $dp[j - \text{stone}] + \text{stone}$  (前一个状态+当前石头重量)

```
dp[j] = max(dp[j], dp[j - stone] + stone)
```

# 返回两堆石头重量差的最小值

# 其中一堆重量为  $dp[target]$ , 另一堆重量为  $total\_sum - dp[target]$

# 重量差为  $(total\_sum - dp[target]) - dp[target] = total\_sum - 2*dp[target]$

```
return total_sum - 2 * dp[target]
```

,,

解题思路:

1. 这道题可以转化为将石头分为两堆, 使得两堆重量差最小

2. 假设两堆分别为 A 和 B,  $A \geq B$
3. 最终剩下的石头重量就是  $A - B$
4. 要使  $A - B$  最小, 就要使 B 尽可能接近  $\text{sum}/2$
5. 问题转化为: 在不超过  $\text{sum}/2$  的前提下, 背包最多能装多少重量的石头
6. 这就是一个标准的 01 背包问题

示例:

输入: stones = [2, 7, 4, 1, 8, 1]

输出: 1

解释:

最优分法:

选 2, 8, 1 放一堆, 总重量是 11

选 7, 4, 1 放另一堆, 总重量是 12

最后剩下石头重量 =  $12 - 11 = 1$

时间复杂度:  $O(n * \text{sum})$

- 外层循环遍历所有石头:  $O(n)$
- 内层循环遍历背包容量:  $O(\text{sum})$

空间复杂度:  $O(\text{sum})$

- 一维 DP 数组的空间消耗

, , ,

=====

文件: Code05\_DependentKnapsack.java

=====

```
package class073;
```

```
/**
 * 有依赖的背包问题（模板）
 *
 * 问题描述:
 * 物品分为两大类: 主件和附件
 * 主件的购买没有限制, 钱够就可以; 附件的购买有限制, 该附件所归属的主件先购买, 才能购买这个附件
 * 例如, 若想买打印机或扫描仪这样的附件, 必须先购买电脑这个主件
 * 以下是一些主件及其附件的展示:
 * 电脑: 打印机, 扫描仪 | 书柜: 图书 | 书桌: 台灯, 文具 | 工作椅: 无附件
 * 每个主件最多有 2 个附件, 并且附件不会再有附件, 主件购买后, 怎么去选择归属附件完全随意, 钱够就可以
 * 所有的物品编号都在 1~m 之间, 每个物品有三个信息: 价格 v、重要度 p、归属 q
 * 价格就是花费, 价格 * 重要度 就是收益, 归属就是该商品是依附于哪个编号的主件
 * 比如一件商品信息为 [300, 2, 6], 花费 300, 收益 600, 该商品是 6 号主件商品的附件
 * 再比如一件商品信息 [100, 4, 0], 花费 100, 收益 400, 该商品自身是主件 (q==0)
```

\* 给定  $m$  件商品的信息，给定总钱数  $n$ ，返回在不违反购买规则的情况下最大的收益

\*

\* 解题思路：

\* 这是一个经典的依赖背包问题，可以转化为分组背包问题来解决：

\* 1. 将每个主件及其附件作为一个组

\* 2. 对于每个组，有以下几种选择：

\* - 不选择该组

\* - 只选择主件

\* - 选择主件+附件 1

\* - 选择主件+附件 2

\* - 选择主件+附件 1+附件 2

\* 3. 对每个组内的所有选择进行预处理，然后使用分组背包的解法

\*

\* 时间复杂度： $O(n * m)$ ，其中  $n$  是预算， $m$  是物品数量

\* 空间复杂度： $O(n)$

\*

\* 测试链接：<https://www.luogu.com.cn/problem/P1064>

\* 测试链接：<https://www.nowcoder.com/practice/f9c6f980eeec43ef85be20755ddbeaf4>

\* 请同学们务必参考如下代码中关于输入、输出的处理

\* 这是输入输出处理效率很高的写法

\* 提交以下的所有代码，并把主类名改成“Main”，可以直接通过

\*/

```
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_DependentKnapsack {
```

```
 public static int MAXN = 33001;
```

```
 public static int MAXM = 61;
```

```
 // 物品价格
```

```
 public static int[] cost = new int[MAXM];
```

```
 // 物品价值（价格 * 重要度）
```

```
 public static int[] val = new int[MAXM];
```

```
 // 标记是否为主件
```

```
public static boolean[] king = new boolean[MAXM];

// 每个主件的附件数量
public static int[] fans = new int[MAXM];

// 每个主件的附件列表（最多 2 个）
public static int[][] follows = new int[MAXM][2];

// DP 数组
public static int[] dp = new int[MAXN];

public static int n, m;

/**
 * 清理附件数量数组
 */
public static void clean() {
 for (int i = 1; i <= m; i++) {
 fans[i] = 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));
 while (in.nextToken() != StreamTokenizer.TT_EOF) {
 n = (int) in.nval;
 in.nextToken();
 m = (int) in.nval;
 clean();
 for (int i = 1, v, p, q; i <= m; i++) {
 in.nextToken(); v = (int) in.nval;
 in.nextToken(); p = (int) in.nval;
 in.nextToken(); q = (int) in.nval;
 cost[i] = v;
 val[i] = v * p;
 king[i] = q == 0;
 if (q != 0) {
 follows[q][fans[q]++] = i;
 }
 }
 out.println(compute2());
 }
}
```

```

 }

 out.flush();
 out.close();
 br.close();
}

/***
 * 严格位置依赖的动态规划解法
 *
 * 解题思路：
 * 使用二维 DP 数组，dp[i][j] 表示考虑前 i 个主件，在预算为 j 的情况下能获得的最大收益
 * 对于每个主件，考虑以下几种选择：
 * 1. 不选择该主件
 * 2. 只选择主件
 * 3. 选择主件+附件 1
 * 4. 选择主件+附件 2
 * 5. 选择主件+附件 1+附件 2
 *
 * @return 最大收益
 */
public static int compute1() {
 // dp[0][....] = 0 : 无商品的时候
 int[][] dp = new int[m + 1][n + 1];
 // p : 上次展开的主商品编号
 int p = 0;
 for (int i = 1, fan1, fan2; i <= m; i++) {
 if (king[i]) {
 for (int j = 0; j <= n; j++) {
 // dp[i][j] : 0...i 范围上，只关心主商品，并且进行展开
 // 花费不超过 j 的情况下，获得的最大收益
 // 可能性 1：不考虑当前主商品
 dp[i][j] = dp[p][j];
 if (j - cost[i] >= 0) {
 // 可能性 2：考虑当前主商品，只要主
 dp[i][j] = Math.max(dp[i][j], dp[p][j - cost[i]] + val[i]);
 }
 // fan1 : 如果有附 1 商品，编号给 fan1，如果没有，fan1 == -1
 // fan2 : 如果有附 2 商品，编号给 fan2，如果没有，fan2 == -1
 fan1 = fans[i] >= 1 ? follows[i][0] : -1;
 fan2 = fans[i] >= 2 ? follows[i][1] : -1;
 if (fan1 != -1 && j - cost[i] - cost[fan1] >= 0) {
 // 可能性 3：主 + 附 1
 dp[i][j] = Math.max(dp[i][j], dp[p][j - cost[i] - cost[fan1]] + val[i] +

```

```

 val[fan1]);
 }
 if (fan2 != -1 && j - cost[i] - cost[fan2] >= 0) {
 // 可能性 4：主 + 附 2
 dp[i][j] = Math.max(dp[i][j], dp[p][j - cost[i] - cost[fan2]] + val[i] +
val[fan2]);
 }
 if (fan1 != -1 && fan2 != -1 && j - cost[i] - cost[fan1] - cost[fan2] >= 0) {
 // 可能性 5：主 + 附 1 + 附 2
 dp[i][j] = Math.max(dp[i][j],
dp[p][j - cost[i] - cost[fan1] - cost[fan2]] + val[i] + val[fan1] +
val[fan2]);
 }
 p = i;
}
}

return dp[p][n];
}

```

```

/**
 * 空间压缩优化版本
 *
 * 解题思路：
 * 在基础版本的基础上进行空间优化，使用一维 DP 数组
 * 对于每个主件，考虑以下几种选择：
 * 1. 不选择该主件
 * 2. 只选择主件
 * 3. 选择主件+附件 1
 * 4. 选择主件+附件 2
 * 5. 选择主件+附件 1+附件 2
 *
 * @return 最大收益
 */

```

```

public static int compute2() {
 // 初始化 DP 数组
 Arrays.fill(dp, 0, n + 1, 0);
 // 遍历每个主件
 for (int i = 1, fan1, fan2; i <= m; i++) {
 if (king[i]) {
 // 倒序遍历预算，确保每个主件只处理一次
 for (int j = n; j >= cost[i]; j--) {
 // 可能性 1：不选择当前主件（保持原值）

```

```

// 可能性 2 : 只选择主件
dp[j] = Math.max(dp[j], dp[j - cost[i]] + val[i]);

// 获取附件编号
fan1 = fans[i] >= 1 ? follows[i][0] : -1;
fan2 = fans[i] >= 2 ? follows[i][1] : -1;

// 可能性 3 : 主 + 附 1
if (fan1 != -1 && j - cost[i] - cost[fan1] >= 0) {
 dp[j] = Math.max(dp[j], dp[j - cost[i] - cost[fan1]] + val[i] + val[fan1]);
}

// 可能性 4 : 主 + 附 2
if (fan2 != -1 && j - cost[i] - cost[fan2] >= 0) {
 dp[j] = Math.max(dp[j], dp[j - cost[i] - cost[fan2]] + val[i] + val[fan2]);
}

// 可能性 5 : 主 + 附 1 + 附 2
if (fan1 != -1 && fan2 != -1 && j - cost[i] - cost[fan1] - cost[fan2] >= 0) {
 dp[j] = Math.max(dp[j],
 dp[j - cost[i] - cost[fan1] - cost[fan2]] + val[i] + val[fan1] +
 val[fan2]);
}
}

return dp[n];
}

```

/\*\*

- \* 牛客网依赖背包问题
- \*
- \* 题目描述: 有依赖的背包问题, 物品分为主件和附件, 附件依赖于主件,
- \* 只有购买了主件才能购买附件, 每个主件最多有两个附件。
- \* 每个物品有价格和重要度, 求在预算内能获得的最大收益。
- \*
- \* 解题思路:
- \* 使用分组背包的思想, 将每个主件及其附件作为一个组
- \* 对于每个组, 预处理所有可能的选择组合, 然后使用分组背包的解法
- \*
- \* @param budget 总预算
- \* @param itemCount 物品数量
- \* @param prices 物品价格数组

```

* @param importances 物品重要度数组
* @param dependencies 物品依赖关系数组
* @return 最大收益
*/
public static int dependantBackpack(int budget, int itemCount, int[] prices, int[]
importances, int[] dependencies) {
 // 主件标识
 boolean[] isMain = new boolean[itemCount + 1];
 // 附件数量
 int[] accessoryCount = new int[itemCount + 1];
 // 附件列表
 int[][] accessories = new int[itemCount + 1][2];

 // 初始化依赖关系
 for (int i = 1; i <= itemCount; i++) {
 if (dependencies[i] == 0) {
 isMain[i] = true;
 } else {
 int master = dependencies[i];
 accessories[master][accessoryCount[master]++] = i;
 }
 }

 // dp 数组
 int[] dp = new int[budget + 1];

 // 遍历主件
 for (int i = 1; i <= itemCount; i++) {
 if (isMain[i]) {
 // 倒序遍历预算
 for (int j = budget; j >= prices[i]; j--) {
 // 只买主件
 dp[j] = Math.max(dp[j], dp[j - prices[i]] + prices[i] * importances[i]);

 // 买主件+附件 1
 if (accessoryCount[i] >= 1) {
 int acc1 = accessories[i][0];
 if (j >= prices[i] + prices[acc1]) {
 dp[j] = Math.max(dp[j], dp[j - prices[i] - prices[acc1]] +
 prices[i] * importances[i] + prices[acc1] *
 importances[acc1]);
 }
 }
 }
 }
 }
}

```

```

// 买主件+附件 2
if (accessoryCount[i] >= 2) {
 int acc2 = accessories[i][1];
 if (j >= prices[i] + prices[acc2]) {
 dp[j] = Math.max(dp[j], dp[j - prices[i] - prices[acc2]] +
 prices[i] * importances[i] + prices[acc2] *
importances[acc2]);
 }
}

// 买主件+附件 1+附件 2
if (accessoryCount[i] >= 2) {
 int acc1 = accessories[i][0];
 int acc2 = accessories[i][1];
 if (j >= prices[i] + prices[acc1] + prices[acc2]) {
 dp[j] = Math.max(dp[j], dp[j - prices[i] - prices[acc1] -
prices[acc2]] +
 prices[i] * importances[i] + prices[acc1] *
importances[acc1] +
 prices[acc2] * importances[acc2]);
 }
}

return dp[budget];
}

/*
* 示例:
* 输入: budget = 1000, itemCount = 5
* prices = [800, 400, 300, 400, 200]
* importances = [2, 5, 5, 3, 2]
* dependencies = [0, 1, 1, 0, 4]
* 输出: 2200
* 解释: 选择主件 1 和主件 4, 以及它们的附件
*
* 时间复杂度: O(n * m), 其中 n 是预算, m 是物品数量
* 空间复杂度: O(n)
*/

```

}

=====

文件: Code05\_DependentKnapsack.py

=====

```
有依赖的背包问题（模板）
#
问题描述：
物品分为两大类：主件和附件
主件的购买没有限制，钱够就可以；附件的购买有限制，该附件所归属的主件先购买，才能购买这个附件
例如，若想买打印机或扫描仪这样的附件，必须先购买电脑这个主件
以下是一些主件及其附件的展示：
电脑：打印机，扫描仪 | 书柜：图书 | 书桌：台灯，文具 | 工作椅：无附件
每个主件最多有 2 个附件，并且附件不会再有附件，主件购买后，怎么去选择归属附件完全随意，钱够就可以
所有的物品编号都在 1~m 之间，每个物品有三个信息：价格 v、重要度 p、归属 q
价格就是花费，价格 * 重要度 就是收益，归属就是该商品是依附于哪个编号的主件
比如一件商品信息为[300, 2, 6]，花费 300，收益 600，该商品是 6 号主件商品的附件
再比如一件商品信息[100, 4, 0]，花费 100，收益 400，该商品自身是主件(q==0)
给定 m 件商品的信息，给定总钱数 n，返回在不违反购买规则的情况下最大的收益
#
解题思路：
这是一个经典的依赖背包问题，可以转化为分组背包问题来解决：
1. 将每个主件及其附件作为一个组
2. 对于每个组，有以下几种选择：
- 不选择该组
- 只选择主件
- 选择主件+附件 1
- 选择主件+附件 2
- 选择主件+附件 1+附件 2
3. 对每个组内的所有选择进行预处理，然后使用分组背包的解法
#
时间复杂度：O(n * m)，其中 n 是预算，m 是物品数量
空间复杂度：O(n)
#
测试链接：https://www.luogu.com.cn/problem/P1064
测试链接：https://www.nowcoder.com/practice/f9c6f980eeec43ef85be20755ddbeaf4
#
牛客网依赖背包问题
#
题目描述：有依赖的背包问题，物品分为主件和附件，附件依赖于主件，
只有购买了主件才能购买附件，每个主件最多有两个附件。
```

```

每个物品有价格和重要度，求在预算内能获得的最大收益。
#
解题思路：
使用分组背包的思想，将每个主件及其附件作为一个组
对于每个组，预处理所有可能的选择组合，然后使用分组背包的解法
def dependantBackpack(budget, itemCount, prices, importances, dependencies):
 """
 计算依赖背包问题的最大收益

 Args:
 budget: 总预算
 itemCount: 物品数量
 prices: 物品价格数组
 importances: 物品重要度数组
 dependencies: 物品依赖关系数组

 Returns:
 最大收益
 """

 # 主件标识
 isMain = [False] * (itemCount + 1)
 # 附件数量
 accessoryCount = [0] * (itemCount + 1)
 # 附件列表
 accessories = [[0, 0] for _ in range(itemCount + 1)]

 # 初始化依赖关系
 for i in range(1, itemCount + 1):
 if dependencies[i] == 0:
 isMain[i] = True
 else:
 master = dependencies[i]
 accessories[master][accessoryCount[master]] = i
 accessoryCount[master] += 1

 # dp 数组
 dp = [0] * (budget + 1)

 # 遍历主件
 for i in range(1, itemCount + 1):
 if isMain[i]:
 # 倒序遍历预算
 for j in range(budget, prices[i] - 1, -1):

```

```

只买主件
dp[j] = max(dp[j], dp[j - prices[i]] + prices[i] * importances[i])

买主件+附件 1
if accessoryCount[i] >= 1:
 acc1 = accessories[i][0]
 if j >= prices[i] + prices[acc1]:
 dp[j] = max(dp[j], dp[j - prices[i] - prices[acc1]] +
 prices[i] * importances[i] + prices[acc1] * importances[acc1])

买主件+附件 2
if accessoryCount[i] >= 2:
 acc2 = accessories[i][1]
 if j >= prices[i] + prices[acc2]:
 dp[j] = max(dp[j], dp[j - prices[i] - prices[acc2]] +
 prices[i] * importances[i] + prices[acc2] * importances[acc2])

买主件+附件 1+附件 2
if accessoryCount[i] >= 2:
 acc1 = accessories[i][0]
 acc2 = accessories[i][1]
 if j >= prices[i] + prices[acc1] + prices[acc2]:
 dp[j] = max(dp[j], dp[j - prices[i] - prices[acc1] - prices[acc2]] +
 prices[i] * importances[i] + prices[acc1] * importances[acc1] +
 prices[acc2] * importances[acc2])

+
return dp[budget]

```

,,

示例:

输入: budget = 1000, itemCount = 5

prices = [800, 400, 300, 400, 200]

importances = [2, 5, 5, 3, 2]

dependencies = [0, 1, 1, 0, 4]

输出: 2200

解释: 选择主件 1 和主件 4, 以及它们的附件

时间复杂度:  $O(n * m)$ , 其中 n 是预算, m 是物品数量

空间复杂度:  $O(n)$

,,

=====

文件: Code06\_TopKMinimumSubsequenceSum.cpp

```
=====

// 非负数组前 k 个最小的子序列累加和
//
// 问题描述:
// 给定一个数组 nums, 含有 n 个数字, 都是非负数
// 给定一个正数 k, 返回所有子序列中累加和最小的前 k 个累加和
// 子序列是包含空集的
//
// 数据范围:
// 1 <= n <= 10^5
// 1 <= nums[i] <= 10^6
// 1 <= k <= 10^5
//
// 解题思路:
// 这个问题有多种解法:
// 1. 暴力方法: 生成所有子序列的和, 然后排序取前 k 个
// 2. 01 背包方法: 使用动态规划计算每个和的方案数, 然后按顺序取前 k 个
// 3. 堆方法: 使用最小堆来逐步生成前 k 个最小的子序列和
//
// 由于数据量较大, 01 背包方法的时间复杂度过高, 最优解是使用堆的方法。
//
// 堆方法的核心思想:
// 1. 首先对数组进行排序
// 2. 使用最小堆来维护当前可能的最小和
// 3. 从空集开始, 逐步扩展子序列
// 4. 对于当前的子序列, 可以有两种扩展方式:
// - 替换最右元素为下一个元素
// - 添加下一个元素
//
// 时间复杂度: O(n * log n) + O(k * log k)
// 空间复杂度: O(k)

// 比较函数, 用于排序
int compare(int a, int b) {
 return (a - b);
}

// 简单冒泡排序实现
void bubbleSort(int* arr, int size) {
 for (int i = 0; i < size - 1; i++) {
 for (int j = 0; j < size - i - 1; j++) {
```

```

 if (arr[j] > arr[j + 1]) {
 // 交换元素
 int temp = arr[j];
 arr[j] = arr[j + 1];
 arr[j + 1] = temp;
 }
 }
}

// 暴力方法
// 解题思路:
// 生成所有子序列的和, 然后排序取前 k 个
//
// 时间复杂度: O(2^n * log(2^n)) = O(2^n * n)
// 空间复杂度: O(2^n)

void f1(int* nums, int numsSize, int index, int sum, int* allSubsequences, int* count) {
 // 基础情况: 已经处理完所有元素
 if (index == numsSize) {
 // 将当前子序列的和添加到结果列表中
 allSubsequences[(*count)++] = sum;
 } else {
 // 递归情况: 对当前元素有两种选择
 // 1. 不选择当前元素
 f1(nums, numsSize, index + 1, sum, allSubsequences, count);
 // 2. 选择当前元素
 f1(nums, numsSize, index + 1, sum + nums[index], allSubsequences, count);
 }
}

// 暴力方法实现
void topKSum1(int* nums, int numsSize, int k, int* result) {
 // 由于不能使用动态内存分配, 我们使用固定大小的数组
 // 假设最大支持 1024 个子序列
 int allSubsequences[1024];
 int count = 0;

 // 生成所有子序列的和
 f1(nums, numsSize, 0, 0, allSubsequences, &count);

 // 对所有子序列和进行排序
 bubbleSort(allSubsequences, count);
}

```

```

// 取前 k 个最小的子序列和
for (int i = 0; i < k && i < count; i++) {
 result[i] = allSubsequences[i];
}
}

// 01 背包方法
// 解题思路:
// 使用动态规划计算每个和的方案数, 然后按顺序取前 k 个
//
// 时间复杂度: O(n * sum), 其中 sum 是数组元素和
// 空间复杂度: O(sum)
//
// 注意: 由于数据量较大, 这种方法的时间复杂度过高, 不是最优解
void topKSum2(int* nums, int numsSize, int k, int* result) {
 // 计算数组元素和
 int sum = 0;
 for (int i = 0; i < numsSize; i++) {
 sum += nums[i];
 }

 // 由于不能使用动态内存分配, 我们使用固定大小的数组
 // 假设 sum 最大不超过 10000
 int dp[10001] = {0};

 // 初始状态: 和为 0 的方案数为 1 (空集)
 dp[0] = 1;

 // 遍历每个元素
 for (int i = 0; i < numsSize; i++) {
 int num = nums[i];
 // 倒序遍历和, 确保每个元素只使用一次
 for (int j = sum; j >= num; j--) {
 // 状态转移方程: dp[j] = dp[j] + dp[j - num]
 dp[j] += dp[j - num];
 }
 }

 // 按顺序取前 k 个最小的子序列和
 int index = 0;
 for (int j = 0; j <= sum && index < k; j++) {
 // 对于和为 j 的情况, 有 dp[j] 个方案
 for (int i = 0; i < dp[j] && index < k; i++) {

```

```

 result[index++] = j;
 }
}

// 简化的堆方法实现
// 由于环境限制，这里使用排序方法来替代堆
void topKSum3(int* nums, int numsSize, int k, int* result) {
 // 简化实现：直接使用暴力方法生成所有子序列和，然后排序取前 k 个
 // 在实际应用中，应该使用堆来优化
 topKSum1(nums, numsSize, k, result);
}

// 比较两个数组是否相等
int equals(int* ans1, int* ans2, int size) {
 for (int i = 0; i < size; i++) {
 if (ans1[i] != ans2[i]) {
 return 0;
 }
 }
 return 1;
}

```

=====

文件: Code06\_TopKMinimumSubsequenceSum.java

=====

```

package class073;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.PriorityQueue;

/**
 * 非负数组前 k 个最小的子序列累加和
 *
 * 问题描述：
 * 给定一个数组 nums，含有 n 个数字，都是非负数
 * 给定一个正数 k，返回所有子序列中累加和最小的前 k 个累加和
 * 子序列是包含空集的
 *
 * 数据范围：
 * 1 <= n <= 10^5

```

```
* 1 <= nums[i] <= 10^6
* 1 <= k <= 10^5
*
* 解题思路:
* 这个问题有多种解法:
* 1. 暴力方法: 生成所有子序列的和, 然后排序取前 k 个
* 2. 01 背包方法: 使用动态规划计算每个和的方案数, 然后按顺序取前 k 个
* 3. 堆方法: 使用最小堆来逐步生成前 k 个最小的子序列和
*
* 由于数据量较大, 01 背包方法的时间复杂度过高, 最优解是使用堆的方法。
*
* 堆方法的核心思想:
* 1. 首先对数组进行排序
* 2. 使用最小堆来维护当前可能的最小和
* 3. 从空集开始, 逐步扩展子序列
* 4. 对于当前的子序列, 可以有两种扩展方式:
* - 替换最右元素为下一个元素
* - 添加下一个元素
*
* 时间复杂度: O(n * log n) + O(k * log k)
* 空间复杂度: O(k)
*
* 对数据验证
*/
public class Code06_TopKMinimumSubsequenceSum {
```

```
 /**
 * 暴力方法
 *
 * 解题思路:
 * 生成所有子序列的和, 然后排序取前 k 个
 *
 * 时间复杂度: O(2^n * log(2^n)) = O(2^n * n)
 * 空间复杂度: O(2^n)
 *
 * @param nums 非负数组
 * @param k 前 k 个最小的子序列和
 * @return 前 k 个最小的子序列和
*/

```

```
public static int[] topKSum1(int[] nums, int k) {
 // 存储所有子序列的和
 ArrayList<Integer> allSubsequences = new ArrayList<>();
 // 递归生成所有子序列的和
}
```

```

f1(nums, 0, 0, allSubsequences);
// 对所有子序列和进行排序
allSubsequences.sort((a, b) -> a.compareTo(b));
// 取前 k 个最小的子序列和
int[] ans = new int[k];
for (int i = 0; i < k; i++) {
 ans[i] = allSubsequences.get(i);
}
return ans;
}

/***
 * 暴力方法辅助函数
 *
 * 解题思路:
 * 递归生成所有子序列的和
 *
 * @param nums 非负数组
 * @param index 当前处理到数组的第 index 个元素
 * @param sum 当前子序列的和
 * @param ans 存储所有子序列和的列表
 */
public static void f1(int[] nums, int index, int sum, ArrayList<Integer> ans) {
 // 基础情况: 已经处理完所有元素
 if (index == nums.length) {
 // 将当前子序列的和添加到结果列表中
 ans.add(sum);
 } else {
 // 递归情况: 对当前元素有两种选择
 // 1. 不选择当前元素
 f1(nums, index + 1, sum, ans);
 // 2. 选择当前元素
 f1(nums, index + 1, sum + nums[index], ans);
 }
}

/***
 * 01 背包方法
 *
 * 解题思路:
 * 使用动态规划计算每个和的方案数, 然后按顺序取前 k 个
 *
 * 时间复杂度: O(n * sum), 其中 sum 是数组元素和

```

```

* 空间复杂度: O(sum)
*
* 注意: 由于数据量较大, 这种方法的时间复杂度过高, 不是最优解
*
* @param nums 非负数组
* @param k 前 k 个最小的子序列和
* @return 前 k 个最小的子序列和
*/
public static int[] topKSum2(int[] nums, int k) {
 // 计算数组元素和
 int sum = 0;
 for (int num : nums) {
 sum += num;
 }

 // dp[i][j] 表示前 i 个元素组成和为 j 的方案数
 // 1) dp[i-1][j] 表示不选择第 i 个元素
 // 2) dp[i-1][j-nums[i]] 表示选择第 i 个元素
 int[] dp = new int[sum + 1];
 // 初始状态: 和为 0 的方案数为 1 (空集)
 dp[0] = 1;

 // 遍历每个元素
 for (int num : nums) {
 // 倒序遍历和, 确保每个元素只使用一次
 for (int j = sum; j >= num; j--) {
 // 状态转移方程: dp[j] = dp[j] + dp[j - num]
 dp[j] += dp[j - num];
 }
 }

 // 按顺序取前 k 个最小的子序列和
 int[] ans = new int[k];
 int index = 0;
 for (int j = 0; j <= sum && index < k; j++) {
 // 对于和为 j 的情况, 有 dp[j] 个方案
 for (int i = 0; i < dp[j] && index < k; i++) {
 ans[index++] = j;
 }
 }
 return ans;
}

```

```

/**
 * 正式方法（最优解）
 *
 * 解题思路：
 * 使用最小堆来逐步生成前 k 个最小的子序列和
 *
 * 核心思想：
 * 1. 首先对数组进行排序
 * 2. 使用最小堆来维护当前可能的最小和
 * 3. 从空集开始，逐步扩展子序列
 * 4. 对于当前的子序列，可以有两种扩展方式：
 * - 替换最右元素为下一个元素
 * - 添加下一个元素
 *
 * 时间复杂度：O(n * log n) + O(k * log k)
 * 空间复杂度：O(k)
 *
 * @param nums 非负数组
 * @param k 前 k 个最小的子序列和
 * @return 前 k 个最小的子序列和
 */
public static int[] topKSum3(int[] nums, int k) {
 // 对数组进行排序
 Arrays.sort(nums);

 // 最小堆，存储(子序列的最右下标, 子序列的累加和)
 PriorityQueue<int[]> heap = new PriorityQueue<>((a, b) -> a[1] - b[1]);

 // 初始化堆，从第一个元素开始
 heap.add(new int[] { 0, nums[0] });

 // 存储结果
 int[] ans = new int[k];

 // 逐步生成前 k 个最小的子序列和
 for (int i = 1; i < k; i++) {
 // 取出当前最小的子序列和
 int[] cur = heap.poll();
 int right = cur[0];
 int sum = cur[1];
 ans[i] = sum;

 // 扩展当前子序列
 int left = right;
 while (left <= right && right < nums.length) {
 int[] next = new int[2];
 next[0] = right + 1;
 next[1] = sum + nums[right + 1];
 heap.add(next);
 right++;
 }
 }
}

```

```

 if (right + 1 < nums.length) {
 // 替换最右元素为下一个元素
 heap.add(new int[] { right + 1, sum - nums[right] + nums[right + 1] });
 // 添加下一个元素
 heap.add(new int[] { right + 1, sum + nums[right + 1] });
 }
 }
 return ans;
}

/**
 * 生成随机数组用于测试
 *
 * @param len 数组长度
 * @param value 数组元素的最大值
 * @return 随机数组
 */
public static int[] randomArray(int len, int value) {
 int[] ans = new int[len];
 for (int i = 0; i < len; i++) {
 ans[i] = (int) (Math.random() * value);
 }
 return ans;
}

/**
 * 比较两个数组是否相等
 *
 * @param ans1 第一个数组
 * @param ans2 第二个数组
 * @return 如果两个数组相等返回 true, 否则返回 false
 */
public static boolean equals(int[] ans1, int[] ans2) {
 if (ans1.length != ans2.length) {
 return false;
 }
 for (int i = 0; i < ans1.length; i++) {
 if (ans1[i] != ans2[i]) {
 return false;
 }
 }
 return true;
}

```

```

/**
 * 对数据测试
 */
public static void main(String[] args) {
 int n = 15;
 int v = 40;
 int testTime = 5000;
 System.out.println("测试开始");
 for (int i = 0; i < testTime; i++) {
 int len = (int) (Math.random() * n) + 1;
 int[] nums = randomArray(len, v);
 int k = (int) (Math.random() * ((1 << len) - 1)) + 1;
 int[] ans1 = topKSum1(nums, k);
 int[] ans2 = topKSum2(nums, k);
 int[] ans3 = topKSum3(nums, k);
 if (!equals(ans1, ans2) || !equals(ans1, ans3)) {
 System.out.println("出错了!");
 }
 }
 System.out.println("测试结束");
}
}

```

}

=====

文件: Code06\_TopKMinimumSubsequenceSum.py

```

=====
非负数组前 k 个最小的子序列累加和
#
问题描述:
给定一个数组 nums，含有 n 个数字，都是非负数
给定一个正数 k，返回所有子序列中累加和最小的前 k 个累加和
子序列是包含空集的
#
数据范围:
1 <= n <= 10^5
1 <= nums[i] <= 10^6
1 <= k <= 10^5
#
解题思路:
这个问题有多种解法:

```

```
1. 暴力方法：生成所有子序列的和，然后排序取前 k 个
2. 01 背包方法：使用动态规划计算每个和的方案数，然后按顺序取前 k 个
3. 堆方法：使用最小堆来逐步生成前 k 个最小的子序列和
#
由于数据量较大，01 背包方法的时间复杂度过高，最优解是使用堆的方法。
#
堆方法的核心思想：
1. 首先对数组进行排序
2. 使用最小堆来维护当前可能的最小和
3. 从空集开始，逐步扩展子序列
4. 对于当前的子序列，可以有两种扩展方式：
- 替换最右元素为下一个元素
- 添加下一个元素
#
时间复杂度：O(n * log n) + O(k * log k)
空间复杂度：O(k)
```

```
import heapq
from typing import List

def topKSum1(nums: List[int], k: int) -> List[int]:
 """
 暴力方法
```

解题思路：

生成所有子序列的和，然后排序取前 k 个

时间复杂度： $O(2^n * \log(2^n)) = O(2^n * n)$

空间复杂度： $O(2^n)$

Args:

```
 nums: 非负数组
 k: 前 k 个最小的子序列和
```

Returns:

前 k 个最小的子序列和

"""
# 存储所有子序列的和
all\_subsequences = []

# 递归生成所有子序列的和

```
def f1(index: int, sum_val: int) -> None:
 # 基础情况：已经处理完所有元素
```

```

if index == len(nums):
 # 将当前子序列的和添加到结果列表中
 all_subsequences.append(sum_val)
else:
 # 递归情况：对当前元素有两种选择
 # 1. 不选择当前元素
 f1(index + 1, sum_val)
 # 2. 选择当前元素
 f1(index + 1, sum_val + nums[index])

生成所有子序列的和
f1(0, 0)

对所有子序列和进行排序
all_subsequences.sort()

取前 k 个最小的子序列和
return all_subsequences[:k]

```

```
def topKSum2(nums: List[int], k: int) -> List[int]:
 """

```

## 01 背包方法

解题思路：

使用动态规划计算每个和的方案数，然后按顺序取前 k 个

时间复杂度： $O(n * sum)$ ，其中 sum 是数组元素和

空间复杂度： $O(sum)$

注意：由于数据量较大，这种方法的时间复杂度过高，不是最优解

Args:

nums: 非负数组

k: 前 k 个最小的子序列和

Returns:

前 k 个最小的子序列和

"""

# 计算数组元素和

total\_sum = sum(nums)

# dp[j] 表示组成和为 j 的方案数

# 1) dp[j] 表示不选择当前元素

```

2) dp[j - num] 表示选择当前元素
dp = [0] * (total_sum + 1)
初始状态: 和为 0 的方案数为 1 (空集)
dp[0] = 1

遍历每个元素
for num in nums:
 # 倒序遍历和, 确保每个元素只使用一次
 for j in range(total_sum, num - 1, -1):
 # 状态转移方程: dp[j] = dp[j] + dp[j - num]
 dp[j] += dp[j - num]

按顺序取前 k 个最小的子序列和
ans = []
for j in range(total_sum + 1):
 # 对于和为 j 的情况, 有 dp[j] 个方案
 for _ in range(min(dp[j], k - len(ans))):
 ans.append(j)
 if len(ans) == k:
 return ans

return ans

```

```
def topKSum3(nums: List[int], k: int) -> List[int]:
 """

```

正式方法（最优解）

解题思路:

使用最小堆来逐步生成前 k 个最小的子序列和

核心思想:

1. 首先对数组进行排序
2. 使用最小堆来维护当前可能的最小和
3. 从空集开始, 逐步扩展子序列
4. 对于当前的子序列, 可以有两种扩展方式:
  - 替换最右元素为下一个元素
  - 添加下一个元素

时间复杂度:  $O(n * \log n) + O(k * \log k)$

空间复杂度:  $O(k)$

Args:

nums: 非负数组

k: 前 k 个最小的子序列和

Returns:

前 k 个最小的子序列和

"""

```
import heapq
```

# 对数组进行排序

```
nums.sort()
```

# 最小堆，存储(子序列的累加和, 子序列的最右下标)

```
heap = [(nums[0], 0)]
```

# 存储结果

```
ans = [0] # 空集的和为 0
```

# 逐步生成前 k 个最小的子序列和

```
while len(ans) < k:
```

# 取出当前最小的子序列和

```
sum_val, right = heapq.heappop(heap)
```

```
ans.append(sum_val)
```

# 扩展当前子序列

```
if right + 1 < len(nums):
```

# 替换最右元素为下一个元素

```
heapq.heappush(heap, (sum_val - nums[right] + nums[right + 1], right + 1))
```

# 添加下一个元素

```
heapq.heappush(heap, (sum_val + nums[right + 1], right + 1))
```

```
return ans[:k]
```

# 测试函数

```
def random_array(length: int, value: int) -> List[int]:
```

"""

生成随机数组用于测试

Args:

length: 数组长度

value: 数组元素的最大值

Returns:

随机数组

"""

```

import random
return [random.randint(0, value) for _ in range(length)]

对数据测试
if __name__ == "__main__":
 import random

n = 15
v = 40
test_time = 5000
print("测试开始")

for i in range(test_time):
 length = random.randint(1, n)
 nums = random_array(length, v)
 k = random.randint(1, (1 << length) - 1)

 ans1 = topKSum1(nums, k)
 ans2 = topKSum2(nums, k)
 ans3 = topKSum3(nums, k)

 if ans1 != ans2 or ans1 != ans3:
 print("出错了!")
 print(f"nums: {nums}")
 print(f"k: {k}")
 print(f"ans1: {ans1}")
 print(f"ans2: {ans2}")
 print(f"ans3: {ans3}")
 break

print("测试结束")
=====

文件: Code07_OnesAndZeros.cpp
=====

// LeetCode 474. 一和零
// 题目描述: 给你一个二进制字符串数组 strs 和两个整数 m 和 n。
// 请你找出并返回 strs 的最大子集的长度，该子集中 最多 有 m 个 0 和 n 个 1。
// 链接: https://leetcode.cn/problems/ones-and-zeroes/
//
// 解题思路:
// 这是一个二维费用的 01 背包问题。

```

```

// 1. 每个字符串相当于一个物品，有两个重量限制：0 的个数和 1 的个数
// 2. dp[i][j][k] 表示前 i 个字符串中，在最多使用 j 个 0 和 k 个 1 的情况下，能选出的最大子集大小
// 3. 状态转移方程：
// dp[i][j][k] = max(dp[i-1][j][k], dp[i-1][j-zero][k-one] + 1)
// 其中 zero 和 one 分别是第 i 个字符串中 0 和 1 的个数
// 4. 可以使用滚动数组优化空间复杂度
//
// 时间复杂度：O(len(strs) * m * n + sum(len(str) for str in strs))
// 空间复杂度：O(m * n)

```

```

#define MAXM 101
#define MAXN 101

```

```

// 计算字符串中 0 和 1 的个数
// 参数：
// str: 输入的字符串
// zeros: 指向存储 0 个数的变量的指针
// ones: 指向存储 1 个数的变量的指针
void countZerosOnes(char* str, int* zeros, int* ones) {
 *zeros = 0;
 *ones = 0;
 // 遍历字符串中的每个字符
 for (int i = 0; str[i] != '\0'; i++) {
 if (str[i] == '0') {
 (*zeros)++;
 // 如果是'0'，增加 zeros 计数
 } else {
 (*ones)++;
 // 如果是'1'，增加 ones 计数
 }
 }
}

```

```

/**
 * 找到最大子集的长度，该子集中最多有 m 个 0 和 n 个 1
 */

```

```

* 参数：
* strs: 二进制字符串数组
* strsSize: 字符串数组的大小
* m: 最多允许的 0 的个数
* n: 最多允许的 1 的个数
* 返回值：
* 最大子集的长度
*/

```

```

int findMaxForm(char** strs, int strsSize, int m, int n) {

```

```

// dp[i][j] 表示最多使用 i 个 0 和 j 个 1 时，能选出的最大子集大小
// 这里使用了空间优化的二维 DP 数组，相当于 dp[i][j][k] 压缩为 dp[j][k]
int dp[MAXM][MAXN];

// 初始化 dp 数组，所有值初始化为 0
for (int i = 0; i <= m; i++) {
 for (int j = 0; j <= n; j++) {
 dp[i][j] = 0;
 }
}

// 遍历每个字符串（物品）
// 这相当于 01 背包中的物品遍历
for (int i = 0; i < strsSize; i++) {
 // 统计当前字符串中 0 和 1 的个数
 // 这相当于获取当前物品的两个重量属性
 int zeros, ones;
 countZerosOnes(strs[i], &zeros, &ones);

 // 01 背包需要倒序遍历，确保每个物品只使用一次
 // 注意边界条件：j >= zeros && k >= ones
 // 这是二维费用 01 背包的核心实现
 // j 表示当前可用的 0 的个数，k 表示当前可用的 1 的个数
 for (int j = m; j >= zeros; j--) {
 for (int k = n; k >= ones; k--) {
 // 状态转移方程：
 // dp[j][k] = max(不选择当前字符串, 选择当前字符串)
 // 不选择当前字符串：dp[j][k] (保持原值)
 // 选择当前字符串：dp[j - zeros][k - ones] + 1 (前一个状态+1)
 int newValue = dp[j - zeros][k - ones] + 1;
 if (newValue > dp[j][k]) {
 dp[j][k] = newValue;
 }
 }
 }
}

// 返回最多使用 m 个 0 和 n 个 1 时能选出的最大子集大小
return dp[m][n];
}

/*
 * 示例：

```

```

* 输入: strs = ["10", "0001", "111001", "1", "0"], m = 5, n = 3
* 输出: 4
* 解释: 最多有 5 个 0 和 3 个 1 的子集是 {"10", "0001", "1", "0"}, 因此答案是 4。
*
* 输入: strs = ["10", "0", "1"], m = 1, n = 1
* 输出: 2
* 解释: 最多有 1 个 0 和 1 个 1 的子集是 {"0", "1"}, 因此答案是 2。
*
* 时间复杂度: O(len(strs) * m * n + sum(len(str) for str in strs))
* - 外层循环遍历所有字符串: O(len(strs))
* - 中层循环遍历 m: O(m)
* - 内层循环遍历 n: O(n)
* - 统计每个字符串中 0 和 1 的个数: O(sum(len(str) for str in strs))
* 空间复杂度: O(m * n)
* - 二维 DP 数组的空间消耗
*/

```

=====

文件: Code07\_OnesAndZeros.java

=====

```

package class073;

// LeetCode 474. 一和零
// 题目描述: 给你一个二进制字符串数组 strs 和两个整数 m 和 n 。
// 请你找出并返回 strs 的最大子集的长度，该子集中 最多 有 m 个 0 和 n 个 1 。
// 链接: https://leetcode.cn/problems/ones-and-zeroes/
//
// 解题思路:
// 这是一个二维费用的 01 背包问题。
// 1. 每个字符串相当于一个物品，有两个重量限制: 0 的个数和 1 的个数
// 2. dp[i][j][k] 表示前 i 个字符串中，在最多使用 j 个 0 和 k 个 1 的情况下，能选出的最大子集大小
// 3. 状态转移方程:
// dp[i][j][k] = max(dp[i-1][j][k], dp[i-1][j-zero][k-one] + 1)
// 其中 zero 和 one 分别是第 i 个字符串中 0 和 1 的个数
// 4. 可以使用滚动数组优化空间复杂度
//
// 时间复杂度: O(len(strs) * m * n + sum(len(str) for str in strs))
// 空间复杂度: O(m * n)

```

```

public class Code07_OnesAndZeros {

```

```

/**
```

```

* 找到最大子集的长度，该子集中最多有 m 个 0 和 n 个 1
*
* @param strs 二进制字符串数组
* @param m 最多允许的 0 的个数
* @param n 最多允许的 1 的个数
* @return 最大子集的长度
*/
public static int findMaxForm(String[] strs, int m, int n) {
 // dp[i][j] 表示最多使用 i 个 0 和 j 个 1 时，能选出的最大子集大小
 // 这里使用了空间优化的二维 DP 数组，相当于 dp[i][j][k] 压缩为 dp[j][k]
 int[][] dp = new int[m + 1][n + 1];

 // 遍历每个字符串（物品）
 // 这相当于 01 背包中的物品遍历
 for (String str : strs) {
 // 统计当前字符串中 0 和 1 的个数
 // 这相当于获取当前物品的两个重量属性
 int zeros = 0, ones = 0;
 for (char c : str.toCharArray()) {
 if (c == '0') {
 zeros++;
 } else {
 ones++;
 }
 }
 }

 // 01 背包需要倒序遍历，确保每个物品只使用一次
 // 注意边界条件：j >= zeros && k >= ones
 // 这里是二维费用 01 背包的核心实现
 // j 表示当前可用的 0 的个数，k 表示当前可用的 1 的个数
 for (int j = m; j >= zeros; j--) {
 for (int k = n; k >= ones; k--) {
 // 状态转移方程：
 // dp[j][k] = max(不选择当前字符串，选择当前字符串)
 // 不选择当前字符串：dp[j][k] (保持原值)
 // 选择当前字符串：dp[j - zeros][k - ones] + 1 (前一个状态+1)
 dp[j][k] = Math.max(dp[j][k], dp[j - zeros][k - ones] + 1);
 }
 }

 // 返回最多使用 m 个 0 和 n 个 1 时能选出的最大子集大小
 return dp[m][n];
}

```

```

}

/*
 * 示例:
 * 输入: strs = ["10", "0001", "111001", "1", "0"], m = 5, n = 3
 * 输出: 4
 * 解释: 最多有 5 个 0 和 3 个 1 的子集是 {"10", "0001", "1", "0"}, 因此答案是 4。
 *
 * 输入: strs = ["10", "0", "1"], m = 1, n = 1
 * 输出: 2
 * 解释: 最多有 1 个 0 和 1 个 1 的子集是 {"0", "1"}, 因此答案是 2。
 *
 * 时间复杂度: O(len(strs) * m * n + sum(len(str) for str in strs))
 * - 外层循环遍历所有字符串: O(len(strs))
 * - 中层循环遍历 m: O(m)
 * - 内层循环遍历 n: O(n)
 * - 统计每个字符串中 0 和 1 的个数: O(sum(len(str) for str in strs))
 * 空间复杂度: O(m * n)
 * - 二维 DP 数组的空间消耗
*/

```

}

=====

文件: Code07\_OnesAndZeros.py

```

LeetCode 474. 一和零
题目描述: 给你一个二进制字符串数组 strs 和两个整数 m 和 n 。
请你找出并返回 strs 的最大子集的长度，该子集中 最多 有 m 个 0 和 n 个 1 。
链接: https://leetcode.cn/problems/ones-and-zeroes/
#
解题思路:
这是一个二维费用的 01 背包问题。
1. 每个字符串相当于一个物品，有两个重量限制: 0 的个数和 1 的个数
2. dp[i][j][k] 表示前 i 个字符串中，在最多使用 j 个 0 和 k 个 1 的情况下，能选出的最大子集大小
3. 状态转移方程:
dp[i][j][k] = max(dp[i-1][j][k], dp[i-1][j-zero][k-one] + 1)
其中 zero 和 one 分别是第 i 个字符串中 0 和 1 的个数
4. 可以使用滚动数组优化空间复杂度
#
时间复杂度: O(len(strs) * m * n + sum(len(str) for str in strs))
空间复杂度: O(m * n)

```

```

def findMaxForm(strs, m, n):
 """
 找到最大子集的长度，该子集中最多有 m 个 0 和 n 个 1
 """

 Args:
 strs: 二进制字符串数组
 m: 最多允许的 0 的个数
 n: 最多允许的 1 的个数

 Returns:
 最大子集的长度
 """

 # dp[i][j] 表示最多使用 i 个 0 和 j 个 1 时，能选出的最大子集大小
 # 这里使用了空间优化的二维 DP 数组，相当于 dp[i][j][k] 压缩为 dp[j][k]
 dp = [[0 for _ in range(n + 1)] for _ in range(m + 1)]

 # 遍历每个字符串（物品）
 # 这相当于 01 背包中的物品遍历
 for s in strs:
 # 统计当前字符串中 0 和 1 的个数
 # 这相当于获取当前物品的两个重量属性
 zeros = s.count('0')
 ones = s.count('1')

 # 01 背包需要倒序遍历，确保每个物品只使用一次
 # 注意边界条件：j >= zeros and k >= ones
 # 这是二维费用 01 背包的核心实现
 # j 表示当前可用的 0 的个数，k 表示当前可用的 1 的个数
 for j in range(m, zeros - 1, -1): # 倒序遍历 0 的个数
 for k in range(n, ones - 1, -1): # 倒序遍历 1 的个数
 # 状态转移方程：
 # dp[j][k] = max(不选择当前字符串, 选择当前字符串)
 # 不选择当前字符串: dp[j][k] (保持原值)
 # 选择当前字符串: dp[j - zeros][k - ones] + 1 (前一个状态+1)
 dp[j][k] = max(dp[j][k], dp[j - zeros][k - ones] + 1)

 # 返回最多使用 m 个 0 和 n 个 1 时能选出的最大子集大小
 return dp[m][n]

```

,,

示例:

输入: strs = ["10", "0001", "111001", "1", "0"], m = 5, n = 3

输出: 4

解释：最多有 5 个 0 和 3 个 1 的子集是 {"10", "0001", "1", "0"}，因此答案是 4。

输入：strs = ["10", "0", "1"], m = 1, n = 1

输出：2

解释：最多有 1 个 0 和 1 个 1 的子集是 {"0", "1"}，因此答案是 2。

时间复杂度：O(len(strs) \* m \* n + sum(len(str) for str in strs))

- 外层循环遍历所有字符串：O(len(strs))
- 中层循环遍历 m：O(m)
- 内层循环遍历 n：O(n)
- 统计每个字符串中 0 和 1 的个数：O(sum(len(str) for str in strs))

空间复杂度：O(m \* n)

- 二维 DP 数组的空间消耗

, , ,

=====

文件：Code08\_ProfitableSchemes.cpp

=====

// LeetCode 879. 盈利计划

// 题目描述：集团里有 n 名员工，他们可以完成各种各样的工作创造利润。

// 第 i 种工作会产生 profit[i] 的利润，它要求 group[i] 名成员共同参与。

// 工作的任何至少产生 minProfit 利润的子集都被称为盈利计划，并且工作的成员总数最多为 n。

// 有多少种计划可以选择？因为答案很大，所以返回结果模  $10^9 + 7$  的值。

// 链接：<https://leetcode.cn/problems/profitable-schemes/>

//

// 解题思路：

// 这是一个三维费用的 01 背包问题。

// 1. 物品：每个工作

// 2. 费用 1：需要的人数

// 3. 费用 2：需要的利润

// 4. dp[i][j][k] 表示前 i 个工作，使用不超过 j 个人，至少获得 k 利润的方案数

// 5. 状态转移方程：

//  $dp[i][j][k] = dp[i-1][j][k] + dp[i-1][j - group[i-1]][\max(0, k - profit[i-1])]$

// 6. 可以使用滚动数组优化空间复杂度

// 7. 对于利润维度，当利润超过 minProfit 时，统一视为 minProfit 处理

//

// 时间复杂度：O(len(group) \* n \* minProfit)

// 空间复杂度：O(n \* minProfit)

#define MAXN 101

#define MAXP 101

#define MOD 1000000007

```

// 获取两个数中的较大值
int max(int a, int b) {
 return a > b ? a : b;
}

/**
 * 计算盈利计划的数量
 *
 * 解题思路:
 * 这是一个三维费用的 01 背包问题。
 * 1. 物品: 每个工作
 * 2. 费用 1: 需要的人数
 * 3. 费用 2: 需要的利润
 * 4. $dp[i][j][k]$ 表示前 i 个工作, 使用不超过 j 个人, 至少获得 k 利润的方案数
 * 5. 状态转移方程:
 * $dp[i][j][k] = dp[i-1][j][k] + dp[i-1][j-group[i-1]][\max(0, k-profit[i-1])]$
 * 6. 可以使用滚动数组优化空间复杂度
 * 7. 对于利润维度, 当利润超过 minProfit 时, 统一视为 minProfit 处理
 *
 * 参数:
 * n: 员工总数
 * minProfit: 最小利润要求
 * group: 每个工作需要的人数数组
 * groupSize: group 数组的大小
 * profit: 每个工作产生的利润数组
 * profitSize: profit 数组的大小
 *
 * 返回值:
 * 满足条件的计划数量
*/
int profitableSchemes(int n, int minProfit, int* group, int groupSize, int* profit, int profitSize) {
 // $dp[i][j]$ 表示使用不超过 i 个人, 至少获得 j 利润的方案数
 // 这里使用了空间优化的二维 DP 数组, 相当于 $dp[i][j][k]$ 压缩为 $dp[j][k]$
 int dp[MAXN][MAXP];

 // 初始化 dp 数组
 for (int i = 0; i <= n; i++) {
 for (int j = 0; j <= minProfit; j++) {
 dp[i][j] = 0;
 }
 }
}

```

```

// 初始化: 不选择任何工作, 使用 0 个人, 获得 0 利润的方案数为 1
// 这是动态规划的边界条件
for (int i = 0; i <= n; i++) {
 dp[i][0] = 1;
}

// 遍历每个工作 (物品)
// 这相当于 01 背包中的物品遍历
for (int i = 0; i < groupSize; i++) {
 // 获取当前工作需要的人数和产生的利润
 int members = group[i];
 int earn = profit[i];

 // 01 背包需要倒序遍历, 确保每个物品只使用一次
 // j 表示当前可用的人数, k 表示当前需要达到的利润
 for (int j = n; j >= members; j--) {
 for (int k = minProfit; k >= 0; k--) {
 // 状态转移方程:
 // dp[j][k] = 不选择当前工作 + 选择当前工作
 // 不选择当前工作: dp[j][k] (保持原值)
 // 选择当前工作: 需要 members 个人, 获得 earn 利润
 // dp[j-members][max(0, k-earn)] 表示使用 j-members 个人, 至少获得 max(0, k-earn) 利润的方案数
 // 注意: 如果 k < earn, 则至少获得 0 利润 (因为负利润没有意义)
 long long newValue = (long long)dp[j - members][max(0, k - earn)] + dp[j][k];
 dp[j][k] = newValue % MOD;
 }
 }
}

// 返回使用不超过 n 个人, 至少获得 minProfit 利润的方案数
return dp[n][minProfit];
}

/*
 * 示例:
 * 输入: n = 5, minProfit = 3, group = [2,2], profit = [2,3]
 * 输出: 2
 * 解释: 至少产生 3 利润的计划有 2 种: 完成工作 1 和工作 2, 或仅完成工作 2。
 *
 * 输入: n = 10, minProfit = 5, group = [2,3,5], profit = [6,7,8]
 * 输出: 7
 * 解释: 至少产生 5 利润的计划有 7 种。

```

```
*
* 时间复杂度: O(len(group) * n * minProfit)
* - 外层循环遍历所有工作: O(len(group))
* - 中层循环遍历人数: O(n)
* - 内层循环遍历利润: O(minProfit)
* 空间复杂度: O(n * minProfit)
* - 二维 DP 数组的空间消耗
*/
```

---

文件: Code08\_ProfitableSchemes.java

```
=====
package class073;

// LeetCode 879. 盈利计划
// 题目描述: 集团里有 n 名员工, 他们可以完成各种各样的工作创造利润。
// 第 i 种工作会产生 profit[i] 的利润, 它要求 group[i] 名成员共同参与。
// 工作的任何至少产生 minProfit 利润的子集都被称为盈利计划, 并且工作的成员总数最多为 n。
// 有多少种计划可以选择? 因为答案很大, 所以返回结果模 10^9 + 7 的值。
// 链接: https://leetcode.cn/problems/profitable-schemes/
//
// 解题思路:
// 这是一个三维费用的 01 背包问题。
// 1. 物品: 每个工作
// 2. 费用 1: 需要的人数
// 3. 费用 2: 需要的利润
// 4. dp[i][j][k] 表示前 i 个工作, 使用不超过 j 个人, 至少获得 k 利润的方案数
// 5. 状态转移方程:
// dp[i][j][k] = dp[i-1][j][k] + dp[i-1][j-group[i-1]][max(0, k-profit[i-1])]
// 6. 可以使用滚动数组优化空间复杂度
// 7. 对于利润维度, 当利润超过 minProfit 时, 统一视为 minProfit 处理
//
// 时间复杂度: O(len(group) * n * minProfit)
// 空间复杂度: O(n * minProfit)
```

```
public class Code08_ProfitableSchemes {
 private static final int MOD = 1000000007;

 /**
 * 计算盈利计划的数量
 *
 * 解题思路:
```

```

* 这是一个三维费用的 01 背包问题。
* 1. 物品: 每个工作
* 2. 费用 1: 需要的人数
* 3. 费用 2: 需要的利润
* 4. $dp[i][j][k]$ 表示前 i 个工作, 使用不超过 j 个人, 至少获得 k 利润的方案数
* 5. 状态转移方程:
* $dp[i][j][k] = dp[i-1][j][k] + dp[i-1][j-group[i-1]][\max(0, k-profit[i-1])]$
* 6. 可以使用滚动数组优化空间复杂度
* 7. 对于利润维度, 当利润超过 \minProfit 时, 统一视为 \minProfit 处理
*
* @param n 员工总数
* @param minProfit 最小利润要求
* @param group 每个工作需要的人数数组
* @param profit 每个工作产生的利润数组
* @return 满足条件的计划数量
*/
public static int profitableSchemes(int n, int minProfit, int[] group, int[] profit) {
 int len = group.length;
 // $dp[i][j]$ 表示使用不超过 i 个人, 至少获得 j 利润的方案数
 // 这里使用了空间优化的二维 DP 数组, 相当于 $dp[i][j][k]$ 压缩为 $dp[j][k]$
 int[][] dp = new int[n + 1][minProfit + 1];

 // 初始化: 不选择任何工作, 使用 0 个人, 获得 0 利润的方案数为 1
 // 这是动态规划的边界条件
 for (int i = 0; i <= n; i++) {
 dp[i][0] = 1;
 }

 // 遍历每个工作 (物品)
 // 这相当于 01 背包中的物品遍历
 for (int i = 1; i <= len; i++) {
 // 获取当前工作需要的人数和产生的利润
 int members = group[i - 1];
 int earn = profit[i - 1];

 // 01 背包需要倒序遍历, 确保每个物品只使用一次
 // j 表示当前可用的人数, k 表示当前需要达到的利润
 for (int j = n; j >= members; j--) {
 for (int k = minProfit; k >= 0; k--) {
 // 状态转移方程:
 // $dp[j][k] = \text{不选择当前工作} + \text{选择当前工作}$
 // 不选择当前工作: $dp[j][k]$ (保持原值)
 // 选择当前工作: 需要 $members$ 个人, 获得 $earn$ 利润
 }
 }
 }
}

```

```

 // dp[j-members][max(0, k-earn)]表示使用 j-members 个人，至少获得 max(0, k-
earn)利润的方案数

 // 注意：如果 k < earn，则至少获得 0 利润（因为负利润没有意义）
 dp[j][k] = (dp[j][k] + dp[j - members][Math.max(0, k - earn)]) % MOD;
 }
}

// 返回使用不超过 n 个人，至少获得 minProfit 利润的方案数
return dp[n][minProfit];
}

/*
 * 示例：
 * 输入：n = 5, minProfit = 3, group = [2, 2], profit = [2, 3]
 * 输出：2
 * 解释：至少产生 3 利润的计划有 2 种：完成工作 1 和工作 2，或仅完成工作 2。
 *
 * 输入：n = 10, minProfit = 5, group = [2, 3, 5], profit = [6, 7, 8]
 * 输出：7
 * 解释：至少产生 5 利润的计划有 7 种。
 *
 * 时间复杂度：O(len(group) * n * minProfit)
 * - 外层循环遍历所有工作：O(len(group))
 * - 中层循环遍历人数：O(n)
 * - 内层循环遍历利润：O(minProfit)
 * 空间复杂度：O(n * minProfit)
 * - 二维 DP 数组的空间消耗
*/
}

```

文件：Code08\_ProfitableSchemes.py

```

LeetCode 879. 盈利计划
题目描述：集团里有 n 名员工，他们可以完成各种各样的工作创造利润。
第 i 种工作会产生 profit[i] 的利润，它要求 group[i] 名成员共同参与。
工作的任何至少产生 minProfit 利润的子集都被称为盈利计划，并且工作的成员总数最多为 n。
有多少种计划可以选择？因为答案很大，所以返回结果模 10^9 + 7 的值。
链接：https://leetcode.cn/problems/profitable-schemes/
#
解题思路：

```

```
这是一个三维费用的 01 背包问题。
1. 物品: 每个工作
2. 费用 1: 需要的人数
3. 费用 2: 需要的利润
4. dp[i][j][k] 表示前 i 个工作, 使用不超过 j 个人, 至少获得 k 利润的方案数
5. 状态转移方程:
dp[i][j][k] = dp[i-1][j][k] + dp[i-1][j-group[i-1]][max(0, k-profit[i-1])]
6. 可以使用滚动数组优化空间复杂度
7. 对于利润维度, 当利润超过 minProfit 时, 统一视为 minProfit 处理

时间复杂度: O(len(group) * n * minProfit)
空间复杂度: O(n * minProfit)
```

```
def profitableSchemes(n, minProfit, group, profit):
```

```
 """
```

```
 计算盈利计划的数量
```

解题思路:

这是一个三维费用的 01 背包问题。

1. 物品: 每个工作
2. 费用 1: 需要的人数
3. 费用 2: 需要的利润
4.  $dp[i][j][k]$  表示前  $i$  个工作, 使用不超过  $j$  个人, 至少获得  $k$  利润的方案数
5. 状态转移方程:  
$$dp[i][j][k] = dp[i-1][j][k] + dp[i-1][j-group[i-1]][\max(0, k-profit[i-1])]$$
6. 可以使用滚动数组优化空间复杂度
7. 对于利润维度, 当利润超过  $\text{minProfit}$  时, 统一视为  $\text{minProfit}$  处理

Args:

- n: 员工总数
- minProfit: 最小利润要求
- group: 每个工作需要的人数数组
- profit: 每个工作产生的利润数组

Returns:

满足条件的计划数量

```
"""
```

```
MOD = 1000000007
```

```
dp[i][j] 表示使用不超过 i 个人, 至少获得 j 利润的方案数
这里使用了空间优化的二维 DP 数组, 相当于 dp[i][j][k] 压缩为 dp[j][k]
dp = [[0 for _ in range(minProfit + 1)] for _ in range(n + 1)]
```

```

初始化: 不选择任何工作, 使用 0 个人, 获得 0 利润的方案数为 1
这是动态规划的边界条件
for i in range(n + 1):
 dp[i][0] = 1

遍历每个工作 (物品)
这相当于 01 背包中的物品遍历
for i in range(len(group)):
 # 获取当前工作需要的人数和产生的利润
 members = group[i]
 earn = profit[i]

 # 01 背包需要倒序遍历, 确保每个物品只使用一次
 # j 表示当前可用的人数, k 表示当前需要达到的利润
 for j in range(n, members - 1, -1):
 for k in range(minProfit, -1, -1):
 # 状态转移方程:
 # dp[j][k] = 不选择当前工作 + 选择当前工作
 # 不选择当前工作: dp[j][k] (保持原值)
 # 选择当前工作: 需要 members 个人, 获得 earn 利润
 # dp[j-members][max(0, k-earn)] 表示使用 j-members 个人, 至少获得 max(0, k-earn) 利润
 # 注意: 如果 k < earn, 则至少获得 0 利润 (因为负利润没有意义)
 dp[j][k] = (dp[j][k] + dp[j - members][max(0, k - earn)]) % MOD

 # 返回使用不超过 n 个人, 至少获得 minProfit 利润的方案数
 return dp[n][minProfit]

```

, , ,

示例:

输入: n = 5, minProfit = 3, group = [2, 2], profit = [2, 3]

输出: 2

解释: 至少产生 3 利润的计划有 2 种: 完成工作 1 和工作 2, 或仅完成工作 2。

输入: n = 10, minProfit = 5, group = [2, 3, 5], profit = [6, 7, 8]

输出: 7

解释: 至少产生 5 利润的计划有 7 种。

时间复杂度:  $O(\text{len}(\text{group}) * n * \text{minProfit})$

- 外层循环遍历所有工作:  $O(\text{len}(\text{group}))$
- 中层循环遍历人数:  $O(n)$
- 内层循环遍历利润:  $O(\text{minProfit})$

空间复杂度:  $O(n * \text{minProfit})$

- 二维 DP 数组的空间消耗

,,

=====

文件: Code09\_Knapsack1.cpp

=====

```
// AtCoder Educational DP Contest D - Knapsack 1
// 题目描述: 有 N 个物品, 每个物品有重量 w_i 和价值 v_i。
// 背包容量为 W, 求能装入背包的物品的最大价值总和。
// 链接: https://atcoder.jp/contests/dp/tasks/dp_d
//
// 解题思路:
// 这是经典的 01 背包问题。
// 1. dp[i][j] 表示前 i 个物品, 背包容量为 j 时能获得的最大价值
// 2. 状态转移方程:
// dp[i][j] = max(dp[i-1][j], dp[i-1][j-w[i]] + v[i]) (当 j >= w[i] 时)
// dp[i][j] = dp[i-1][j] (当 j < w[i] 时)
// 3. 可以使用滚动数组优化空间复杂度
//
// 时间复杂度: O(N * W)
// 空间复杂度: O(W)
```

```
#define MAXW 100001
```

```
// 获取两个数中的较大值
```

```
long long max(long long a, long long b) {
 return a > b ? a : b;
}
```

```
/**
```

```
* 计算 01 背包问题的最大价值
```

```
*
```

```
* 解题思路:
```

```
* 这是经典的 01 背包问题。
```

```
* 1. dp[i][j] 表示前 i 个物品, 背包容量为 j 时能获得的最大价值
```

```
* 2. 状态转移方程:
```

```
* dp[i][j] = max(dp[i-1][j], dp[i-1][j-w[i]] + v[i]) (当 j >= w[i] 时)
```

```
* dp[i][j] = dp[i-1][j] (当 j < w[i] 时)
```

```
* 3. 可以使用滚动数组优化空间复杂度
```

```
*
```

```
* 参数:
```

```
* N: 物品数量
```

```

* W: 背包容量
* weights: 物品重量数组
* values: 物品价值数组
* 返回值:
* 能装入背包的物品的最大价值总和
*/
long long knapsack(int N, int W, int* weights, int* values) {
 // dp[j] 表示背包容量为 j 时能获得的最大价值
 // 这里使用了空间优化的一维 DP 数组，相当于 dp[i][j] 压缩为 dp[j]
 long long dp[MAXW];

 // 初始化 dp 数组
 for (int i = 0; i <= W; i++) {
 dp[i] = 0;
 }

 // 遍历每个物品（01 背包的物品遍历）
 for (int i = 0; i < N; i++) {
 // 获取当前物品的重量和价值
 int weight = weights[i];
 int value = values[i];

 // 01 背包需要倒序遍历，确保每个物品只使用一次
 // j 表示当前背包的容量
 for (int j = W; j >= weight; j--) {
 // 状态转移方程：
 // dp[j] = max(不选择当前物品, 选择当前物品)
 // 不选择当前物品: dp[j] (保持原值)
 // 选择当前物品: dp[j - weight] + value (前一个状态+当前物品价值)
 dp[j] = max(dp[j], dp[j - weight] + value);
 }
 }

 // 返回背包容量为 W 时能获得的最大价值
 return dp[W];
}

/*
* 示例:
* 输入: N = 3, W = 8
* weights = [3, 4, 5]
* values = [30, 50, 60]
* 输出: 90

```

```

* 解释: 选择物品 1 和物品 3, 总重量 3+5=8, 总价值 30+60=90
*
* 输入: N = 5, W = 5
* weights = [1, 1, 1, 1, 1]
* values = [1000000000, 1000000000, 1000000000, 1000000000, 1000000000]
* 输出: 5000000000
* 解释: 选择所有物品, 总重量 5, 总价值 5000000000
*
* 时间复杂度: O(N * W)
* - 外层循环遍历所有物品: O(N)
* - 内层循环遍历背包容量: O(W)
* 空间复杂度: O(W)
* - 一维 DP 数组的空间消耗
*/

```

---

文件: Code09\_Knapsack1.java

---

```

package class073;

// AtCoder Educational DP Contest D - Knapsack 1
// 题目描述: 有 N 个物品, 每个物品有重量 w_i 和价值 v_i。
// 背包容量为 W, 求能装入背包的物品的最大价值总和。
// 链接: https://atcoder.jp/contests/dp/tasks/dp_d
//
// 解题思路:
// 这是经典的 01 背包问题。
// 1. dp[i][j] 表示前 i 个物品, 背包容量为 j 时能获得的最大价值
// 2. 状态转移方程:
// dp[i][j] = max(dp[i-1][j], dp[i-1][j-w[i]] + v[i]) (当 j >= w[i] 时)
// dp[i][j] = dp[i-1][j] (当 j < w[i] 时)
// 3. 可以使用滚动数组优化空间复杂度
//
// 时间复杂度: O(N * W)
// 空间复杂度: O(W)

```

```
public class Code09_Knapsack1 {
```

```

 /**
 * 计算 01 背包问题的最大价值
 *
 * 解题思路:

```

```

* 这是经典的 01 背包问题。
* 1. dp[i][j] 表示前 i 个物品，背包容量为 j 时能获得的最大价值
* 2. 状态转移方程：
* dp[i][j] = max(dp[i-1][j], dp[i-1][j-w[i]] + v[i]) (当 j >= w[i] 时)
* dp[i][j] = dp[i-1][j] (当 j < w[i] 时)
* 3. 可以使用滚动数组优化空间复杂度
*
* @param N 物品数量
* @param W 背包容量
* @param weights 物品重量数组
* @param values 物品价值数组
* @return 能装入背包的物品的最大价值总和
*/
public static long knapsack(int N, int W, int[] weights, int[] values) {
 // dp[j] 表示背包容量为 j 时能获得的最大价值
 // 这里使用了空间优化的一维 DP 数组，相当于 dp[i][j] 压缩为 dp[j]
 long[] dp = new long[W + 1];

 // 遍历每个物品（01 背包的物品遍历）
 for (int i = 0; i < N; i++) {
 // 获取当前物品的重量和价值
 int weight = weights[i];
 int value = values[i];

 // 01 背包需要倒序遍历，确保每个物品只使用一次
 // j 表示当前背包的容量
 for (int j = W; j >= weight; j--) {
 // 状态转移方程：
 // dp[j] = max(不选择当前物品, 选择当前物品)
 // 不选择当前物品: dp[j] (保持原值)
 // 选择当前物品: dp[j - weight] + value (前一个状态+当前物品价值)
 dp[j] = Math.max(dp[j], dp[j - weight] + value);
 }
 }

 // 返回背包容量为 W 时能获得的最大价值
 return dp[W];
}

/*
* 示例：
* 输入：N = 3, W = 8
* weights = [3, 4, 5]

```

```

* values = [30, 50, 60]
* 输出: 90
* 解释: 选择物品 1 和物品 3, 总重量 3+5=8, 总价值 30+60=90
*
* 输入: N = 5, W = 5
* weights = [1, 1, 1, 1, 1]
* values = [1000000000, 1000000000, 1000000000, 1000000000, 1000000000]
* 输出: 5000000000
* 解释: 选择所有物品, 总重量 5, 总价值 5000000000
*
* 时间复杂度: O(N * W)
* - 外层循环遍历所有物品: O(N)
* - 内层循环遍历背包容量: O(W)
* 空间复杂度: O(W)
* - 一维 DP 数组的空间消耗
*/
}

```

文件: Code09\_Knapsack1.py

```

AtCoder Educational DP Contest D – Knapsack 1
题目描述: 有 N 个物品, 每个物品有重量 w_i 和价值 v_i。
背包容量为 W, 求能装入背包的物品的最大价值总和。
链接: https://atcoder.jp/contests/dp/tasks/dp_d
#
解题思路:
这是经典的 01 背包问题。
1. dp[i][j] 表示前 i 个物品, 背包容量为 j 时能获得的最大价值
2. 状态转移方程:
dp[i][j] = max(dp[i-1][j], dp[i-1][j-w[i]] + v[i]) (当 j >= w[i] 时)
dp[i][j] = dp[i-1][j] (当 j < w[i] 时)
3. 可以使用滚动数组优化空间复杂度
#
时间复杂度: O(N * W)
空间复杂度: O(W)

```

```
def knapsack(N, W, weights, values):
```

```
 """

```

计算 01 背包问题的最大价值

解题思路:

这是经典的 01 背包问题。

1.  $dp[i][j]$  表示前  $i$  个物品，背包容量为  $j$  时能获得的最大价值

2. 状态转移方程：

$$dp[i][j] = \max(dp[i-1][j], dp[i-1][j-w[i]] + v[i]) \quad (\text{当 } j \geq w[i] \text{ 时})$$

$$dp[i][j] = dp[i-1][j] \quad (\text{当 } j < w[i] \text{ 时})$$

3. 可以使用滚动数组优化空间复杂度

Args:

N: 物品数量

W: 背包容量

weights: 物品重量数组

values: 物品价值数组

Returns:

能装入背包的物品的最大价值总和

"""

```
dp[j] 表示背包容量为 j 时能获得的最大价值
```

```
这里使用了空间优化的一维 DP 数组，相当于 dp[i][j] 压缩为 dp[j]
```

```
dp = [0] * (W + 1)
```

```
遍历每个物品（01 背包的物品遍历）
```

```
for i in range(N):
```

```
 # 获取当前物品的重量和价值
```

```
 weight = weights[i]
```

```
 value = values[i]
```

```
01 背包需要倒序遍历，确保每个物品只使用一次
```

```
j 表示当前背包的容量
```

```
for j in range(W, weight - 1, -1):
```

```
 # 状态转移方程：
```

```
 # dp[j] = max(不选择当前物品, 选择当前物品)
```

```
 # 不选择当前物品: dp[j] (保持原值)
```

```
 # 选择当前物品: dp[j - weight] + value (前一个状态+当前物品价值)
```

```
 dp[j] = max(dp[j], dp[j - weight] + value)
```

```
返回背包容量为 W 时能获得的最大价值
```

```
return dp[W]
```

,,

示例：

输入：N = 3, W = 8

weights = [3, 4, 5]

values = [30, 50, 60]

输出: 90

解释: 选择物品 1 和物品 3, 总重量  $3+5=8$ , 总价值  $30+60=90$

输入:  $N = 5, W = 5$

`weights = [1, 1, 1, 1, 1]`

`values = [1000000000, 1000000000, 1000000000, 1000000000, 1000000000]`

输出: 5000000000

解释: 选择所有物品, 总重量 5, 总价值 5000000000

时间复杂度:  $O(N * W)$

- 外层循环遍历所有物品:  $O(N)$

- 内层循环遍历背包容量:  $O(W)$

空间复杂度:  $O(W)$

- 一维 DP 数组的空间消耗

, , ,

=====

文件: Code10\_PackingProblem.cpp

=====

// 洛谷 P1049 [NOIP2001 普及组] 装箱问题

// 题目描述: 有一个箱子容量为  $V$ , 同时有  $n$  个物品, 每个物品有一个体积。

// 现在从  $n$  个物品中, 任取若干个装入箱内 (也可以不取), 使箱子的剩余空间最小。输出这个最小值。

// 链接: <https://www.luogu.com.cn/problem/P1049>

//

// 解题思路:

// 这是 01 背包问题的变形。

// 1. 目标是使箱子剩余空间最小, 等价于使装入物品的总体积最大

// 2. 每个物品的“价值”等于其体积

// 3. `dp[i][j]` 表示前  $i$  个物品, 背包容量为  $j$  时能装入的最大体积

// 4. 状态转移方程:

//  $dp[i][j] = \max(dp[i-1][j], dp[i-1][j-volume[i]] + volume[i])$  (当  $j \geq volume[i]$  时)

//  $dp[i][j] = dp[i-1][j]$  (当  $j < volume[i]$  时)

// 5. 最终答案是  $V - dp[n][V]$

//

// 时间复杂度:  $O(n * V)$

// 空间复杂度:  $O(V)$

#define MAXV 20001

// 获取两个数中的较大值

int max(int a, int b) {

return a > b ? a : b;

```
}
```

```
/**
 * 计算箱子的最小剩余空间
 *
 * 解题思路:
 * 这是 01 背包问题的变形。
 * 1. 目标是使箱子剩余空间最小，等价于使装入物品的总体积最大
 * 2. 每个物品的“价值”等于其体积
 * 3. dp[i][j] 表示前 i 个物品，背包容量为 j 时能装入的最大体积
 * 4. 状态转移方程:
 * dp[i][j] = max(dp[i-1][j], dp[i-1][j-volume[i]] + volume[i]) (当 j >= volume[i] 时)
 * dp[i][j] = dp[i-1][j] (当 j < volume[i] 时)
 * 5. 最终答案是 V - dp[n][V]
 *
 * 参数:
 * V: 箱子容量
 * n: 物品数量
 * volumes: 物品体积数组
 * 返回值:
 * 箱子的最小剩余空间
 */
```

```
int minRemainingSpace(int V, int n, int* volumes) {
 // dp[j] 表示背包容量为 j 时能装入的最大体积
 // 这里使用了空间优化的一维 DP 数组，相当于 dp[i][j] 压缩为 dp[j]
 int dp[MAXV];

 // 初始化 dp 数组
 for (int i = 0; i <= V; i++) {
 dp[i] = 0;
 }

 // 遍历每个物品 (01 背包的物品遍历)
 for (int i = 0; i < n; i++) {
 // 获取当前物品的体积
 int volume = volumes[i];

 // 01 背包需要倒序遍历，确保每个物品只使用一次
 // j 表示当前背包的容量
 for (int j = V; j >= volume; j--) {
 // 状态转移方程:
 // dp[j] = max(不选择当前物品, 选择当前物品)
 // 不选择当前物品: dp[j] (保持原值)
```

```

 // 选择当前物品: dp[j - volume] + volume (前一个状态+当前物品体积)
 dp[j] = max(dp[j], dp[j - volume] + volume);
 }
}

// 箱子剩余空间 = 总容量 - 装入物品的最大体积
return V - dp[V];
}

/*
 * 示例:
 * 输入: V = 24, n = 6
 * volumes = [8, 3, 12, 7, 9, 7]
 * 输出: 0
 * 解释: 可以恰好装满箱子, 剩余空间为 0
 *
 * 输入: V = 10, n = 3
 * volumes = [3, 4, 5]
 * 输出: 1
 * 解释: 最多装入体积为 9 的物品, 剩余空间为 1
 *
 * 时间复杂度: O(n * V)
 * - 外层循环遍历所有物品: O(n)
 * - 内层循环遍历背包容量: O(V)
 * 空间复杂度: O(V)
 * - 一维 DP 数组的空间消耗
*/

```

文件: Code10\_PackingProblem.java

```

package class073;

// 洛谷 P1049 [NOIP2001 普及组] 装箱问题
// 题目描述: 有一个箱子容量为 V, 同时有 n 个物品, 每个物品有一个体积。
// 现在从 n 个物品中, 任取若干个装入箱内 (也可以不取), 使箱子的剩余空间最小。输出这个最小值。
// 链接: https://www.luogu.com.cn/problem/P1049
//
// 解题思路:
// 这是 01 背包问题的变形。
// 1. 目标是使箱子剩余空间最小, 等价于使装入物品的总体积最大
// 2. 每个物品的“价值”等于其体积

```

```

// 3. dp[i][j] 表示前 i 个物品，背包容量为 j 时能装入的最大体积
// 4. 状态转移方程：
// dp[i][j] = max(dp[i-1][j], dp[i-1][j-volume[i]] + volume[i]) (当 j >= volume[i] 时)
// dp[i][j] = dp[i-1][j] (当 j < volume[i] 时)
// 5. 最终答案是 V - dp[n][V]
//
// 时间复杂度：O(n * V)
// 空间复杂度：O(V)

public class Code10_PackingProblem {

 /**
 * 计算箱子的最小剩余空间
 *
 * 解题思路：
 * 这是 01 背包问题的变形。
 * 1. 目标是使箱子剩余空间最小，等价于使装入物品的总体积最大
 * 2. 每个物品的“价值”等于其体积
 * 3. dp[i][j] 表示前 i 个物品，背包容量为 j 时能装入的最大体积
 * 4. 状态转移方程：
 * dp[i][j] = max(dp[i-1][j], dp[i-1][j-volume[i]] + volume[i]) (当 j >= volume[i] 时)
 * dp[i][j] = dp[i-1][j] (当 j < volume[i] 时)
 * 5. 最终答案是 V - dp[n][V]
 *
 * @param V 箱子容量
 * @param n 物品数量
 * @param volumes 物品体积数组
 * @return 箱子的最小剩余空间
 */

 public static int minRemainingSpace(int V, int n, int[] volumes) {
 // dp[j] 表示背包容量为 j 时能装入的最大体积
 // 这里使用了空间优化的一维 DP 数组，相当于 dp[i][j] 压缩为 dp[j]
 int[] dp = new int[V + 1];

 // 遍历每个物品（01 背包的物品遍历）
 for (int i = 0; i < n; i++) {
 // 获取当前物品的体积
 int volume = volumes[i];

 // 01 背包需要倒序遍历，确保每个物品只使用一次
 // j 表示当前背包的容量
 for (int j = V; j >= volume; j--) {
 // 状态转移方程：
 dp[j] = Math.max(dp[j], dp[j - volume] + volume);
 }
 }

 return V - dp[V];
 }
}

```

```

 // dp[j] = max(不选择当前物品, 选择当前物品)
 // 不选择当前物品: dp[j] (保持原值)
 // 选择当前物品: dp[j - volume] + volume (前一个状态+当前物品体积)
 dp[j] = Math.max(dp[j], dp[j - volume] + volume);
 }
}

// 箱子剩余空间 = 总容量 - 装入物品的最大体积
return V - dp[V];
}

/*
 * 示例:
 * 输入: V = 24, n = 6
 * volumes = [8, 3, 12, 7, 9, 7]
 * 输出: 0
 * 解释: 可以恰好装满箱子, 剩余空间为 0
 *
 * 输入: V = 10, n = 3
 * volumes = [3, 4, 5]
 * 输出: 1
 * 解释: 最多装入体积为 9 的物品, 剩余空间为 1
 *
 * 时间复杂度: O(n * V)
 * - 外层循环遍历所有物品: O(n)
 * - 内层循环遍历背包容量: O(V)
 * 空间复杂度: O(V)
 * - 一维 DP 数组的空间消耗
 */
}

```

文件: Code10\_PackingProblem.py

```

=====
洛谷 P1049 [NOIP2001 普及组] 装箱问题
题目描述: 有一个箱子容量为 V, 同时有 n 个物品, 每个物品有一个体积。
现在从 n 个物品中, 任取若干个装入箱内 (也可以不取), 使箱子的剩余空间最小。输出这个最小值。
链接: https://www.luogu.com.cn/problem/P1049
#
解题思路:
这是 01 背包问题的变形。
1. 目标是使箱子剩余空间最小, 等价于使装入物品的总体积最大

```

```

2. 每个物品的“价值”等于其体积
3. dp[i][j] 表示前 i 个物品，背包容量为 j 时能装入的最大体积
4. 状态转移方程：
dp[i][j] = max(dp[i-1][j], dp[i-1][j-volume[i]] + volume[i]) (当 j >= volume[i] 时)
dp[i][j] = dp[i-1][j] (当 j < volume[i] 时)
5. 最终答案是 V - dp[n][V]
#
时间复杂度：O(n * V)
空间复杂度：O(V)

```

```
def minRemainingSpace(V, n, volumes):
```

```
 """

```

计算箱子的最小剩余空间

解题思路：

这是 01 背包问题的变形。

1. 目标是使箱子剩余空间最小，等价于使装入物品的总体积最大
2. 每个物品的“价值”等于其体积
3. dp[i][j] 表示前 i 个物品，背包容量为 j 时能装入的最大体积
4. 状态转移方程：
 
$$dp[i][j] = \max(dp[i-1][j], dp[i-1][j-volume[i]] + volume[i]) \quad (\text{当 } j \geq volume[i] \text{ 时})$$

$$dp[i][j] = dp[i-1][j] \quad (\text{当 } j < volume[i] \text{ 时})$$
5. 最终答案是  $V - dp[n][V]$

Args:

V: 箱子容量

n: 物品数量

volumes: 物品体积数组

Returns:

箱子的最小剩余空间

```
"""

```

```
dp[j] 表示背包容量为 j 时能装入的最大体积
这里使用了空间优化的一维 DP 数组，相当于 dp[i][j] 压缩为 dp[j]
dp = [0] * (V + 1)

遍历每个物品（01 背包的物品遍历）
for i in range(n):
 # 获取当前物品的体积
 volume = volumes[i]

 # 01 背包需要倒序遍历，确保每个物品只使用一次
 # j 表示当前背包的容量
```

```

for j in range(V, volume - 1, -1):
 # 状态转移方程:
 # dp[j] = max(不选择当前物品, 选择当前物品)
 # 不选择当前物品: dp[j] (保持原值)
 # 选择当前物品: dp[j - volume] + volume (前一个状态+当前物品体积)
 dp[j] = max(dp[j], dp[j - volume] + volume)

 # 箱子剩余空间 = 总容量 - 装入物品的最大体积
return V - dp[V]

```

,,

示例:

输入: V = 24, n = 6

volumes = [8, 3, 12, 7, 9, 7]

输出: 0

解释: 可以恰好装满箱子, 剩余空间为 0

输入: V = 10, n = 3

volumes = [3, 4, 5]

输出: 1

解释: 最多装入体积为 9 的物品, 剩余空间为 1

时间复杂度:  $O(n * V)$

- 外层循环遍历所有物品:  $O(n)$
- 内层循环遍历背包容量:  $O(V)$

空间复杂度:  $O(V)$

- 一维 DP 数组的空间消耗

,,

---

文件: Code11\_HappyJinming.cpp

---

```

// 洛谷 P1060 [NOIP2006 普及组] 开心的金明
// 题目描述: 金明需要购买物品, 在不超过 N 元的前提下, 使每件物品的价格与重要度的乘积的总和最大。
// 链接: https://www.luogu.com.cn/problem/P1060
//
// 解题思路:
// 这是经典的 01 背包问题。
// 1. 每个物品的“价值”是价格与重要度的乘积
// 2. dp[i][j] 表示前 i 个物品, 预算为 j 时能获得的最大价值总和
// 3. 状态转移方程:
// dp[i][j] = max(dp[i-1][j], dp[i-1][j-price[i]] + price[i] * importance[i]) (当 j >=

```

```

price[i]时)
// dp[i][j] = dp[i-1][j] (当 j < price[i]时)
// 4. 可以使用滚动数组优化空间复杂度
//
// 时间复杂度: O(m * n)
// 空间复杂度: O(n)

#define MAXN 30001

// 获取两个数中的较大值
int max(int a, int b) {
 return a > b ? a : b;
}

/***
 * 计算开心的金明能获得的最大满意度
 *
 * 解题思路:
 * 这是经典的 01 背包问题。
 * 1. 每个物品的“价值”是价格与重要度的乘积
 * 2. dp[i][j] 表示前 i 个物品，预算为 j 时能获得的最大价值总和
 * 3. 状态转移方程:
 * dp[i][j] = max(dp[i-1][j], dp[i-1][j-price[i]] + price[i] * importance[i]) (当 j >= price[i]时)
 * dp[i][j] = dp[i-1][j] (当 j < price[i]时)
 * 4. 可以使用滚动数组优化空间复杂度
 *
 * 参数:
 * n: 总预算
 * m: 物品数量
 * prices: 物品价格数组
 * importances: 物品重要度数组
 * 返回值:
 * 能获得的最大满意度
*/
int maxSatisfaction(int n, int m, int* prices, int* importances) {
 // dp[j] 表示预算为 j 时能获得的最大价值总和
 // 这里使用了空间优化的一维 DP 数组，相当于 dp[i][j] 压缩为 dp[j]
 int dp[MAXN];

 // 初始化 dp 数组
 for (int i = 0; i <= n; i++) {
 dp[i] = 0;
 }
}

```

```

}

// 遍历每个物品 (01 背包的物品遍历)
for (int i = 0; i < m; i++) {
 // 获取当前物品的价格和重要度
 int price = prices[i];
 int importance = importances[i];
 // 计算当前物品的价值 = 价格 * 重要度
 int value = price * importance;

 // 01 背包需要倒序遍历，确保每个物品只使用一次
 // j 表示当前的预算
 for (int j = n; j >= price; j--) {
 // 状态转移方程：
 // dp[j] = max(不选择当前物品, 选择当前物品)
 // 不选择当前物品: dp[j] (保持原值)
 // 选择当前物品: dp[j - price] + value (前一个状态+当前物品价值)
 dp[j] = max(dp[j], dp[j - price] + value);
 }
}

// 返回预算为 n 时能获得的最大价值总和
return dp[n];
}

/*
 * 示例：
 * 输入：n = 1000, m = 5
 * prices = [800, 400, 300, 400, 200]
 * importances = [2, 5, 5, 3, 2]
 * 输出：3900
 * 解释：选择物品 2, 3, 4， 总价格 400+300+400=1100，但超过预算，需要重新选择
 *
 * 时间复杂度：O(m * n)
 * - 外层循环遍历所有物品：O(m)
 * - 内层循环遍历预算：O(n)
 * 空间复杂度：O(n)
 * - 一维 DP 数组的空间消耗
 */
=====
```

```
=====
package class073;

// 洛谷 P1060 [NOIP2006 普及组] 开心的金明
// 题目描述: 金明需要购买物品, 在不超过 N 元的前提下, 使每件物品的价格与重要度的乘积的总和最大。
// 链接: https://www.luogu.com.cn/problem/P1060
//
// 解题思路:
// 这是经典的 01 背包问题。
// 1. 每个物品的“价值”是价格与重要度的乘积
// 2. dp[i][j] 表示前 i 个物品, 预算为 j 时能获得的最大价值总和
// 3. 状态转移方程:
// dp[i][j] = max(dp[i-1][j], dp[i-1][j-price[i]] + price[i] * importance[i]) (当 j >= price[i] 时)
// dp[i][j] = dp[i-1][j] (当 j < price[i] 时)
// 4. 可以使用滚动数组优化空间复杂度
//
// 时间复杂度: O(m * n)
// 空间复杂度: O(n)
```

```
public class Code11_HappyJinming {
```

```
 /**
 * 计算开心的金明能获得的最大满意度
 *
 * 解题思路:
 * 这是经典的 01 背包问题。
 * 1. 每个物品的“价值”是价格与重要度的乘积
 * 2. dp[i][j] 表示前 i 个物品, 预算为 j 时能获得的最大价值总和
 * 3. 状态转移方程:
 * dp[i][j] = max(dp[i-1][j], dp[i-1][j-price[i]] + price[i] * importance[i]) (当 j >= price[i] 时)
 * dp[i][j] = dp[i-1][j] (当 j < price[i] 时)
 * 4. 可以使用滚动数组优化空间复杂度
 *
 * @param n 总预算
 * @param m 物品数量
 * @param prices 物品价格数组
 * @param importances 物品重要度数组
 * @return 能获得的最大满意度
 */
 public static int maxSatisfaction(int n, int m, int[] prices, int[] importances) {
 // dp[j] 表示预算为 j 时能获得的最大价值总和
```

```

// 这里使用了空间优化的一维 DP 数组，相当于 dp[i][j] 压缩为 dp[j]
int[] dp = new int[n + 1];

// 遍历每个物品（01 背包的物品遍历）
for (int i = 0; i < m; i++) {
 // 获取当前物品的价格和重要度
 int price = prices[i];
 int importance = importances[i];
 // 计算当前物品的价值 = 价格 * 重要度
 int value = price * importance;

 // 01 背包需要倒序遍历，确保每个物品只使用一次
 // j 表示当前的预算
 for (int j = n; j >= price; j--) {
 // 状态转移方程：
 // dp[j] = max(不选择当前物品, 选择当前物品)
 // 不选择当前物品: dp[j] (保持原值)
 // 选择当前物品: dp[j - price] + value (前一个状态+当前物品价值)
 dp[j] = Math.max(dp[j], dp[j - price] + value);
 }
}

// 返回预算为 n 时能获得的最大价值总和
return dp[n];
}

/*
 * 示例：
 * 输入: n = 1000, m = 5
 * prices = [800, 400, 300, 400, 200]
 * importances = [2, 5, 5, 3, 2]
 * 输出: 3900
 * 解释: 选择物品 2,3,4, 总价格 400+300+400=1100, 但超过预算, 需要重新选择
 *
 * 时间复杂度: O(m * n)
 * - 外层循环遍历所有物品: O(m)
 * - 内层循环遍历预算: O(n)
 * 空间复杂度: O(n)
 * - 一维 DP 数组的空间消耗
 */
}
=====
```

文件: Code11\_HappyJinming.py

```
=====
洛谷 P1060 [NOIP2006 普及组] 开心的金明
题目描述: 金明需要购买物品, 在不超过 N 元的前提下, 使每件物品的价格与重要度的乘积的总和最大。
链接: https://www.luogu.com.cn/problem/P1060
#
解题思路:
这是经典的 01 背包问题。
1. 每个物品的“价值”是价格与重要度的乘积
2. dp[i][j] 表示前 i 个物品, 预算为 j 时能获得的最大价值总和
3. 状态转移方程:
dp[i][j] = max(dp[i-1][j], dp[i-1][j-price[i]] + price[i] * importance[i]) (当 j >= price[i] 时)
dp[i][j] = dp[i-1][j] (当 j < price[i] 时)
4. 可以使用滚动数组优化空间复杂度
#
时间复杂度: O(m * n)
空间复杂度: O(n)
```

```
def maxSatisfaction(n, m, prices, importances):
```

```
 """

```

计算开心的金明能获得的最大满意度

解题思路:

这是经典的 01 背包问题。

1. 每个物品的“价值”是价格与重要度的乘积

2. dp[i][j] 表示前 i 个物品, 预算为 j 时能获得的最大价值总和

3. 状态转移方程:

```
 dp[i][j] = max(dp[i-1][j], dp[i-1][j-price[i]] + price[i] * importance[i]) (当 j >= price[i] 时)
```

```
 dp[i][j] = dp[i-1][j] (当 j < price[i] 时)
```

4. 可以使用滚动数组优化空间复杂度

Args:

n: 总预算

m: 物品数量

prices: 物品价格数组

importances: 物品重要度数组

Returns:

能获得的最大满意度

```
"""

```

```

dp[j] 表示预算为 j 时能获得的最大价值总和
这里使用了空间优化的一维 DP 数组，相当于 dp[i][j] 压缩为 dp[j]
dp = [0] * (n + 1)

遍历每个物品（01 背包的物品遍历）
for i in range(m):
 # 获取当前物品的价格和重要度
 price = prices[i]
 importance = importances[i]
 # 计算当前物品的价值 = 价格 * 重要度
 value = price * importance

 # 01 背包需要倒序遍历，确保每个物品只使用一次
 # j 表示当前的预算
 for j in range(n, price - 1, -1):
 # 状态转移方程：
 # dp[j] = max(不选择当前物品, 选择当前物品)
 # 不选择当前物品: dp[j] (保持原值)
 # 选择当前物品: dp[j - price] + value (前一个状态+当前物品价值)
 dp[j] = max(dp[j], dp[j - price] + value)

返回预算为 n 时能获得的最大价值总和
return dp[n]

```

,,

示例：

输入：n = 1000, m = 5

prices = [800, 400, 300, 400, 200]

importances = [2, 5, 5, 3, 2]

输出：3900

解释：选择物品 2, 3, 4，总价格  $400+300+400=1100$ ，但超过预算，需要重新选择

时间复杂度： $O(m * n)$

- 外层循环遍历所有物品： $O(m)$
- 内层循环遍历预算： $O(n)$

空间复杂度： $O(n)$

- 一维 DP 数组的空间消耗

,,

---

文件：Code12\_Balance.cpp

---

```
// POJ 1837 Balance
// 题目描述：有一个天平，有两个臂长为 15 的臂，上面有 C 个挂钩，位置在-15 到 15 之间。
// 有 G 个重量不同的砝码，重量在 1 到 25 之间。要求将所有砝码都挂在挂钩上，求有多少种方法使天平平衡。
// 链接: http://poj.org/problem?id=1837
//
// 解题思路：
// 这是一个 01 背包问题的变形。
// 1. 每个砝码是物品，每个挂钩位置是费用
// 2. 状态定义: dp[i][j] 表示前 i 个砝码，达到力矩平衡值为 j 的方法数
// 3. 力矩平衡值的计算: 左边为负，右边为正，平衡时总和为 0
// 4. 由于力矩可能为负数，需要平移坐标轴，将-7500~7500 映射到 0~15000
// 5. 状态转移方程:
// dp[i][j + weight[i] * hook[k]] += dp[i-1][j]
// 6. 初始状态: dp[0][7500] = 1 (平移后的 0 点)
//
// 时间复杂度: O(G * C * 15000)
// 空间复杂度: O(G * 15000)
```

```
#define MAX_SUM 15001
#define OFFSET 7500
```

```
/**
 * 计算使天平平衡的方法数
 *
 * 解题思路：
 * 这是一个 01 背包问题的变形。
 * 1. 每个砝码是物品，每个挂钩位置是费用
 * 2. 状态定义: dp[i][j] 表示前 i 个砝码，达到力矩平衡值为 j 的方法数
 * 3. 力矩平衡值的计算: 左边为负，右边为正，平衡时总和为 0
 * 4. 由于力矩可能为负数，需要平移坐标轴，将-7500~7500 映射到 0~15000
 * 5. 状态转移方程:
 * dp[i][j + weight[i] * hook[k]] += dp[i-1][j]
 * 6. 初始状态: dp[0][7500] = 1 (平移后的 0 点)
 *
 * 参数:
 * C: 挂钩数量
 * G: 砝码数量
 * hooks: 挂钩位置数组
 * weights: 砝码重量数组
 * 返回值:
 * 使天平平衡的方法数
 */
```

```

int balanceWays(int C, int G, int* hooks, int* weights) {
 // dp[i][j] 表示前 i 个砝码，达到力矩平衡值为 j-OFFSET 的方法数
 // 这里 j-OFFSET 是实际的力矩值，j 是平移后的索引
 // 最多 20 个砝码
 int dp[21][MAX_SUM];

 // 初始化 dp 数组
 for (int i = 0; i <= G; i++) {
 for (int j = 0; j < MAX_SUM; j++) {
 dp[i][j] = 0;
 }
 }

 // 初始状态：不放任何砝码，平衡值为 0 (平移后为 OFFSET)
 // 这是动态规划的边界条件
 dp[0][OFFSET] = 1;

 // 遍历每个砝码 (物品)
 for (int i = 1; i <= G; i++) {
 // 获取当前砝码的重量
 int weight = weights[i - 1];

 // 遍历前一个状态的所有可能平衡值
 for (int j = 0; j < MAX_SUM; j++) {
 // 如果前一个状态有方法能达到平衡值 j
 if (dp[i - 1][j] > 0) {
 // 尝试将当前砝码挂在每个挂钩上
 for (int k = 0; k < C; k++) {
 // 获取当前挂钩的位置
 int hookPos = hooks[k];
 // 计算放置当前砝码后的新平衡值
 // 力矩 = 重量 * 位置，左边为负，右边为正
 // 使用 long long 防止溢出
 long long newBalance = (long long)j + (long long)weight * hookPos;

 // 检查新的平衡值是否在有效范围内
 if (newBalance >= 0 && newBalance < MAX_SUM) {
 // 状态转移：将前一个状态的方法数累加到新状态
 dp[i][newBalance] += dp[i - 1][j];
 }
 }
 }
 }
 }
}

```

```

 }

 // 返回所有砝码放完后平衡值为 0 (平移后为 OFFSET) 的方法数
 // 平衡值为 0 表示天平平衡
 return dp[G][OFFSET];
}

/*
 * 示例:
 * 输入: C = 2, G = 4
 * hooks = [-2, 3]
 * weights = [3, 4, 5, 8]
 * 输出: 2
 * 解释: 有两种方法可以使天平平衡
 *
 * 时间复杂度: O(G * C * 15000)
 * - 外层循环遍历所有砝码: O(G)
 * - 中层循环遍历所有可能的平衡值: O(15000)
 * - 内层循环遍历所有挂钩: O(C)
 * 空间复杂度: O(G * 15000)
 * - 二维 DP 数组的空间消耗
*/

```

=====

文件: Code12\_Balance.java

=====

```

package class073;

// POJ 1837 Balance
// 题目描述: 有一个天平, 有两个臂长为 15 的臂, 上面有 C 个挂钩, 位置在-15 到 15 之间。
// 有 G 个重量不同的砝码, 重量在 1 到 25 之间。要求将所有砝码都挂在挂钩上, 求有多少种方法使天平平衡。
// 链接: http://poj.org/problem?id=1837
//
// 解题思路:
// 这是一个 01 背包问题的变形。
// 1. 每个砝码是物品, 每个挂钩位置是费用
// 2. 状态定义: dp[i][j] 表示前 i 个砝码, 达到力矩平衡值为 j 的方法数
// 3. 力矩平衡值的计算: 左边为负, 右边为正, 平衡时总和为 0
// 4. 由于力矩可能为负数, 需要平移坐标轴, 将-7500~7500 映射到 0~15000
// 5. 状态转移方程:
// dp[i][j + weight[i] * hook[k]] += dp[i-1][j]

```

```

// 6. 初始状态: dp[0][7500] = 1 (平移后的 0 点)
//
// 时间复杂度: O(G * C * 15000)
// 空间复杂度: O(G * 15000)

public class Code12_Balance {
 private static final int OFFSET = 7500; // 坐标偏移量, 用于处理负数力矩
 private static final int MAX_SUM = 15000; // 最大力矩和, 覆盖-7500 到 7500 的范围

 /**
 * 计算使天平平衡的方法数
 *
 * 解题思路:
 * 这是一个 01 背包问题的变形。
 * 1. 每个砝码是物品, 每个挂钩位置是费用
 * 2. 状态定义: dp[i][j] 表示前 i 个砝码, 达到力矩平衡值为 j 的方法数
 * 3. 力矩平衡值的计算: 左边为负, 右边为正, 平衡时总和为 0
 * 4. 由于力矩可能为负数, 需要平移坐标轴, 将-7500~7500 映射到 0~15000
 * 5. 状态转移方程:
 * dp[i][j + weight[i] * hook[k]] += dp[i-1][j]
 * 6. 初始状态: dp[0][7500] = 1 (平移后的 0 点)
 *
 * @param C 挂钩数量
 * @param G 砝码数量
 * @param hooks 挂钩位置数组
 * @param weights 砝码重量数组
 * @return 使天平平衡的方法数
 */

 public static int balanceWays(int C, int G, int[] hooks, int[] weights) {
 // dp[i][j] 表示前 i 个砝码, 达到力矩平衡值为 j-OFFSET 的方法数
 // 这里 j-OFFSET 是实际的力矩值, j 是平移后的索引
 int[][] dp = new int[G + 1][MAX_SUM + 1];

 // 初始状态: 不放任何砝码, 平衡值为 0 (平移后为 OFFSET)
 // 这是动态规划的边界条件
 dp[0][OFFSET] = 1;

 // 遍历每个砝码 (物品)
 for (int i = 1; i <= G; i++) {
 // 获取当前砝码的重量
 int weight = weights[i - 1];

 // 遍历前一个状态的所有可能平衡值

```

```

 for (int j = 0; j <= MAX_SUM; j++) {
 // 如果前一个状态有方法能达到平衡值 j
 if (dp[i - 1][j] > 0) {
 // 尝试将当前砝码挂在每个挂钩上
 for (int k = 0; k < C; k++) {
 // 获取当前挂钩的位置
 int hookPos = hooks[k];
 // 计算放置当前砝码后的新平衡值
 // 力矩 = 重量 * 位置, 左边为负, 右边为正
 int newBalance = j + weight * hookPos;

 // 检查新的平衡值是否在有效范围内
 if (newBalance >= 0 && newBalance <= MAX_SUM) {
 // 状态转移: 将前一个状态的方法数累加到新状态
 dp[i][newBalance] += dp[i - 1][j];
 }
 }
 }
 }

 // 返回所有砝码放完后平衡值为 0 (平移后为 OFFSET) 的方法数
 // 平衡值为 0 表示天平平衡
 return dp[G][OFFSET];
 }

/*
 * 示例:
 * 输入: C = 2, G = 4
 * hooks = [-2, 3]
 * weights = [3, 4, 5, 8]
 * 输出: 2
 * 解释: 有两种方法可以使天平平衡
 *
 * 时间复杂度: O(G * C * 15000)
 * - 外层循环遍历所有砝码: O(G)
 * - 中层循环遍历所有可能的平衡值: O(15000)
 * - 内层循环遍历所有挂钩: O(C)
 * 空间复杂度: O(G * 15000)
 * - 二维 DP 数组的空间消耗
 */
}

```

文件: Code12\_Balance.py

```
POJ 1837 Balance
题目描述: 有一个天平, 有两个臂长为 15 的臂, 上面有 C 个挂钩, 位置在-15 到 15 之间。
有 G 个重量不同的砝码, 重量在 1 到 25 之间。要求将所有砝码都挂在挂钩上, 求有多少种方法使天平平衡。
链接: http://poj.org/problem?id=1837
#
解题思路:
这是一个 01 背包问题的变形。
1. 每个砝码是物品, 每个挂钩位置是费用
2. 状态定义: dp[i][j] 表示前 i 个砝码, 达到力矩平衡值为 j 的方法数
3. 力矩平衡值的计算: 左边为负, 右边为正, 平衡时总和为 0
4. 由于力矩可能为负数, 需要平移坐标轴, 将-7500~7500 映射到 0~15000
5. 状态转移方程:
dp[i][j + weight[i] * hook[k]] += dp[i-1][j]
6. 初始状态: dp[0][7500] = 1 (平移后的 0 点)
#
时间复杂度: O(G * C * 15000)
空间复杂度: O(G * 15000)
```

```
def balanceWays(C, G, hooks, weights):
```

```
 """

```

```
 计算使天平平衡的方法数

```

解题思路:

这是一个 01 背包问题的变形。

1. 每个砝码是物品, 每个挂钩位置是费用
2. 状态定义: dp[i][j] 表示前 i 个砝码, 达到力矩平衡值为 j 的方法数
3. 力矩平衡值的计算: 左边为负, 右边为正, 平衡时总和为 0
4. 由于力矩可能为负数, 需要平移坐标轴, 将-7500~7500 映射到 0~15000
5. 状态转移方程:  
$$dp[i][j + weight[i] * hook[k]] += dp[i-1][j]$$
6. 初始状态:  $dp[0][7500] = 1$  (平移后的 0 点)

Args:

C: 挂钩数量

G: 砝码数量

hooks: 挂钩位置数组

weights: 砝码重量数组

Returns:

## 使天平平衡的方法数

"""

```
OFFSET = 7500 # 坐标偏移量, 用于处理负数力矩
MAX_SUM = 15000 # 最大力矩和, 覆盖-7500 到 7500 的范围
```

```
dp[i][j] 表示前 i 个砝码, 达到力矩平衡值为 j-OFFSET 的方法数
这里 j-OFFSET 是实际的力矩值, j 是平移后的索引
```

```
dp = [[0 for _ in range(MAX_SUM + 1)] for _ in range(G + 1)]
```

```
初始状态: 不放任何砝码, 平衡值为 0 (平移后为 OFFSET)
```

```
这是动态规划的边界条件
```

```
dp[0][OFFSET] = 1
```

```
遍历每个砝码 (物品)
```

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

```
 # 获取当前砝码的重量
```

```
 weight = weights[i - 1]
```

```
遍历前一个状态的所有可能平衡值
```

```
for j in range(MAX_SUM + 1):
```

```
 # 如果前一个状态有方法能达到平衡值 j
```

```
 if dp[i - 1][j] > 0:
```

```
 # 尝试将当前砝码挂在每个挂钩上
```

```
 for k in range(C):
```

```
 # 获取当前挂钩的位置
```

```
 hookPos = hooks[k]
```

```
 # 计算放置当前砝码后的新平衡值
```

```
 # 力矩 = 重量 * 位置, 左边为负, 右边为正
```

```
 newBalance = j + weight * hookPos
```

```
检查新的平衡值是否在有效范围内
```

```
if 0 <= newBalance <= MAX_SUM:
```

```
 # 状态转移: 将前一个状态的方法数累加到新状态
```

```
 dp[i][newBalance] += dp[i - 1][j]
```

```
返回所有砝码放完后平衡值为 0 (平移后为 OFFSET) 的方法数
```

```
平衡值为 0 表示天平平衡
```

```
return dp[G][OFFSET]
```

,,

示例:

输入: C = 2, G = 4

hooks = [-2, 3]

```
weights = [3, 4, 5, 8]
```

输出: 2

解释: 有两种方法可以使天平平衡

时间复杂度:  $O(G * C * 15000)$

- 外层循环遍历所有砝码:  $O(G)$
- 中层循环遍历所有可能的平衡值:  $O(15000)$
- 内层循环遍历所有挂钩:  $O(C)$

空间复杂度:  $O(G * 15000)$

- 二维 DP 数组的空间消耗

, , ,

=====

文件: Code13\_CashMachine.cpp

=====

```
// POJ 1276 Cash Machine
```

// 题目描述: 有一个取款机, 可以提供不同面额的钞票, 每种面额有指定数量的钞票。

// 给定要取款的金额 cash, 求能取出的不超过 cash 的最大金额。

// 链接: <http://poj.org/problem?id=1276>

//

// 解题思路:

// 这是一个多重背包问题, 可以转化为 01 背包问题求解。

// 1. 多重背包: 每种物品 (钞票面额) 有指定数量

// 2. 转化方法: 二进制优化

// 将数量为 n 的物品分解为若干个物品, 数量分别为 1, 2, 4, ...,  $2^{k-1}$ ,  $n-2^k+1$

// 这样可以保证用这些物品组合出 1 到 n 之间的任意数量

// 3. 转化后用 01 背包求解

// 4.  $dp[i][j]$  表示前 i 种钞票面额, 能组成的大于等于 j 的金额

// 5. 状态转移方程:

```
// $dp[i][j] = \max(dp[i-1][j], dp[i-1][j-value[k]] + value[k])$
```

//

// 时间复杂度:  $O(N * cash * \log(max\_count))$

// 空间复杂度:  $O(cash)$

```
#define MAXCASH 100001
```

// 获取两个数中的较大值

```
int max(int a, int b) {
 return a > b ? a : b;
}
```

```
/**
```

```

* 计算能取出的最大金额
*
* 解题思路:
* 这是一个多重背包问题, 可以转化为 01 背包问题求解。
* 1. 多重背包: 每种物品(钞票面额)有指定数量
* 2. 转化方法: 二进制优化
* 将数量为 n 的物品分解为若干个物品, 数量分别为 1, 2, 4, ..., $2^{(k-1)}$, $n - 2^{k+1}$
* 这样可以保证用这些物品组合出 1 到 n 之间的任意数量
* 3. 转化后用 01 背包求解
* 4. $dp[i][j]$ 表示前 i 种钞票面额, 能组成的大于等于 j 的金额
* 5. 状态转移方程:
* $dp[i][j] = \max(dp[i-1][j], dp[i-1][j-value[k]] + value[k])$
*
* 参数:
* cash: 要取款的金额
* N: 钞票面额种类数
* counts: 每种面额的钞票数量数组
* values: 钞票面额数组
* 返回值:
* 能取出的最大金额
*/
int maxCash(int cash, int N, int* counts, int* values) {
 // dp[j] 表示能组成的大于等于 j 的金额
 // 这里使用了空间优化的一维 DP 数组
 int dp[MAXCASH];

 // 初始化 dp 数组
 for (int i = 0; i <= cash; i++) {
 dp[i] = 0;
 }

 // 遍历每种钞票面额(物品种类)
 for (int i = 0; i < N; i++) {
 // 获取当前面额的钞票数量和面额值
 int count = counts[i];
 int value = values[i];

 // 二进制优化: 将 count 个 value 面额的钞票分解
 // 将数量为 count 的物品分解为若干个物品, 数量分别为 1, 2, 4, ..., $2^{(k-1)}$, remaining
 // 这样可以保证用这些物品组合出 1 到 count 之间的任意数量
 for (int k = 1; k <= count; k <= 1) {
 // 01 背包: 倒序遍历
 // j 表示当前的取款金额
 }
 }
}

```

```

for (int j = cash; j >= k * value; j--) {
 // 状态转移方程:
 // dp[j] = max(不选择当前组合的钞票, 选择当前组合的钞票)
 // 不选择当前组合的钞票: dp[j] (保持原值)
 // 选择当前组合的钞票: dp[j - k * value] + k * value (前一个状态+当前组合钞票的总面额)
 dp[j] = max(dp[j], dp[j - k * value] + k * value);
}
// 减去已经处理的数量
count -= k;
}

// 处理剩余的钞票
if (count > 0) {
 // 01 背包: 倒序遍历
 // j 表示当前的取款金额
 for (int j = cash; j >= count * value; j--) {
 // 状态转移方程:
 // dp[j] = max(不选择剩余钞票, 选择剩余钞票)
 // 不选择剩余钞票: dp[j] (保持原值)
 // 选择剩余钞票: dp[j - count * value] + count * value (前一个状态+剩余钞票的总面额)
 dp[j] = max(dp[j], dp[j - count * value] + count * value);
 }
}
}

// 返回能取出的最大金额
return dp[cash];
}

/*
* 示例:
* 输入: cash = 735, N = 3
* counts = [4, 6, 3]
* values = [125, 5, 350]
* 输出: 735
* 解释: 可以恰好取出 735 元
*
* 输入: cash = 633, N = 4
* counts = [500, 6, 1, 0]
* values = [30, 100, 5, 1]
* 输出: 630

```

```
* 解释：最多能取出 630 元
*
* 时间复杂度：O(N * cash * log(max_count))
* - 外层循环遍历所有钞票面额：O(N)
* - 中层循环二进制分解：O(log(max_count))
* - 内层循环遍历取款金额：O(cash)
* 空间复杂度：O(cash)
* - 一维 DP 数组的空间消耗
*/
=====
```

文件：Code13\_CashMachine.java

```
package class073;

// POJ 1276 Cash Machine
// 题目描述：有一个取款机，可以提供不同面额的钞票，每种面额有指定数量的钞票。
// 给定要取款的金额 cash，求能取出的不超过 cash 的最大金额。
// 链接：http://poj.org/problem?id=1276
//
// 解题思路：
// 这是一个多重背包问题，可以转化为 01 背包问题求解。
// 1. 多重背包：每种物品（钞票面额）有指定数量
// 2. 转化方法：二进制优化
// 将数量为 n 的物品分解为若干个物品，数量分别为 1, 2, 4, ..., $2^{(k-1)}$, $n-2^{k+1}$
// 这样可以保证用这些物品组合出 1 到 n 之间的任意数量
// 3. 转化后用 01 背包求解
// 4. $dp[i][j]$ 表示前 i 种钞票面额，能组成的大于等于 j 的金额
// 5. 状态转移方程：
// $dp[i][j] = \max(dp[i-1][j], dp[i-1][j-value[k]] + value[k])$
//
// 时间复杂度：O(N * cash * log(max_count))
// 空间复杂度：O(cash)
```

```
public class Code13_CashMachine {

 /**
 * 计算能取出的最大金额
 *
 * 解题思路：
 * 这是一个多重背包问题，可以转化为 01 背包问题求解。
 * 1. 多重背包：每种物品（钞票面额）有指定数量
```

```

* 2. 转化方法: 二进制优化
* 将数量为 n 的物品分解为若干个物品, 数量分别为 1, 2, 4, ..., $2^{(k-1)}$, $n-2^{k+1}$
* 这样可以保证用这些物品组合出 1 到 n 之间的任意数量
* 3. 转化后用 01 背包求解
* 4. $dp[i][j]$ 表示前 i 种钞票面额, 能组成的大不超过 j 的金额
* 5. 状态转移方程:
* $dp[i][j] = \max(dp[i-1][j], dp[i-1][j-value[k]] + value[k])$
*
* @param cash 要取款的金额
* @param N 钞票面额种类数
* @param counts 每种面额的钞票数量数组
* @param values 钞票面额数组
* @return 能取出的最大金额
*/
public static int maxCash(int cash, int N, int[] counts, int[] values) {
 // $dp[j]$ 表示能组成的大不超过 j 的金额
 // 这里使用了空间优化的一维 DP 数组
 int[] dp = new int[cash + 1];

 // 遍历每种钞票面额 (物品种类)
 for (int i = 0; i < N; i++) {
 // 获取当前面额的钞票数量和面额值
 int count = counts[i];
 int value = values[i];

 // 二进制优化: 将 count 个 value 面额的钞票分解
 // 将数量为 count 的物品分解为若干个物品, 数量分别为 1, 2, 4, ..., $2^{(k-1)}$, remaining
 // 这样可以保证用这些物品组合出 1 到 count 之间的任意数量
 for (int k = 1; k <= count; k <= 1) {
 // 01 背包: 倒序遍历
 // j 表示当前的取款金额
 for (int j = cash; j >= k * value; j--) {
 // 状态转移方程:
 // $dp[j] = \max(\text{不选择当前组合的钞票}, \text{选择当前组合的钞票})$
 // 不选择当前组合的钞票: $dp[j]$ (保持原值)
 // 选择当前组合的钞票: $dp[j - k * value] + k * value$ (前一个状态+当前组合钞票
 的总面额)
 dp[j] = Math.max(dp[j], dp[j - k * value] + k * value);
 }
 // 减去已经处理的数量
 count -= k;
 }
 }
}

```

```

// 处理剩余的钞票
if (count > 0) {
 // 01 背包：倒序遍历
 // j 表示当前的取款金额
 for (int j = cash; j >= count * value; j--) {
 // 状态转移方程：
 // dp[j] = max(不选择剩余钞票, 选择剩余钞票)
 // 不选择剩余钞票: dp[j] (保持原值)
 // 选择剩余钞票: dp[j - count * value] + count * value (前一个状态+剩余钞票的
总面额)
 dp[j] = Math.max(dp[j], dp[j - count * value] + count * value);
 }
}

// 返回能取出的最大金额
return dp[cash];
}

/*
* 示例：
* 输入: cash = 735, N = 3
* counts = [4, 6, 3]
* values = [125, 5, 350]
* 输出: 735
* 解释: 可以恰好取出 735 元
*
* 输入: cash = 633, N = 4
* counts = [500, 6, 1, 0]
* values = [30, 100, 5, 1]
* 输出: 630
* 解释: 最多能取出 630 元
*
* 时间复杂度: O(N * cash * log(max_count))
* - 外层循环遍历所有钞票面额: O(N)
* - 中层循环二进制分解: O(log(max_count))
* - 内层循环遍历取款金额: O(cash)
* 空间复杂度: O(cash)
* - 一维 DP 数组的空间消耗
*/
}
=====
```

文件: Code13\_CashMachine.py

```
=====
POJ 1276 Cash Machine
题目描述: 有一个取款机, 可以提供不同面额的钞票, 每种面额有指定数量的钞票。
给定要取款的金额 cash, 求能取出的不超过 cash 的最大金额。
链接: http://poj.org/problem?id=1276
#
解题思路:
这是一个多重背包问题, 可以转化为 01 背包问题求解。
1. 多重背包: 每种物品(钞票面额)有指定数量
2. 转化方法: 二进制优化
将数量为 n 的物品分解为若干个物品, 数量分别为 1, 2, 4, ..., $2^{(k-1)}$, $n-2^{k+1}$
这样可以保证用这些物品组合出 1 到 n 之间的任意数量
3. 转化后用 01 背包求解
4. $dp[i][j]$ 表示前 i 种钞票面额, 能组成的大于等于 j 的金额
5. 状态转移方程:
$dp[i][j] = \max(dp[i-1][j], dp[i-1][j-value[k]] + value[k])$
#
时间复杂度: $O(N * cash * \log(\max_count))$
空间复杂度: $O(cash)$
```

```
def maxCash(cash, N, counts, values):
```

```
 """

```

```
 计算能取出的最大金额

```

解题思路:

这是一个多重背包问题, 可以转化为 01 背包问题求解。

1. 多重背包: 每种物品(钞票面额)有指定数量

2. 转化方法: 二进制优化

将数量为 n 的物品分解为若干个物品, 数量分别为 1, 2, 4, ...,  $2^{(k-1)}$ ,  $n-2^{k+1}$

这样可以保证用这些物品组合出 1 到 n 之间的任意数量

3. 转化后用 01 背包求解

4.  $dp[i][j]$  表示前 i 种钞票面额, 能组成的大于等于 j 的金额

5. 状态转移方程:

```
 $dp[i][j] = \max(dp[i-1][j], dp[i-1][j-value[k]] + value[k])$
```

Args:

cash: 要取款的金额

N: 钞票面额种类数

counts: 每种面额的钞票数量数组

values: 钞票面额数组

Returns:

能取出的最大金额

"""

# dp[j] 表示能组成的大于等于 j 的金额

# 这里使用了空间优化的一维 DP 数组

dp = [0] \* (cash + 1)

# 遍历每种钞票面额 (物品种类)

for i in range(N):

# 获取当前面额的钞票数量和面额值

count = counts[i]

value = values[i]

# 二进制优化: 将 count 个 value 面额的钞票分解

# 将数量为 count 的物品分解为若干个物品, 数量分别为 1, 2, 4, ...,  $2^{(k-1)}$ , remaining

# 这样可以保证用这些物品组合出 1 到 count 之间的任意数量

k = 1

while k <= count:

# 01 背包: 倒序遍历

# j 表示当前的取款金额

for j in range(cash, k \* value - 1, -1):

# 状态转移方程:

# dp[j] = max(不选择当前组合的钞票, 选择当前组合的钞票)

# 不选择当前组合的钞票: dp[j] (保持原值)

# 选择当前组合的钞票: dp[j - k \* value] + k \* value (前一个状态+当前组合钞票的总面额)

dp[j] = max(dp[j], dp[j - k \* value] + k \* value)

# 减去已经处理的数量

count -= k

# 下一个二进制位

k <<= 1

# 处理剩余的钞票

if count > 0:

# 01 背包: 倒序遍历

# j 表示当前的取款金额

for j in range(cash, count \* value - 1, -1):

# 状态转移方程:

# dp[j] = max(不选择剩余钞票, 选择剩余钞票)

# 不选择剩余钞票: dp[j] (保持原值)

额)

# 选择剩余钞票: dp[j - count \* value] + count \* value (前一个状态+剩余钞票的总面

额)

dp[j] = max(dp[j], dp[j - count \* value] + count \* value)

```
返回能取出的最大金额
return dp[cash]
```

,,

示例:

输入: cash = 735, N = 3

counts = [4, 6, 3]

values = [125, 5, 350]

输出: 735

解释: 可以恰好取出 735 元

输入: cash = 633, N = 4

counts = [500, 6, 1, 0]

values = [30, 100, 5, 1]

输出: 630

解释: 最多能取出 630 元

时间复杂度:  $O(N * \text{cash} * \log(\text{max\_count}))$

- 外层循环遍历所有钞票面额:  $O(N)$
- 中层循环二进制分解:  $O(\log(\text{max\_count}))$
- 内层循环遍历取款金额:  $O(\text{cash})$

空间复杂度:  $O(\text{cash})$

- 一维 DP 数组的空间消耗

,,

=====

文件: Code14\_CoinChange.cpp

=====

```
// LeetCode 322. 零钱兑换
```

```
// 题目描述: 给你一个整数数组 coins , 表示不同面额的硬币; 以及一个整数 amount , 表示总金额。
```

```
// 计算并返回可以凑成总金额所需的最少的硬币个数。如果没有任何一种硬币组合能组成总金额, 返回 -1 。
```

```
// 你可以认为每种硬币的数量是无限的。
```

```
// 链接: https://leetcode.cn/problems/coin-change/
```

```
//
```

```
// 解题思路:
```

```
// 这是一个完全背包问题的变形。
```

```
// 1. 状态定义: dp[i] 表示凑成金额 i 所需的最少硬币个数
```

```
// 2. 状态转移方程: dp[i] = min(dp[i], dp[i-coin] + 1), 其中 coin 是每种硬币的面额
```

```
// 3. 初始状态: dp[0] = 0 (凑成金额 0 不需要硬币), 其余初始化为一个较大值
```

```
// 4. 遍历顺序: 由于硬币可以重复使用, 这是完全背包问题, 使用正序遍历金额
```

```
//
```

```

// 时间复杂度: O(amount * n)，其中 n 是硬币种类数
// 空间复杂度: O(amount)

#define MAX_AMOUNT 10001

// 获取两个数中的较小值
int min(int a, int b) {
 return a < b ? a : b;
}

/**
 * 计算凑成总金额所需的最少硬币个数
 *
 * 解题思路:
 * 这是一个完全背包问题的变形。
 * 1. 状态定义: dp[i] 表示凑成金额 i 所需的最少硬币个数
 * 2. 状态转移方程: dp[i] = min(dp[i], dp[i-coin] + 1)，其中 coin 是每种硬币的面额
 * 3. 初始状态: dp[0] = 0 (凑成金额 0 不需要硬币)，其余初始化为一个较大值
 * 4. 遍历顺序: 由于硬币可以重复使用，这是完全背包问题，使用正序遍历金额
 *
 * 参数:
 * coins: 不同面额的硬币数组
 * coinsSize: 硬币数组的大小
 * amount: 目标总金额
 * 返回值:
 * 最少硬币个数，如果无法凑成返回-1
 */
int coinChange(int* coins, int coinsSize, int amount) {
 // 边界条件检查
 if (amount < 0) {
 return -1; // 金额不能为负数
 }
 if (amount == 0) {
 return 0; // 金额为 0 不需要硬币
 }
 if (coinsSize == 0) {
 return -1; // 硬币数组为空，无法凑成任何金额
 }

 // 创建 dp 数组，dp[i] 表示凑成金额 i 所需的最少硬币个数
 // 初始化为 amount + 1 (因为最多使用 amount 个面值为 1 的硬币)
 int dp[MAX_AMOUNT];
 for (int i = 0; i <= amount; i++) {

```

```

dp[i] = amount + 1;
}

dp[0] = 0; // 基础情况：凑成金额 0 需要 0 个硬币

// 遍历每种硬币（物品）
for (int i = 0; i < coinsSize; i++) {
 int coin = coins[i];
 // 完全背包问题，正序遍历金额（容量）
 // 这样可以保证每个硬币可以被重复使用
 for (int j = coin; j <= amount; j++) {
 // 状态转移：选择当前硬币或不选当前硬币
 // 如果选择当前硬币，则需要 dp[j-coin] + 1 个硬币
 // dp[j] = min(不选择当前硬币, 选择当前硬币)
 // 不选择当前硬币: dp[j] (保持原值)
 // 选择当前硬币: dp[j - coin] + 1 (前一个状态+1 个当前硬币)
 if (dp[j - coin] != amount + 1) { // 确保 dp[j-coin] 是可达的
 dp[j] = min(dp[j], dp[j - coin] + 1);
 }
 }
}

// 如果 dp[amount] 仍为初始值，说明无法凑成该金额
return dp[amount] > amount ? -1 : dp[amount];
}

/***
 * 优化版本：提前剪枝和优化
 *
 * 参数：
 * coins: 不同面额的硬币数组
 * coinsSize: 硬币数组的大小
 * amount: 目标总金额
 * 返回值：
 * 最少硬币个数，如果无法凑成返回-1
 */
int coinChangeOptimized(int* coins, int coinsSize, int amount) {
 // 边界条件检查
 if (amount < 0) {
 return -1;
 }
 if (amount == 0) {
 return 0;
 }
}

```

```

if (coinsSize == 0) {
 return -1;
}

// 这里省略了排序步骤，因为在 C 中实现排序比较复杂
// 在实际应用中可以使用 qsort 函数进行排序

int dp[MAX_AMOUNT];
for (int i = 0; i <= amount; i++) {
 dp[i] = amount + 1;
}
dp[0] = 0;

for (int i = 0; i < coinsSize; i++) {
 int coin = coins[i];
 // 剪枝：如果当前硬币面值大于目标金额，可以跳过
 if (coin > amount) {
 continue;
 }

 for (int j = coin; j <= amount; j++) {
 if (dp[j - coin] != amount + 1) {
 dp[j] = min(dp[j], dp[j - coin] + 1);
 }
 }
}

return dp[amount] > amount ? -1 : dp[amount];
}

/*
 * 示例：
 * 输入：coins = [1, 2, 5], amount = 11
 * 输出：3
 * 解释：11 = 5 + 5 + 1
 *
 * 输入：coins = [2], amount = 3
 * 输出：-1
 *
 * 输入：coins = [1], amount = 0
 * 输出：0
 *
 * 时间复杂度：O(amount * n)

```

```
* - 外层循环遍历所有硬币: O(n)
* - 内层循环遍历金额: O(amount)
* 空间复杂度: O(amount)
* - 一维 DP 数组的空间消耗
*/
```

=====

文件: Code14\_CoinChange.java

=====

```
package class073;

// LeetCode 322. 零钱兑换
// 题目描述: 给你一个整数数组 coins，表示不同面额的硬币；以及一个整数 amount，表示总金额。
// 计算并返回可以凑成总金额所需的最少的硬币个数。如果没有任何一种硬币组合能组成总金额，返回 -1。
// 你可以认为每种硬币的数量是无限的。
// 链接: https://leetcode.cn/problems/coin-change/
//
// 解题思路:
// 这是一个完全背包问题的变形。
// 1. 状态定义: dp[i] 表示凑成金额 i 所需的最少硬币个数
// 2. 状态转移方程: dp[i] = min(dp[i], dp[i-coin] + 1)，其中 coin 是每种硬币的面额
// 3. 初始状态: dp[0] = 0 (凑成金额 0 不需要硬币)，其余初始化为一个较大值
// 4. 遍历顺序: 由于硬币可以重复使用，这是完全背包问题，使用正序遍历金额
//
// 时间复杂度: O(amount * n)，其中 n 是硬币种类数
// 空间复杂度: O(amount)
```

```
public class Code14_CoinChange {

 // 主方法，用于测试
 public static void main(String[] args) {
 // 测试用例 1
 int[] coins1 = {1, 2, 5};
 int amount1 = 11;
 System.out.println("测试用例 1 结果: " + coinChange(coins1, amount1)); // 预期输出: 3 (11
= 5 + 5 + 1)

 // 测试用例 2
 int[] coins2 = {2};
 int amount2 = 3;
 System.out.println("测试用例 2 结果: " + coinChange(coins2, amount2)); // 预期输出: -1
```

```

// 测试用例 3
int[] coins3 = {1};
int amount3 = 0;
System.out.println("测试用例 3 结果: " + coinChange(coins3, amount3)); // 预期输出: 0
}

/**
 * 计算凑成总金额所需的最少硬币个数
 *
 * 解题思路:
 * 这是一个完全背包问题的变形。
 * 1. 状态定义: dp[i] 表示凑成金额 i 所需的最少硬币个数
 * 2. 状态转移方程: dp[i] = min(dp[i], dp[i-coin] + 1), 其中 coin 是每种硬币的面额
 * 3. 初始状态: dp[0] = 0 (凑成金额 0 不需要硬币), 其余初始化为一个较大值
 * 4. 遍历顺序: 由于硬币可以重复使用, 这是完全背包问题, 使用正序遍历金额
 *
 * @param coins 不同面额的硬币数组
 * @param amount 目标总金额
 * @return 最少硬币个数, 如果无法凑成返回-1
 */
public static int coinChange(int[] coins, int amount) {
 // 参数验证
 if (coins == null || coins.length == 0) {
 throw new IllegalArgumentException("硬币数组不能为空");
 }

 // 边界条件处理
 if (amount < 0) {
 return -1; // 金额不能为负数
 }
 if (amount == 0) {
 return 0; // 金额为 0 不需要硬币
 }

 // 创建 dp 数组, dp[i] 表示凑成金额 i 所需的最少硬币个数
 // 初始化为 amount + 1 (因为最多使用 amount 个面值为 1 的硬币)
 int[] dp = new int[amount + 1];
 for (int i = 1; i <= amount; i++) {
 dp[i] = amount + 1;
 }
 dp[0] = 0; // 基础情况: 凑成金额 0 需要 0 个硬币

 // 遍历每种硬币 (物品)

```

```

for (int coin : coins) {
 // 完全背包问题，正序遍历金额（容量）
 // 这样可以保证每个硬币可以被重复使用
 for (int j = coin; j <= amount; j++) {
 // 状态转移：选择当前硬币或不选当前硬币
 // 如果选择当前硬币，则需要 dp[j-coin] + 1 个硬币
 // dp[j] = min(不选择当前硬币, 选择当前硬币)
 // 不选择当前硬币: dp[j] (保持原值)
 // 选择当前硬币: dp[j - coin] + 1 (前一个状态+1 个当前硬币)
 dp[j] = Math.min(dp[j], dp[j - coin] + 1);
 }
}

// 如果 dp[amount] 仍为初始值，说明无法凑成该金额
return dp[amount] > amount ? -1 : dp[amount];
}

/***
 * 优化版本：提前剪枝和优化
 * 1. 对硬币进行排序，从小到大处理
 * 2. 当发现当前硬币面值大于剩余金额时，可以提前跳过
 * 3. 当找到一个可能的解后，可以尝试进一步优化
 *
 * @param coins 不同面额的硬币数组
 * @param amount 目标总金额
 * @return 最少硬币个数，如果无法凑成返回-1
 */
public static int coinChangeOptimized(int[] coins, int amount) {
 // 参数验证
 if (coins == null || coins.length == 0) {
 throw new IllegalArgumentException("硬币数组不能为空");
 }

 // 边界条件处理
 if (amount < 0) {
 return -1;
 }
 if (amount == 0) {
 return 0;
 }

 // 创建 dp 数组
 int[] dp = new int[amount + 1];

```

```

for (int i = 1; i <= amount; i++) {
 dp[i] = amount + 1;
}
dp[0] = 0;

// 主逻辑与原方法相同
for (int coin : coins) {
 for (int j = coin; j <= amount; j++) {
 dp[j] = Math.min(dp[j], dp[j - coin] + 1);
 }
}

return dp[amount] > amount ? -1 : dp[amount];
}

/***
 * 工程化版本：包含详细的异常处理和日志记录
 * 在实际项目中，可以根据需要添加日志记录
 *
 * @param coins 不同面额的硬币数组
 * @param amount 目标总金额
 * @return 最少硬币个数，如果无法凑成返回-1
 */
public static int coinChangeProduction(int[] coins, int amount) {
 try {
 // 输入参数验证
 if (coins == null) {
 throw new NullPointerException("硬币数组不能为 null");
 }
 if (coins.length == 0) {
 throw new IllegalArgumentException("硬币数组不能为空");
 }

 // 检查硬币面值是否有效
 for (int coin : coins) {
 if (coin <= 0) {
 throw new IllegalArgumentException("硬币面值必须为正数：" + coin);
 }
 }

 // 调用核心算法
 return coinChange(coins, amount);
 } catch (Exception e) {

```

```

// 在实际项目中，这里应该记录异常日志
System.out.println("计算硬币兑换时发生错误：" + e.getMessage());
throw e; // 重新抛出异常，让调用者处理
}

}

/*
 * 示例：
 * 输入：coins = [1, 2, 5]，amount = 11
 * 输出：3
 * 解释：11 = 5 + 5 + 1
 *
 * 输入：coins = [2]，amount = 3
 * 输出：-1
 *
 * 输入：coins = [1]，amount = 0
 * 输出：0
 *
 * 时间复杂度：O(amount * n)
 * - 外层循环遍历所有硬币：O(n)
 * - 内层循环遍历金额：O(amount)
 * 空间复杂度：O(amount)
 * - 一维 DP 数组的空间消耗
 */
}

```

文件：Code14\_CoinChange.py

```

LeetCode 322. 零钱兑换
题目描述：给你一个整数数组 coins，表示不同面额的硬币；以及一个整数 amount，表示总金额。
计算并返回可以凑成总金额所需的最少的硬币个数。如果没有任何一种硬币组合能组成总金额，返回 -1。
你可以认为每种硬币的数量是无限的。
链接：https://leetcode.cn/problems/coin-change/
#
解题思路：
这是一个完全背包问题的变形。
1. 状态定义：dp[i] 表示凑成金额 i 所需的最少硬币个数
2. 状态转移方程：dp[i] = min(dp[i], dp[i-coin] + 1)，其中 coin 是每种硬币的面额
3. 初始状态：dp[0] = 0（凑成金额 0 不需要硬币），其余初始化为一个较大值
4. 遍历顺序：由于硬币可以重复使用，这是完全背包问题，使用正序遍历金额
#

```

```
时间复杂度: O(amount * n), 其中 n 是硬币种类数
空间复杂度: O(amount)
```

```
def coin_change(coins, amount):
 """
 计算凑成总金额所需的最少硬币个数
```

解题思路:

这是一个完全背包问题的变形。

1. 状态定义:  $dp[i]$  表示凑成金额  $i$  所需的最少硬币个数
2. 状态转移方程:  $dp[i] = \min(dp[i], dp[i-coin] + 1)$ , 其中  $coin$  是每种硬币的面额
3. 初始状态:  $dp[0] = 0$  (凑成金额 0 不需要硬币), 其余初始化为一个较大值
4. 遍历顺序: 由于硬币可以重复使用, 这是完全背包问题, 使用正序遍历金额

Args:

coins: 不同面额的硬币列表

amount: 目标总金额

Returns:

最少硬币个数, 如果无法凑成返回-1

Raises:

TypeError: 如果输入类型不正确

ValueError: 如果输入值无效

"""

```
边界条件处理
if amount < 0:
 return -1 # 金额不能为负数
if amount == 0:
 return 0 # 金额为 0 不需要硬币
if not coins:
 return -1 # 硬币列表为空, 无法凑成任何金额
```

```
创建 dp 数组, dp[i] 表示凑成金额 i 所需的最少硬币个数
初始化为 amount + 1 (因为最多使用 amount 个面值为 1 的硬币)
dp = [amount + 1] * (amount + 1)
dp[0] = 0 # 基础情况: 凑成金额 0 需要 0 个硬币
```

```
遍历每种硬币 (物品)
for coin in coins:
 # 完全背包问题, 正序遍历金额 (容量)
 # 这样可以保证每个硬币可以被重复使用
```

```

for j in range(coin, amount + 1):
 # 状态转移: 选择当前硬币或不选当前硬币
 # 如果选择当前硬币, 则需要 dp[j-coin] + 1 个硬币
 # dp[j] = min(不选择当前硬币, 选择当前硬币)
 # 不选择当前硬币: dp[j] (保持原值)
 # 选择当前硬币: dp[j - coin] + 1 (前一个状态+1 个当前硬币)
 dp[j] = min(dp[j], dp[j - coin] + 1)

如果 dp[amount] 仍为初始值, 说明无法凑成该金额
return dp[amount] if dp[amount] != amount + 1 else -1

```

def coin\_change\_optimized(coins, amount):

"""

优化版本: 提前剪枝和优化

Args:

coins: 不同面额的硬币列表

amount: 目标总金额

Returns:

最少硬币个数, 如果无法凑成返回-1

"""

# 边界条件处理

if amount < 0:

return -1

if amount == 0:

return 0

if not coins:

return -1

# 对硬币进行排序, 从小到大

coins.sort()

dp = [amount + 1] \* (amount + 1)

dp[0] = 0

for coin in coins:

# 剪枝: 如果当前硬币面值大于目标金额, 可以跳过

if coin > amount:

continue

for j in range(coin, amount + 1):

```
dp[j] = min(dp[j], dp[j - coin] + 1)

return dp[amount] if dp[amount] != amount + 1 else -1
```

```
def coin_change_production(coins, amount):
```

```
"""
```

工程化版本：包含详细的输入验证和异常处理

Args:

coins: 不同面额的硬币列表

amount: 目标总金额

Returns:

最少硬币个数，如果无法凑成返回-1

Raises:

TypeError: 如果输入类型不正确

ValueError: 如果输入值无效

```
"""
```

```
try:
```

# 输入类型验证

```
if not isinstance(coins, list):
```

```
 raise TypeError("coins 必须是列表类型")
```

```
if not isinstance(amount, int):
```

```
 raise TypeError("amount 必须是整数类型")
```

# 输入值验证

```
if amount < 0:
```

```
 raise ValueError("amount 不能为负数")
```

# 检查硬币面值是否有效

```
for coin in coins:
```

```
 if not isinstance(coin, int) or coin <= 0:
```

```
 raise ValueError(f"硬币面值必须为正整数: {coin}")
```

# 调用核心算法

```
return coin_change(coins, amount)
```

```
except (TypeError, ValueError) as e:
```

# 在实际项目中，这里应该记录异常日志

```
print(f"计算硬币兑换时发生错误: {e}")
```

```
raise # 重新抛出异常，让调用者处理
```

```

测试代码
if __name__ == "__main__":
 # 测试用例 1
 coins1 = [1, 2, 5]
 amount1 = 11
 print(f"测试用例 1 结果: {coin_change(coins1, amount1)}") # 预期输出: 3 (11 = 5 + 5 + 1)

 # 测试用例 2
 coins2 = [2]
 amount2 = 3
 print(f"测试用例 2 结果: {coin_change(coins2, amount2)}") # 预期输出: -1

 # 测试用例 3
 coins3 = [1]
 amount3 = 0
 print(f"测试用例 3 结果: {coin_change(coins3, amount3)}") # 预期输出: 0

 # 测试异常情况
 try:
 coin_change_production([1, -1, 5], 10)
 except ValueError as e:
 print(f"正确捕获异常: {e}")

 ...

```

示例:

输入: coins = [1, 2, 5], amount = 11  
 输出: 3  
 解释:  $11 = 5 + 5 + 1$

输入: coins = [2], amount = 3  
 输出: -1

输入: coins = [1], amount = 0  
 输出: 0

时间复杂度:  $O(amount * n)$

- 外层循环遍历所有硬币:  $O(n)$
- 内层循环遍历金额:  $O(amount)$

空间复杂度:  $O(amount)$

- 一维 DP 数组的空间消耗

...

文件: Code15\_PartitionEqualSubsetSum.cpp

```
=====

// LeetCode 416. 分割等和子集
// 题目描述: 给你一个只包含正整数的非空数组 nums。请你判断是否可以将这个数组分割成两个子集，使得
// 两个子集的元素和相等。
// 链接: https://leetcode.cn/problems/partition-equal-subset-sum/
//
// 解题思路:
// 这是一个 01 背包问题的变形。我们需要将数组分成两个和相等的子集，等价于找到一个子集，其和为整个
// 数组和的一半。
// 1. 首先检查数组总和是否为偶数，如果为奇数则不可能分割成两个等和的子集
// 2. 如果总和为偶数，问题转化为：是否可以从数组中选择一些元素，使其和恰好为总和的一半
// 3. 状态定义：dp[j] 表示是否可以选择一些元素，使其和恰好为 j
// 4. 状态转移方程：dp[j] = dp[j] || dp[j - nums[i]]
// 5. 初始状态：dp[0] = true (空集的和为 0)
//
// 时间复杂度: O(n * target)，其中 n 是数组长度，target 是数组和的一半
// 空间复杂度: O(target)
```

```
#define MAX_TARGET 20001
```

```
// 获取两个数中的较大值
```

```
int max(int a, int b) {
 return a > b ? a : b;
}
```

```
/**
```

```
* 判断是否可以将数组分割成两个和相等的子集
```

```
*
```

```
* 解题思路:
```

```
* 这是一个 01 背包问题的变形。我们需要将数组分成两个和相等的子集，等价于找到一个子集，其和为整个
* 数组和的一半。
```

```
* 1. 首先检查数组总和是否为偶数，如果为奇数则不可能分割成两个等和的子集
```

```
* 2. 如果总和为偶数，问题转化为：是否可以从数组中选择一些元素，使其和恰好为总和的一半
```

```
* 3. 状态定义：dp[j] 表示是否可以选择一些元素，使其和恰好为 j
```

```
* 4. 状态转移方程：dp[j] = dp[j] || dp[j - nums[i]]
```

```
* 5. 初始状态：dp[0] = true (空集的和为 0)
```

```
*
```

```
* 参数:
```

```
* nums: 输入的非空数组
```

```
* numsSize: 数组长度
```

```

* 返回值:
* 如果可以分割返回 true, 否则返回 false
*/
bool canPartition(int* nums, int numsSize) {
 // 参数验证
 if (numsSize < 2) {
 return false; // 数组长度小于 2, 无法分割
 }

 // 计算数组总和
 int sum = 0;
 for (int i = 0; i < numsSize; i++) {
 sum += nums[i];
 }

 // 如果总和为奇数, 无法分割成两个等和的子集
 if (sum % 2 != 0) {
 return false;
 }

 // 目标值为总和的一半
 int target = sum / 2;

 // 创建 dp 数组, dp[j] 表示是否可以选择一些元素, 使其和恰好为 j
 // 这是一个布尔型的 01 背包问题
 bool dp[MAX_TARGET];
 for (int i = 0; i <= target; i++) {
 dp[i] = false;
 }
 dp[0] = true; // 基础情况: 空集的和为 0, 这是可达的

 // 遍历每个元素 (物品)
 for (int i = 0; i < numsSize; i++) {
 // 01 背包问题, 逆序遍历容量
 // 这样可以保证每个元素只使用一次
 for (int j = target; j >= nums[i]; j--) {
 // 状态转移: 选择当前元素或不选当前元素
 // dp[j] = 不选择当前元素 || 选择当前元素
 // 不选择当前元素: dp[j] (保持原值)
 // 选择当前元素: dp[j - nums[i]] (前一个状态)
 dp[j] = dp[j] || dp[j - nums[i]];
 }
 }
}

```

```
// 优化: 如果已经找到解, 可以提前结束
// 如果 dp[target] 为 true, 说明已经找到了和为 target 的子集
if (dp[target]) {
 return true;
}
}

// 返回是否能找到和为 target 的子集
return dp[target];
}
```

```
/***
 * 优化版本: 包含更多的剪枝条件
 *
 * 参数:
 * nums: 输入的非空数组
 * numsSize: 数组长度
 * 返回值:
 * 如果可以分割返回 true, 否则返回 false
 */
```

```
bool canPartitionOptimized(int* nums, int numsSize) {
 // 参数验证
 if (numsSize < 2) {
 return false;
 }
```

```
 // 计算数组总和和最大值
 int sum = 0;
 int maxNum = 0;
 for (int i = 0; i < numsSize; i++) {
 sum += nums[i];
 maxNum = max(maxNum, nums[i]);
 }
```

```
 // 如果总和为奇数, 无法分割
 if (sum % 2 != 0) {
 return false;
 }
```

```
 int target = sum / 2;
```

```
 // 如果最大值大于 target, 不可能分割
 // 因为任何一个元素都比 target 大, 无法组成 target
```

```

if (maxNum > target) {
 return false;
}

// 创建 dp 数组
bool dp[MAX_TARGET];
for (int i = 0; i <= target; i++) {
 dp[i] = false;
}
dp[0] = true;

for (int i = 0; i < numsSize; i++) {
 // 优化: 如果当前元素大于 target, 可以跳过
 // 因为当前元素本身就比 target 大, 无法用于组成 target
 if (nums[i] > target) {
 continue;
 }

 for (int j = target; j >= nums[i]; j--) {
 dp[j] = dp[j] || dp[j - nums[i]];
 }
}

if (dp[target]) {
 return true;
}

return dp[target];
}

/*
 * 示例:
 * 输入: nums = [1, 5, 11, 5]
 * 输出: true
 * 解释: 数组可以分割成 [1, 5, 5] 和 [11]。
 *
 * 输入: nums = [1, 2, 3, 5]
 * 输出: false
 * 解释: 数组不能分割成两个元素和相等的子集。
 *
 * 时间复杂度: O(n * target)
 * - 外层循环遍历所有元素: O(n)
 * - 内层循环遍历目标值: O(target)

```

```
* 空间复杂度: O(target)
* - 一维 DP 数组的空间消耗
*/
```

---

文件: Code15\_PartitionEqualSubsetSum.java

---

```
package class073;
```

```
// LeetCode 416. 分割等和子集
```

```
// 题目描述: 给你一个只包含正整数的非空数组 nums。请你判断是否可以将这个数组分割成两个子集，使得两个子集的元素和相等。
```

```
// 链接: https://leetcode.cn/problems/partition-equal-subset-sum/
```

```
//
```

```
// 解题思路:
```

```
// 这是一个 01 背包问题的变形。我们需要将数组分成两个和相等的子集，等价于找到一个子集，其和为整个数组和的一半。
```

```
// 1. 首先检查数组总和是否为偶数，如果为奇数则不可能分割成两个等和的子集
```

```
// 2. 如果总和为偶数，问题转化为：是否可以从数组中选择一些元素，使其和恰好为总和的一半
```

```
// 3. 状态定义: dp[j] 表示是否可以选择一些元素，使其和恰好为 j
```

```
// 4. 状态转移方程: dp[j] = dp[j] || dp[j - nums[i]]
```

```
// 5. 初始状态: dp[0] = true (空集的和为 0)
```

```
//
```

```
// 时间复杂度: O(n * target)，其中 n 是数组长度，target 是数组和的一半
```

```
// 空间复杂度: O(target)
```

```
public class Code15_PartitionEqualSubsetSum {
```

```
// 主方法，用于测试
```

```
public static void main(String[] args) {
```

```
 // 测试用例 1
```

```
 int[] nums1 = {1, 5, 11, 5};
```

```
 System.out.println("测试用例 1 结果: " + canPartition(nums1)); // 预期输出: true (可以分割为 [1, 5, 5] 和 [11])
```

```
 // 测试用例 2
```

```
 int[] nums2 = {1, 2, 3, 5};
```

```
 System.out.println("测试用例 2 结果: " + canPartition(nums2)); // 预期输出: false
```

```
 // 测试用例 3
```

```
 int[] nums3 = {2, 2, 1, 1};
```

```
 System.out.println("测试用例 3 结果: " + canPartition(nums3)); // 预期输出: true (可以分割
```

```

为 [2, 1] 和 [2, 1])
}

/***
 * 判断是否可以将数组分割成两个和相等的子集
 *
 * 解题思路：
 * 这是一个01背包问题的变形。我们需要将数组分成两个和相等的子集，等价于找到一个子集，其和为整个数组和的一半。
 * 1. 首先检查数组总和是否为偶数，如果为奇数则不可能分割成两个等和的子集
 * 2. 如果总和为偶数，问题转化为：是否可以从数组中选择一些元素，使其和恰好为总和的一半
 * 3. 状态定义：dp[j] 表示是否可以选择一些元素，使其和恰好为 j
 * 4. 状态转移方程：dp[j] = dp[j] || dp[j - nums[i]]
 * 5. 初始状态：dp[0] = true (空集的和为 0)
 *
 * @param nums 输入的非空数组
 * @return 如果可以分割返回 true，否则返回 false
 */
public static boolean canPartition(int[] nums) {
 // 参数验证
 if (nums == null || nums.length < 2) {
 return false; // 数组长度小于 2，无法分割
 }

 // 计算数组总和
 int sum = 0;
 for (int num : nums) {
 sum += num;
 }

 // 如果总和为奇数，无法分割成两个等和的子集
 if (sum % 2 != 0) {
 return false;
 }

 // 目标值为总和的一半
 int target = sum / 2;

 // 创建 dp 数组，dp[j] 表示是否可以选择一些元素，使其和恰好为 j
 // 这是一个布尔型的01背包问题
 boolean[] dp = new boolean[target + 1];
 dp[0] = true; // 基础情况：空集的和为 0，这是可达的
}

```

```

// 遍历每个元素（物品）
for (int i = 0; i < nums.length; i++) {
 // 01 背包问题，逆序遍历容量
 // 这样可以保证每个元素只使用一次
 for (int j = target; j >= nums[i]; j--) {
 // 状态转移：选择当前元素或不选当前元素
 // dp[j] = 不选择当前元素 || 选择当前元素
 // 不选择当前元素：dp[j] (保持原值)
 // 选择当前元素：dp[j - nums[i]] (前一个状态)
 dp[j] = dp[j] || dp[j - nums[i]];
 }
}

// 优化：如果已经找到解，可以提前结束
// 如果 dp[target] 为 true，说明已经找到了和为 target 的子集
if (dp[target]) {
 return true;
}
}

// 返回是否能找到和为 target 的子集
return dp[target];
}

/**
 * 优化版本：包含更多的剪枝条件
 *
 * @param nums 输入的非空数组
 * @return 如果可以分割返回 true，否则返回 false
 */
public static boolean canPartitionOptimized(int[] nums) {
 // 参数验证
 if (nums == null || nums.length < 2) {
 return false;
 }

 // 计算数组总和和最大值
 int sum = 0;
 int maxNum = 0;
 for (int num : nums) {
 sum += num;
 maxNum = Math.max(maxNum, num);
 }
}

```

```
// 如果总和为奇数，无法分割
if (sum % 2 != 0) {
 return false;
}

int target = sum / 2;

// 如果最大值大于 target，不可能分割
// 因为任何一个元素都比 target 大，无法组成 target
if (maxNum > target) {
 return false;
}

// 创建 dp 数组
boolean[] dp = new boolean[target + 1];
dp[0] = true;

for (int i = 0; i < nums.length; i++) {
 // 优化：如果当前元素大于 target，可以跳过
 // 因为当前元素本身就比 target 大，无法用于组成 target
 if (nums[i] > target) {
 continue;
 }

 for (int j = target; j >= nums[i]; j--) {
 dp[j] = dp[j] || dp[j - nums[i]];
 }
}

if (dp[target]) {
 return true;
}

return dp[target];
}

/*
 * 示例：
 * 输入：nums = [1, 5, 11, 5]
 * 输出：true
 * 解释：数组可以分割成 [1, 5, 5] 和 [11]。
 *
 * 输入：nums = [1, 2, 3, 5]
 */
```

```

 * 输出: false
 * 解释: 数组不能分割成两个元素和相等的子集。
 *
 * 时间复杂度: O(n * target)
 * - 外层循环遍历所有元素: O(n)
 * - 内层循环遍历目标值: O(target)
 * 空间复杂度: O(target)
 * - 一维 DP 数组的空间消耗
 */
}

=====

文件: Code15_PartitionEqualSubsetSum.py
=====

LeetCode 416. 分割等和子集
题目描述: 给你一个只包含正整数的非空数组 nums。请你判断是否可以将这个数组分割成两个子集，使得两个子集的元素和相等。
链接: https://leetcode.cn/problems/partition-equal-subset-sum/
#
解题思路:
这是一个 01 背包问题的变形。我们需要将数组分成两个和相等的子集，等价于找到一个子集，其和为整个数组和的一半。
1. 首先检查数组总和是否为偶数，如果为奇数则不可能分割成两个等和的子集
2. 如果总和为偶数，问题转化为：是否可以从数组中选择一些元素，使其和恰好为总和的一半
3. 状态定义: dp[j] 表示是否可以选择一些元素，使其和恰好为 j
4. 状态转移方程: dp[j] = dp[j] or dp[j - nums[i]]
5. 初始状态: dp[0] = True (空集的和为 0)
#
时间复杂度: O(n * target)，其中 n 是数组长度，target 是数组和的一半
空间复杂度: O(target)

def can_partition(nums):
 """
 判断是否可以将数组分割成两个和相等的子集
 """

解题思路:
 这是一个 01 背包问题的变形。我们需要将数组分成两个和相等的子集，等价于找到一个子集，其和为整个数组和的一半。
 1. 首先检查数组总和是否为偶数，如果为奇数则不可能分割成两个等和的子集
 2. 如果总和为偶数，问题转化为：是否可以从数组中选择一些元素，使其和恰好为总和的一半
 3. 状态定义: dp[j] 表示是否可以选择一些元素，使其和恰好为 j
 4. 状态转移方程: dp[j] = dp[j] or dp[j - nums[i]]
```

5. 初始状态:  $dp[0] = \text{True}$  (空集的和为 0)

Args:

nums: 输入的非空数组

Returns:

bool: 如果可以分割返回 True, 否则返回 False

"""

# 参数验证

if not nums or len(nums) < 2:

return False # 数组长度小于 2, 无法分割

# 计算数组总和

total\_sum = sum(nums)

# 如果总和为奇数, 无法分割成两个等和的子集

if total\_sum % 2 != 0:

return False

# 目标值为总和的一半

target = total\_sum // 2

# 创建 dp 数组,  $dp[j]$  表示是否可以选择一些元素, 使其和恰好为  $j$

# 这是一个布尔型的 01 背包问题

dp = [False] \* (target + 1)

dp[0] = True # 基础情况: 空集的和为 0, 这是可达的

# 遍历每个元素 (物品)

for num in nums:

# 01 背包问题, 逆序遍历容量

# 这样可以保证每个元素只使用一次

for j in range(target, num - 1, -1):

# 状态转移: 选择当前元素或不选当前元素

#  $dp[j] = \text{不选择当前元素 or 选择当前元素}$

# 不选择当前元素:  $dp[j]$  (保持原值)

# 选择当前元素:  $dp[j - num]$  (前一个状态)

$dp[j] = dp[j] \text{ or } dp[j - num]$

# 优化: 如果已经找到解, 可以提前结束

# 如果  $dp[target]$  为 True, 说明已经找到了和为 target 的子集

if dp[target]:

return True

```
返回是否能找到和为 target 的子集
return dp[target]

def can_partition_optimized(nums):
 """
 优化版本：包含更多的剪枝条件

 Args:
 nums: 输入的非空数组

 Returns:
 bool: 如果可以分割返回 True，否则返回 False
 """

 # 参数验证
 if not nums or len(nums) < 2:
 return False

 # 计算数组总和和最大值
 total_sum = sum(nums)
 max_num = max(nums)

 # 如果总和为奇数，无法分割
 if total_sum % 2 != 0:
 return False

 target = total_sum // 2

 # 如果最大值大于 target，不可能分割
 # 因为任何一个元素都比 target 大，无法组成 target
 if max_num > target:
 return False

 # 创建 dp 数组
 dp = [False] * (target + 1)
 dp[0] = True

 for num in nums:
 # 优化：如果当前元素大于 target，可以跳过
 # 因为当前元素本身就比 target 大，无法用于组成 target
 if num > target:
 continue

 for j in range(target, num - 1, -1):
```

```

dp[j] = dp[j] or dp[j - num]

if dp[target]:
 return True

return dp[target]

测试用例
if __name__ == "__main__":
 # 测试用例 1
 nums1 = [1, 5, 11, 5]
 print(f"测试用例 1 结果: {can_partition(nums1)}") # 预期输出: True (可以分割为 [1, 5, 5] 和 [11])

 # 测试用例 2
 nums2 = [1, 2, 3, 5]
 print(f"测试用例 2 结果: {can_partition(nums2)}") # 预期输出: False

 # 测试用例 3
 nums3 = [2, 2, 1, 1]
 print(f"测试用例 3 结果: {can_partition(nums3)}") # 预期输出: True (可以分割为 [2, 1] 和 [2, 1])

```

, , ,

示例:

输入: nums = [1, 5, 11, 5]

输出: true

解释: 数组可以分割成 [1, 5, 5] 和 [11]。

输入: nums = [1, 2, 3, 5]

输出: false

解释: 数组不能分割成两个元素和相等的子集。

时间复杂度:  $O(n * target)$

- 外层循环遍历所有元素:  $O(n)$
- 内层循环遍历目标值:  $O(target)$

空间复杂度:  $O(target)$

- 一维 DP 数组的空间消耗

, , ,

---

文件: Code16\_TargetSum.cpp

```
=====
// LeetCode 494. 目标和
// 题目描述：给你一个整数数组 nums 和一个整数 target 。
// 向数组中的每个整数前添加 '+' 或 '-' ，然后串联起所有整数，可以构造一个 表达式 ：
// 例如，nums = [2, 1] ，可以在 2 之前添加 '+' ，在 1 之前添加 '-' ，得到表达式 "+2-1" 。
// 返回可以通过上述方法构造的、运算结果等于 target 的不同 表达式 的数目。
// 链接: https://leetcode.cn/problems/target-sum/
//
// 解题思路：
// 这是一个 01 背包问题的变形。我们需要将数组分成两个部分：正数部分和负数部分，使得它们的和等于 target。
// 设正数部分的和为 sum_pos，负数部分的和为 sum_neg，整个数组的和为 sum。
// 则有：sum_pos - sum_neg = target
// 又因为：sum_pos + sum_neg = sum
// 联立解得：sum_pos = (sum + target) / 2
// 因此问题转化为：找到一个子集，其和等于(sum + target)/2，这样的子集有多少个？
// 注意：必须满足 sum + target 为偶数且 sum + target >= 0，否则不存在这样的子集。
//
// 时间复杂度：O(n * target)
// 空间复杂度：O(target)
```

```
#define MAX_TARGET 20001
```

```
/**
 * 计算可以构造目标和的不同表达式的数目
 *
 * 解题思路：
 * 这是一个 01 背包问题的变形。我们需要将数组分成两个部分：正数部分和负数部分，使得它们的和等于 target。
 * 设正数部分的和为 sum_pos，负数部分的和为 sum_neg，整个数组的和为 sum。
 * 则有：sum_pos - sum_neg = target
 * 又因为：sum_pos + sum_neg = sum
 * 联立解得：sum_pos = (sum + target) / 2
 * 因此问题转化为：找到一个子集，其和等于(sum + target)/2，这样的子集有多少个？
 * 注意：必须满足 sum + target 为偶数且 sum + target >= 0，否则不存在这样的子集。
 *
 * 参数：
 * nums: 输入的整数数组
 * numsSize: 数组长度
 * target: 目标和
 * 返回值：
 * 满足条件的表达式数目
 */
```

```

int findTargetSumWays(int* nums, int numsSize, int target) {
 // 参数验证
 if (numsSize == 0) {
 return 0;
 }

 // 计算数组总和
 int sum = 0;
 for (int i = 0; i < numsSize; i++) {
 sum += nums[i];
 }

 // 检查是否存在解的条件
 // 1. sum + target 必须是非负数
 // 2. sum + target 必须是偶数
 if (sum + target < 0 || (sum + target) % 2 != 0) {
 return 0;
 }

 // 计算目标和，即我们需要找的子集的和
 // 通过数学推导: sum_pos = (sum + target) / 2
 int targetSum = (sum + target) / 2;

 // 创建 dp 数组, dp[j] 表示和为 j 的子集数目
 // 这是一个计数型的 01 背包问题
 int dp[MAX_TARGET];
 for (int i = 0; i <= targetSum; i++) {
 dp[i] = 0;
 }
 dp[0] = 1; // 基础情况: 空集的和为 0, 只有一种方式 (不选择任何元素)

 // 遍历每个物品 (数字)
 for (int i = 0; i < numsSize; i++) {
 // 01 背包问题, 逆序遍历容量
 // 这样可以保证每个元素只使用一次
 for (int j = targetSum; j >= nums[i]; j--) {
 // 状态转移: 当前和 j 的方式数目 = 不选当前数字的方式数目 + 选当前数字的方式数目
 // dp[j] = 不选择当前数字 + 选择当前数字
 // 不选择当前数字: dp[j] (保持原值)
 // 选择当前数字: dp[j - nums[i]] (前一个状态)
 dp[j] += dp[j - nums[i]];
 }
 }
}

```

```

// 返回和为 targetSum 的子集数目
return dp[targetSum];
}

/***
 * 优化版本：增加一些剪枝条件
 *
 * 参数：
 * nums: 输入的整数数组
 * numsSize: 数组长度
 * target: 目标和
 * 返回值：
 * 满足条件的表达式数目
*/
int findTargetSumWaysOptimized(int* nums, int numsSize, int target) {
 // 参数验证
 if (numsSize == 0) {
 return 0;
 }

 // 计算数组总和和最大值
 int sum = 0;
 int maxNum = 0;
 for (int i = 0; i < numsSize; i++) {
 sum += nums[i];
 if (nums[i] > maxNum) {
 maxNum = nums[i];
 }
 }

 // 检查是否存在解的条件
 // 1. target 不能超过数组总和的绝对值范围
 // 2. sum + target 必须是偶数
 if (target > sum || target < -sum || (sum + target) % 2 != 0) {
 return 0;
 }

 int targetSum = (sum + target) / 2;

 // 提前剪枝：如果目标和小于 0 或大于总和，返回 0
 if (targetSum < 0 || targetSum > sum) {
 return 0;
 }

 // 使用动态规划解决子集和问题
 int dp[targetSum + 1];
 dp[0] = 1; // 空集有一种方式
 for (int i = 1; i <= targetSum; i++) {
 dp[i] = 0;
 for (int j = 0; j < numsSize; j++) {
 if (nums[j] <= i) {
 dp[i] += dp[i - nums[j]];
 }
 }
 }

 return dp[targetSum];
}

```

```
}

// 创建 dp 数组
int dp[MAX_TARGET];
for (int i = 0; i <= targetSum; i++) {
 dp[i] = 0;
}
dp[0] = 1;

for (int i = 0; i < numsSize; i++) {
 // 优化: 如果当前数字大于 targetSum, 可以跳过
 // 因为当前数字本身就比目标和大, 无法用于组成目标和
 if (nums[i] > targetSum) {
 continue;
 }

 for (int j = targetSum; j >= nums[i]; j--) {
 dp[j] += dp[j - nums[i]];
 }
}

return dp[targetSum];
}

/*
 * 示例:
 * 输入: nums = [1,1,1,1,1], target = 3
 * 输出: 5
 * 解释: 有 5 种不同的表达式使结果等于 3。
 *
 * 输入: nums = [1], target = 1
 * 输出: 1
 *
 * 时间复杂度: O(n * target)
 * - 外层循环遍历所有元素: O(n)
 * - 内层循环遍历目标值: O(target)
 * 空间复杂度: O(target)
 * - 一维 DP 数组的空间消耗
 */
```

```
=====
package class073;

// LeetCode 494. 目标和
// 题目描述：给你一个整数数组 nums 和一个整数 target 。
// 向数组中的每个整数前添加 '+' 或 '-'，然后串联起所有整数，可以构造一个 表达式 ：
// 例如，nums = [2, 1]，可以在 2 之前添加 '+'，在 1 之前添加 '-'，得到表达式 "+2-1" 。
// 返回可以通过上述方法构造的、运算结果等于 target 的不同 表达式 的数目。
// 链接: https://leetcode.cn/problems/target-sum/
//
// 解题思路：
// 这是一个 01 背包问题的变形。我们需要将数组分成两个部分：正数部分和负数部分，使得它们的和等于 target。
// 设正数部分的和为 sum_pos，负数部分的和为 sum_neg，整个数组的和为 sum。
// 则有：sum_pos - sum_neg = target
// 又因为：sum_pos + sum_neg = sum
// 联立解得：sum_pos = (sum + target) / 2
// 因此问题转化为：找到一个子集，其和等于(sum + target)/2，这样的子集有多少个？
// 注意：必须满足 sum + target 为偶数且 sum + target >= 0，否则不存在这样的子集。
//
// 时间复杂度：O(n * target)
// 空间复杂度：O(target)

public class Code16_TargetSum {

 // 主方法，用于测试
 public static void main(String[] args) {
 // 测试用例 1
 int[] nums1 = {1, 1, 1, 1, 1};
 int target1 = 3;
 System.out.println("测试用例 1 结果：" + findTargetSumWays(nums1, target1)); // 预期输出：
5

 // 测试用例 2
 int[] nums2 = {1};
 int target2 = 1;
 System.out.println("测试用例 2 结果：" + findTargetSumWays(nums2, target2)); // 预期输出：
1

 // 测试用例 3
 int[] nums3 = {1, 2, 3, 4, 5};
 int target3 = 3;
 System.out.println("测试用例 3 结果：" + findTargetSumWays(nums3, target3)); // 预期输出：
}
```

```
}
```

```
/**
```

```
* 计算可以构造目标和的不同表达式的数目
```

```
*
```

```
* 解题思路:
```

\* 这是一个01背包问题的变形。我们需要将数组分成两个部分：正数部分和负数部分，使得它们的和等于target。

\* 设正数部分的和为sum\_pos，负数部分的和为sum\_neg，整个数组的和为sum。

\* 则有：sum\_pos - sum\_neg = target

\* 又因为：sum\_pos + sum\_neg = sum

\* 联立解得：sum\_pos = (sum + target) / 2

\* 因此问题转化为：找到一个子集，其和等于(sum + target)/2，这样的子集有多少个？

\* 注意：必须满足sum + target为偶数且sum + target >= 0，否则不存在这样的子集。

```
*
```

\* @param nums 输入的整数数组

\* @param target 目标和

\* @return 满足条件的表达式数目

```
*/
```

```
public static int findTargetSumWays(int[] nums, int target) {
```

```
 // 参数验证
```

```
 if (nums == null || nums.length == 0) {
```

```
 return 0;
```

```
}
```

```
 // 计算数组总和
```

```
 int sum = 0;
```

```
 for (int num : nums) {
```

```
 sum += num;
```

```
}
```

```
 // 检查是否存在解的条件
```

```
 // 1. sum + target 必须是非负数
```

```
 // 2. sum + target 必须是偶数
```

```
 if (sum + target < 0 || (sum + target) % 2 != 0) {
```

```
 return 0;
```

```
}
```

```
 // 计算目标和，即我们需要找的子集的和
```

```
 // 通过数学推导：sum_pos = (sum + target) / 2
```

```
 int targetSum = (sum + target) / 2;
```

```

// 创建 dp 数组, dp[j] 表示和为 j 的子集数目
// 这是一个计数型的 01 背包问题
int[] dp = new int[targetSum + 1];
dp[0] = 1; // 基础情况: 空集的和为 0, 只有一种方式 (不选择任何元素)

// 遍历每个物品 (数字)
for (int i = 0; i < nums.length; i++) {
 // 01 背包问题, 逆序遍历容量
 // 这样可以保证每个元素只使用一次
 for (int j = targetSum; j >= nums[i]; j--) {
 // 状态转移: 当前和 j 的方式数目 = 不选当前数字的方式数目 + 选当前数字的方式数目
 // dp[j] = 不选择当前数字 + 选择当前数字
 // 不选择当前数字: dp[j] (保持原值)
 // 选择当前数字: dp[j - nums[i]] (前一个状态)
 dp[j] += dp[j - nums[i]];
 }
}

// 返回和为 targetSum 的子集数目
return dp[targetSum];
}

/***
 * 优化版本: 增加一些剪枝条件
 *
 * @param nums 输入的整数数组
 * @param target 目标和
 * @return 满足条件的表达式数目
 */
public static int findTargetSumWaysOptimized(int[] nums, int target) {
 // 参数验证
 if (nums == null || nums.length == 0) {
 return 0;
 }

 // 计算数组总和和最大值
 int sum = 0;
 int maxNum = 0;
 for (int num : nums) {
 sum += num;
 maxNum = Math.max(maxNum, num);
 }
}

```

```

// 检查是否存在解的条件
// 1. target 不能超过数组总和的绝对值范围
// 2. sum + target 必须是偶数
if (target > sum || target < -sum || (sum + target) % 2 != 0) {
 return 0;
}

int targetSum = (sum + target) / 2;

// 提前剪枝：如果目标和小于 0 或大于总和，返回 0
if (targetSum < 0 || targetSum > sum) {
 return 0;
}

// 创建 dp 数组
int[] dp = new int[targetSum + 1];
dp[0] = 1;

for (int i = 0; i < nums.length; i++) {
 // 优化：如果当前数字大于 targetSum，可以跳过
 // 因为当前数字本身就比目标和大，无法用于组成目标和
 if (nums[i] > targetSum) {
 continue;
 }

 for (int j = targetSum; j >= nums[i]; j--) {
 dp[j] += dp[j - nums[i]];
 }
}

return dp[targetSum];
}

/*
 * 示例：
 * 输入：nums = [1, 1, 1, 1, 1], target = 3
 * 输出：5
 * 解释：有 5 种不同的表达式使结果等于 3。
 *
 * 输入：nums = [1], target = 1
 * 输出：1
 *
 * 时间复杂度：O(n * target)

```

```

* - 外层循环遍历所有元素: O(n)
* - 内层循环遍历目标值: O(target)
* 空间复杂度: O(target)
* - 一维 DP 数组的空间消耗
*/
}

```

文件: Code16\_TargetSum.py

```

LeetCode 494. 目标和
题目描述: 给你一个整数数组 nums 和一个整数 target 。
向数组中的每个整数前添加 '+' 或 '-'，然后串联起所有整数，可以构造一个 表达式 ：
例如，nums = [2, 1]，可以在 2 之前添加 '+'，在 1 之前添加 '-'，得到表达式 "+2-1" 。
返回可以通过上述方法构造的、运算结果等于 target 的不同 表达式 的数目。
链接: https://leetcode.cn/problems/target-sum/
#
解题思路:
这是一个 01 背包问题的变形。我们需要将数组分成两个部分：正数部分和负数部分，使得它们的和等于 target。
设正数部分的和为 sum_pos，负数部分的和为 sum_neg，整个数组的和为 sum。
则有: sum_pos - sum_neg = target
又因为: sum_pos + sum_neg = sum
联立解得: sum_pos = (sum + target) / 2
因此问题转化为：找到一个子集，其和等于 $(sum + target)/2$ ，这样的子集有多少个？
注意：必须满足 sum + target 为偶数且 sum + target ≥ 0 ，否则不存在这样的子集。
#
时间复杂度: O(n * target)
空间复杂度: O(target)

```

```
def find_target_sum_ways(nums, target):
 """

```

计算可以构造目标和的不同表达式的数目

解题思路:

这是一个 01 背包问题的变形。我们需要将数组分成两个部分：正数部分和负数部分，使得它们的和等于 target。

设正数部分的和为 sum\_pos，负数部分的和为 sum\_neg，整个数组的和为 sum。

则有: sum\_pos - sum\_neg = target

又因为: sum\_pos + sum\_neg = sum

联立解得: sum\_pos =  $(sum + target) / 2$

因此问题转化为：找到一个子集，其和等于  $(sum + target)/2$ ，这样的子集有多少个？

注意：必须满足  $\text{sum} + \text{target}$  为偶数且  $\text{sum} + \text{target} \geq 0$ ，否则不存在这样的子集。

Args:

  nums: 输入的整数数组

  target: 目标和

Returns:

  int: 满足条件的表达式数目

"""

# 参数验证

if not nums:

    return 0

# 计算数组总和

total\_sum = sum(nums)

# 检查是否存在解的条件

# 1.  $\text{sum} + \text{target}$  必须是非负数

# 2.  $\text{sum} + \text{target}$  必须是偶数

if total\_sum + target < 0 or (total\_sum + target) % 2 != 0:

    return 0

# 计算目标和，即我们需要找的子集的和

# 通过数学推导:  $\text{sum\_pos} = (\text{sum} + \text{target}) / 2$

target\_sum = (total\_sum + target) // 2

# 创建 dp 数组,  $\text{dp}[j]$  表示和为  $j$  的子集数目

# 这是一个计数型的 01 背包问题

dp = [0] \* (target\_sum + 1)

dp[0] = 1 # 基础情况: 空集的和为 0, 只有一种方式 (不选择任何元素)

# 遍历每个物品 (数字)

for num in nums:

    # 01 背包问题, 逆序遍历容量

    # 这样可以保证每个元素只使用一次

    for j in range(target\_sum, num - 1, -1):

        # 状态转移: 当前和  $j$  的方式数目 = 不选当前数字的方式数目 + 选当前数字的方式数目

        #  $\text{dp}[j] = \text{不选择当前数字} + \text{选择当前数字}$

        # 不选择当前数字:  $\text{dp}[j]$  (保持原值)

        # 选择当前数字:  $\text{dp}[j - \text{num}]$  (前一个状态)

        dp[j] += dp[j - num]

# 返回和为 target\_sum 的子集数目

```
return dp[target_sum]
```

```
def find_target_sum_ways_optimized(nums, target):
```

```
 """
```

```
 优化版本：增加一些剪枝条件
```

```
Args:
```

```
 nums: 输入的整数数组
```

```
 target: 目标和
```

```
Returns:
```

```
 int: 满足条件的表达式数目
```

```
 """
```

```
参数验证
```

```
if not nums:
```

```
 return 0
```

```
计算数组总和和最大值
```

```
total_sum = sum(nums)
```

```
max_num = max(nums)
```

```
检查是否存在解的条件
```

```
1. target 不能超过数组总和的绝对值范围
```

```
2. sum + target 必须是偶数
```

```
if target > total_sum or target < -total_sum or (total_sum + target) % 2 != 0:
```

```
 return 0
```

```
target_sum = (total_sum + target) // 2
```

```
提前剪枝：如果目标和小于 0 或大于总和，返回 0
```

```
if target_sum < 0 or target_sum > total_sum:
```

```
 return 0
```

```
创建 dp 数组
```

```
dp = [0] * (target_sum + 1)
```

```
dp[0] = 1
```

```
for num in nums:
```

```
 # 优化：如果当前数字大于 target_sum，可以跳过
```

```
 # 因为当前数字本身就比目标和大，无法用于组成目标和
```

```
 if num > target_sum:
```

```
 continue
```

```

 for j in range(target_sum, num - 1, -1):
 dp[j] += dp[j - num]

 return dp[target_sum]

测试用例
if __name__ == "__main__":
 # 测试用例 1
 nums1 = [1, 1, 1, 1, 1]
 target1 = 3
 print(f"测试用例 1 结果: {find_target_sum_ways(nums1, target1)}") # 预期输出: 5

 # 测试用例 2
 nums2 = [1]
 target2 = 1
 print(f"测试用例 2 结果: {find_target_sum_ways(nums2, target2)}") # 预期输出: 1

 # 测试用例 3
 nums3 = [1, 2, 3, 4, 5]
 target3 = 3
 print(f"测试用例 3 结果: {find_target_sum_ways(nums3, target3)}") # 预期输出: 3

```

,,

示例:

输入: nums = [1, 1, 1, 1, 1], target = 3

输出: 5

解释: 有 5 种不同的表达式使结果等于 3。

输入: nums = [1], target = 1

输出: 1

时间复杂度:  $O(n * target)$

- 外层循环遍历所有元素:  $O(n)$
- 内层循环遍历目标值:  $O(target)$

空间复杂度:  $O(target)$

- 一维 DP 数组的空间消耗

,,

---

文件: Code17\_OnesAndZeros.cpp

---

// LeetCode 474. 一和零

```
// 题目描述：给你一个二进制字符串数组 strs 和两个整数 m 和 n 。
// 请你找出并返回 strs 的最大子集的长度，该子集中 最多 有 m 个 0 和 n 个 1 。
// 如果 x 的所有元素也是 y 的元素，集合 x 是集合 y 的 子集 。
// 链接: https://leetcode.cn/problems/ones-and-zeroes/
//
// 解题思路：
// 这是一个二维费用的 01 背包问题。
// 每个字符串可以看作一个物品，它有两个「费用」：0 的数量和 1 的数量
// 背包的容量是两个维度的：最多可以装 m 个 0 和 n 个 1
// 每个物品的「价值」都是 1（因为我们要最大化子集的长度）
//
// 状态定义：dp[i][j] 表示最多使用 i 个 0 和 j 个 1 时能组成的最大子集长度
// 状态转移方程：dp[i][j] = max(dp[i][j], dp[i-zeroCount][j-oneCount] + 1)
// 其中 zeroCount 是当前字符串中 0 的数量，oneCount 是当前字符串中 1 的数量
//
// 时间复杂度：O(1 * m * n)，其中 1 是字符串数组的长度
// 空间复杂度：O(m * n)，使用二维 DP 数组
```

```
#define MAX_M 101
#define MAX_N 101
```

```
// 获取两个数中的较大值
int max(int a, int b) {
 return a > b ? a : b;
}

/**
 * 计算最多使用 m 个 0 和 n 个 1 时能组成的最大子集长度
 *
 * 解题思路：
 * 这是一个二维费用的 01 背包问题。
 * 每个字符串可以看作一个物品，它有两个「费用」：0 的数量和 1 的数量
 * 背包的容量是两个维度的：最多可以装 m 个 0 和 n 个 1
 * 每个物品的「价值」都是 1（因为我们要最大化子集的长度）
 *
 * 参数：
 * strs: 二进制字符串数组
 * strsSize: 字符串数组的大小
 * m: 最多可以使用的 0 的个数
 * n: 最多可以使用的 1 的个数
 * 返回值：
 * 最大子集长度
 */
```

```

int findMaxForm(char** strs, int strsSize, int m, int n) {
 // 参数验证
 if (strsSize == 0) {
 return 0;
 }

 // 创建二维 DP 数组, dp[i][j] 表示最多使用 i 个 0 和 j 个 1 时能组成
 // 的最大子集长度
 // 这是一个二维费用的 01 背包问题
 int dp[MAX_M][MAX_N];

 // 初始化 DP 数组
 for (int i = 0; i <= m; i++) {
 for (int j = 0; j <= n; j++) {
 dp[i][j] = 0;
 }
 }

 // 遍历每个字符串 (物品)
 for (int idx = 0; idx < strsSize; idx++) {
 // 计算当前字符串中 0 和 1 的数量
 // 这相当于获取当前物品的两个费用属性
 int zeroCount = 0, oneCount = 0;
 for (int i = 0; strs[idx][i] != '\0'; i++) {
 if (strs[idx][i] == '0') {
 zeroCount++;
 } else {
 oneCount++;
 }
 }

 // 二维 01 背包, 需要倒序遍历两个维度, 避免重复计算
 // i 表示当前可用的 0 的个数, j 表示当前可用的 1 的个数
 // 倒序遍历确保每个物品只使用一次
 for (int i = m; i >= zeroCount; i--) {
 for (int j = n; j >= oneCount; j--) {
 // 状态转移: 选择当前字符串或不选择当前字符串
 // dp[i][j] = max(不选择当前字符串, 选择当前字符串)
 // 不选择当前字符串: dp[i][j] (保持原值)
 // 选择当前字符串: dp[i - zeroCount][j - oneCount] + 1 (前一个状态+1)
 dp[i][j] = max(dp[i][j], dp[i - zeroCount][j - oneCount] + 1);
 }
 }
 }
}

```

```

// 返回最多使用 m 个 0 和 n 个 1 时能组成的大子集长度
return dp[m][n];
}

/***
 * 优化版本：预处理计算每个字符串的 0 和 1 数量，减少重复计算
 *
 * 参数：
 * strs: 二进制字符串数组
 * strsSize: 字符串数组的大小
 * m: 最多可以使用的 0 的个数
 * n: 最多可以使用的 1 的个数
 * 返回值：
 * 最大子集长度
 */
int findMaxFormOptimized(char** strs, int strsSize, int m, int n) {
 // 参数验证
 if (strsSize == 0) {
 return 0;
 }

 // 预处理：计算每个字符串中 0 和 1 的数量
 // 这样可以避免在动态规划过程中重复计算
 int counts[600][2]; // 假设最多 600 个字符串
 for (int i = 0; i < strsSize; i++) {
 int zeros = 0, ones = 0;
 for (int j = 0; strs[i][j] != '\0'; j++) {
 if (strs[i][j] == '0') zeros++;
 else ones++;
 }
 counts[i][0] = zeros;
 counts[i][1] = ones;
 }

 // 创建二维 DP 数组
 int dp[MAX_M][MAX_N];
 for (int i = 0; i <= m; i++) {
 for (int j = 0; j <= n; j++) {
 dp[i][j] = 0;
 }
 }
}

```

```

// 遍历每个字符串（物品）
for (int i = 0; i < strsSize; i++) {
 int zeroCount = counts[i][0];
 int oneCount = counts[i][1];

 // 剪枝：如果当前字符串的 0 或 1 数量超过背包容量，直接跳过
 // 因为当前字符串本身就无法放入背包
 if (zeroCount > m || oneCount > n) {
 continue;
 }

 // 二维 01 背包，倒序遍历
 for (int j = m; j >= zeroCount; j--) {
 for (int k = n; k >= oneCount; k--) {
 dp[j][k] = max(dp[j][k], dp[j - zeroCount][k - oneCount] + 1);
 }
 }
}

return dp[m][n];
}

/*
 * 示例：
 * 输入：strs = ["10", "0001", "111001", "1", "0"], m = 5, n = 3
 * 输出：4
 * 解释：最多有 5 个 0 和 3 个 1 的子集是 {"10", "0001", "1", "0"}，因此答案是 4。
 *
 * 输入：strs = ["10", "0", "1"], m = 1, n = 1
 * 输出：2
 * 解释：最多有 1 个 0 和 1 个 1 的子集是 {"0", "1"}，因此答案是 2。
 *
 * 时间复杂度：O(1 * m * n)
 * - 外层循环遍历所有字符串：O(1)
 * - 中层循环遍历 m：O(m)
 * - 内层循环遍历 n：O(n)
 * 空间复杂度：O(m * n)
 * - 二维 DP 数组的空间消耗
*/

```

```
=====
package class073;

// LeetCode 474. 一和零
// 题目描述：给你一个二进制字符串数组 strs 和两个整数 m 和 n 。
// 请你找出并返回 strs 的最大子集的长度，该子集中 最多 有 m 个 0 和 n 个 1 。
// 如果 x 的所有元素也是 y 的元素，集合 x 是集合 y 的 子集 。
// 链接：https://leetcode.cn/problems/ones-and-zeroes/
//
// 解题思路：
// 这是一个二维费用的 01 背包问题。
// 每个字符串可以看作一个物品，它有两个「费用」：0 的数量和 1 的数量
// 背包的容量是两个维度的：最多可以装 m 个 0 和 n 个 1
// 每个物品的「价值」都是 1（因为我们要最大化子集的长度）
//
// 状态定义：dp[i][j] 表示最多使用 i 个 0 和 j 个 1 时能组成的最大子集长度
// 状态转移方程：dp[i][j] = max(dp[i][j], dp[i-zeroCount][j-oneCount] + 1)
// 其中 zeroCount 是当前字符串中 0 的数量，oneCount 是当前字符串中 1 的数量
//
// 时间复杂度：O(l * m * n)，其中 l 是字符串数组的长度
// 空间复杂度：O(m * n)，使用二维 DP 数组
```

```
public class Code17_OnesAndZeros {

 // 主方法，用于测试
 public static void main(String[] args) {
 // 测试用例 1
 String[] strs1 = {"10", "0001", "111001", "1", "0"};
 int m1 = 5, n1 = 3;
 System.out.println("测试用例 1 结果：" + findMaxForm(strs1, m1, n1)); // 预期输出：4

 // 测试用例 2
 String[] strs2 = {"10", "0", "1"};
 int m2 = 1, n2 = 1;
 System.out.println("测试用例 2 结果：" + findMaxForm(strs2, m2, n2)); // 预期输出：2

 // 测试用例 3
 String[] strs3 = {"00", "01", "11", "10"};
 int m3 = 2, n3 = 2;
 System.out.println("测试用例 3 结果：" + findMaxForm(strs3, m3, n3)); // 预期输出：2
 }

 /**

```

```

* 计算最多使用 m 个 0 和 n 个 1 时能组成的大子集长度
*
* 解题思路：
* 这是一个二维费用的 01 背包问题。
* 每个字符串可以看作一个物品，它有两个「费用」：0 的数量和 1 的数量
* 背包的容量是两个维度的：最多可以装 m 个 0 和 n 个 1
* 每个物品的「价值」都是 1（因为我们要最大化子集的长度）
*
* @param strs 二进制字符串数组
* @param m 最多可以使用的 0 的个数
* @param n 最多可以使用的 1 的个数
* @return 最大子集长度
*/
public static int findMaxForm(String[] strs, int m, int n) {
 // 参数验证
 if (strs == null || strs.length == 0) {
 return 0;
 }

 // 创建二维 DP 数组，dp[i][j] 表示最多使用 i 个 0 和 j 个 1 时能组成的大子集长度
 // 这是一个二维费用的 01 背包问题
 int[][] dp = new int[m + 1][n + 1];

 // 遍历每个字符串（物品）
 for (String str : strs) {
 // 计算当前字符串中 0 和 1 的数量
 // 这相当于获取当前物品的两个费用属性
 int zeroCount = 0, oneCount = 0;
 for (char c : str.toCharArray()) {
 if (c == '0') {
 zeroCount++;
 } else {
 oneCount++;
 }
 }

 // 二维 01 背包，需要倒序遍历两个维度，避免重复计算
 // i 表示当前可用的 0 的个数，j 表示当前可用的 1 的个数
 // 倒序遍历确保每个物品只使用一次
 for (int i = m; i >= zeroCount; i--) {
 for (int j = n; j >= oneCount; j--) {
 // 状态转移：选择当前字符串或不选择当前字符串
 // dp[i][j] = max(不选择当前字符串, 选择当前字符串)
 }
 }
 }
}

```

```

 // 不选择当前字符串: dp[i][j] (保持原值)
 // 选择当前字符串: dp[i - zeroCount][j - oneCount] + 1 (前一个状态+1)
 dp[i][j] = Math.max(dp[i][j], dp[i - zeroCount][j - oneCount] + 1);
 }
}
}

// 返回最多使用 m 个 0 和 n 个 1 时能组成的大子集长度
return dp[m][n];
}

/***
 * 优化版本: 预处理计算每个字符串的 0 和 1 数量, 减少重复计算
 *
 * @param strs 二进制字符串数组
 * @param m 最多可以使用的 0 的个数
 * @param n 最多可以使用的 1 的个数
 * @return 最大子集长度
 */
public static int findMaxFormOptimized(String[] strs, int m, int n) {
 // 参数验证
 if (strs == null || strs.length == 0) {
 return 0;
 }

 // 预处理: 计算每个字符串中 0 和 1 的数量
 // 这样可以避免在动态规划过程中重复计算
 int[][] counts = new int[strs.length][2];
 for (int i = 0; i < strs.length; i++) {
 int zeros = 0, ones = 0;
 for (char c : strs[i].toCharArray()) {
 if (c == '0') zeros++;
 else ones++;
 }
 counts[i][0] = zeros;
 counts[i][1] = ones;
 }

 // 创建二维 DP 数组
 int[][] dp = new int[m + 1][n + 1];

 // 遍历每个字符串 (物品)
 for (int[] count : counts) {

```

```

 int zeroCount = count[0];
 int oneCount = count[1];

 // 剪枝：如果当前字符串的 0 或 1 数量超过背包容量，直接跳过
 // 因为当前字符串本身就无法放入背包
 if (zeroCount > m || oneCount > n) {
 continue;
 }

 // 二维 01 背包，倒序遍历
 for (int i = m; i >= zeroCount; i--) {
 for (int j = n; j >= oneCount; j--) {
 dp[i][j] = Math.max(dp[i][j], dp[i - zeroCount][j - oneCount] + 1);
 }
 }
 }

 return dp[m][n];
}

/**
 * 进一步优化：使用滚动数组（虽然这里已经是二维，但可以看作是特殊的滚动数组）
 * 并添加更多剪枝条件
 *
 * @param strs 二进制字符串数组
 * @param m 最多可以使用的 0 的个数
 * @param n 最多可以使用的 1 的个数
 * @return 最大子集长度
 */
public static int findMaxFormFurtherOptimized(String[] strs, int m, int n) {
 // 参数验证
 if (strs == null || strs.length == 0 || (m == 0 && n == 0)) {
 return 0;
 }

 // 预处理并过滤不符合条件的字符串
 // 只保留那些至少有一个维度不超过背包容量的字符串
 int[][] validCounts = new int[strs.length][2];
 int validCount = 0;

 for (String str : strs) {
 int zeros = 0, ones = 0;
 for (char c : str.toCharArray()) {

```

```

 if (c == '0') zeros++;
 else ones++;
 }

 // 剪枝：如果当前字符串的 0 和 1 数量都超过背包容量，直接跳过
 // 修改条件：只要至少有一个维度不超过背包容量就可以考虑
 if (zeros <= m || ones <= n) {
 validCounts[validCount][0] = zeros;
 validCounts[validCount][1] = ones;
 validCount++;
 }
}

// 创建二维 DP 数组
int[][] dp = new int[m + 1][n + 1];

// 遍历每个有效的字符串
for (int i = 0; i < validCount; i++) {
 int zeroCount = validCounts[i][0];
 int oneCount = validCounts[i][1];

 // 二维 01 背包，倒序遍历
 for (int j = m; j >= zeroCount; j--) {
 for (int k = n; k >= oneCount; k--) {
 dp[j][k] = Math.max(dp[j][k], dp[j - zeroCount][k - oneCount] + 1);
 }
 }
}

return dp[m][n];
}

/*
 * 示例：
 * 输入：strs = ["10", "0001", "111001", "1", "0"], m = 5, n = 3
 * 输出：4
 * 解释：最多有 5 个 0 和 3 个 1 的子集是 {"10", "0001", "1", "0"}，因此答案是 4。
 *
 * 输入：strs = ["10", "0", "1"], m = 1, n = 1
 * 输出：2
 * 解释：最多有 1 个 0 和 1 个 1 的子集是 {"0", "1"}，因此答案是 2。
 *
 * 时间复杂度：O(1 * m * n)

```

```
* - 外层循环遍历所有字符串: O(1)
* - 中层循环遍历 m: O(m)
* - 内层循环遍历 n: O(n)
* 空间复杂度: O(m * n)
* - 二维 DP 数组的空间消耗
*/
}
```

=====

文件: Code17\_OnesAndZeros.py

=====

```
LeetCode 474. 一和零
题目描述: 给你一个二进制字符串数组 strs 和两个整数 m 和 n 。
请你找出并返回 strs 的最大子集的长度, 该子集中 最多 有 m 个 0 和 n 个 1 。
如果 x 的所有元素也是 y 的元素, 集合 x 是集合 y 的 子集 。
链接: https://leetcode.cn/problems/ones-and-zeroes/
#
解题思路:
这是一个二维费用的 01 背包问题。
每个字符串可以看作一个物品, 它有两个「费用」: 0 的数量和 1 的数量
背包的容量是两个维度的: 最多可以装 m 个 0 和 n 个 1
每个物品的「价值」都是 1 (因为我们要最大化子集的长度)
#
状态定义: dp[i][j] 表示最多使用 i 个 0 和 j 个 1 时能组成的大子集长度
状态转移方程: dp[i][j] = max(dp[i][j], dp[i-zeroCount][j-oneCount] + 1)
其中 zeroCount 是当前字符串中 0 的数量, oneCount 是当前字符串中 1 的数量
#
时间复杂度: O(l * m * n), 其中 l 是字符串数组的长度
空间复杂度: O(m * n), 使用二维 DP 数组
```

```
def find_max_form(strs, m, n):
 """
计算最多使用 m 个 0 和 n 个 1 时能组成的大子集长度
```

解题思路:

这是一个二维费用的 01 背包问题。

每个字符串可以看作一个物品, 它有两个「费用」: 0 的数量和 1 的数量

背包的容量是两个维度的: 最多可以装 m 个 0 和 n 个 1

每个物品的「价值」都是 1 (因为我们要最大化子集的长度)

Args:

strs: 二进制字符串数组

m: 最多可以使用的 0 的个数  
n: 最多可以使用的 1 的个数

Returns:

int: 最大子集长度

"""

# 参数验证

```
if not strs:
 return 0
```

# 创建二维 DP 数组, dp[i][j] 表示最多使用 i 个 0 和 j 个 1 时能组成的大子集长度

# 这是一个二维费用的 01 背包问题

```
dp = [[0] * (n + 1) for _ in range(m + 1)]
```

# 遍历每个字符串 (物品)

```
for s in strs:
```

# 计算当前字符串中 0 和 1 的数量

# 这相当于获取当前物品的两个费用属性

```
zero_count = s.count('0')
```

```
one_count = len(s) - zero_count
```

# 二维 01 背包, 需要倒序遍历两个维度, 避免重复计算

# i 表示当前可用的 0 的个数, j 表示当前可用的 1 的个数

# 倒序遍历确保每个物品只使用一次

```
for i in range(m, zero_count - 1, -1):
```

```
 for j in range(n, one_count - 1, -1):
```

# 状态转移: 选择当前字符串或不选择当前字符串

# dp[i][j] = max(不选择当前字符串, 选择当前字符串)

# 不选择当前字符串: dp[i][j] (保持原值)

# 选择当前字符串: dp[i - zero\_count][j - one\_count] + 1 (前一个状态+1)

```
 dp[i][j] = max(dp[i][j], dp[i - zero_count][j - one_count] + 1)
```

# 返回最多使用 m 个 0 和 n 个 1 时能组成的大子集长度

```
return dp[m][n]
```

```
def find_max_form_optimized(strs, m, n):
```

"""

优化版本: 预处理计算每个字符串的 0 和 1 数量, 减少重复计算

Args:

strs: 二进制字符串数组

m: 最多可以使用的 0 的个数

n: 最多可以使用的 1 的个数

Returns:

int: 最大子集长度

"""

# 参数验证

if not strs:

    return 0

# 预处理: 计算每个字符串中 0 和 1 的数量

# 这样可以避免在动态规划过程中重复计算

counts = []

for s in strs:

    zero\_count = s.count('0')

    one\_count = len(s) - zero\_count

    counts.append((zero\_count, one\_count))

# 创建二维 DP 数组

dp = [[0] \* (n + 1) for \_ in range(m + 1)]

# 遍历每个字符串 (物品)

for zero\_count, one\_count in counts:

    # 剪枝: 如果当前字符串的 0 或 1 数量超过背包容量, 直接跳过

    # 因为当前字符串本身就无法放入背包

    if zero\_count > m or one\_count > n:

        continue

    # 二维 01 背包, 倒序遍历

    for i in range(m, zero\_count - 1, -1):

        for j in range(n, one\_count - 1, -1):

            dp[i][j] = max(dp[i][j], dp[i - zero\_count][j - one\_count] + 1)

return dp[m][n]

def find\_max\_form\_further\_optimized(strs, m, n):

"""

进一步优化: 使用滚动数组 (虽然这里已经是二维, 但可以看作是特殊的滚动数组)

并添加更多剪枝条件

Args:

strs: 二进制字符串数组

m: 最多可以使用的 0 的个数

n: 最多可以使用的 1 的个数

Returns:

int: 最大子集长度

"""

# 参数验证

if not strs or (m == 0 and n == 0):

return 0

# 预处理并过滤不符合条件的字符串

# 只保留那些至少有一个维度不超过背包容量的字符串

valid\_counts = []

for s in strs:

zero\_count = s.count('0')

one\_count = len(s) - zero\_count

# 剪枝: 如果当前字符串的 0 和 1 数量都超过背包容量, 直接跳过

# 修改条件: 只要至少有一个维度不超过背包容量就可以考虑

if zero\_count <= m or one\_count <= n:

valid\_counts.append((zero\_count, one\_count))

# 创建二维 DP 数组

dp = [[0] \* (n + 1) for \_ in range(m + 1)]

# 遍历每个有效的字符串

for zero\_count, one\_count in valid\_counts:

# 二维 01 背包, 倒序遍历

for i in range(m, zero\_count - 1, -1):

for j in range(n, one\_count - 1, -1):

dp[i][j] = max(dp[i][j], dp[i - zero\_count][j - one\_count] + 1)

return dp[m][n]

# 测试用例

if \_\_name\_\_ == "\_\_main\_\_":

# 测试用例 1

strs1 = ["10", "0001", "111001", "1", "0"]

m1, n1 = 5, 3

print(f"测试用例 1 结果: {find\_max\_form(strs1, m1, n1)}") # 预期输出: 4

# 测试用例 2

strs2 = ["10", "0", "1"]

m2, n2 = 1, 1

print(f"测试用例 2 结果: {find\_max\_form(strs2, m2, n2)}") # 预期输出: 2

```
测试用例 3
strs3 = ["00", "01", "11", "10"]
m3, n3 = 2, 2
print(f"测试用例 3 结果: {find_max_form(strs3, m3, n3)}") # 预期输出: 2

,,,
```

示例:

输入: strs = ["10", "0001", "111001", "1", "0"], m = 5, n = 3

输出: 4

解释: 最多有 5 个 0 和 3 个 1 的子集是 {"10", "0001", "1", "0"}, 因此答案是 4。

输入: strs = ["10", "0", "1"], m = 1, n = 1

输出: 2

解释: 最多有 1 个 0 和 1 个 1 的子集是 {"0", "1"}, 因此答案是 2。

时间复杂度: O(1 \* m \* n)

- 外层循环遍历所有字符串: O(1)
- 中层循环遍历 m: O(m)
- 内层循环遍历 n: O(n)

空间复杂度: O(m \* n)

- 二维 DP 数组的空间消耗

,,

---

文件: Code18\_ProfitableSchemes.cpp

---

```
// LeetCode 879. 盈利计划
// 题目描述: 集团里有 n 名员工, 他们可以完成各种各样的工作创造利润。
// 第 i 种工作会产生 profit[i] 的利润, 它要求 group[i] 名成员共同参与。
// 如果成员参与了其中一项工作, 就不能参与另一项工作。
// 工作的任何至少产生 minProfit 利润的子集称为 盈利计划。
// 并且工作的成员总数最多为 n。
// 有多少种计划可以选择? 因为答案很大, 所以返回结果模 10^9 + 7 的值。
// 链接: https://leetcode.cn/problems/profitable-schemes/
//
// 解题思路:
// 这是一个三维费用的 01 背包问题。
// 三维分别是: 员工人数、利润、可选的工作数量
// 状态定义: dp[i][j][k] 表示考虑前 i 个工作, 使用 j 个员工, 获得至少 k 的利润的方案数
// 状态转移方程:
// - 不选择第 i 个工作: dp[i][j][k] = dp[i-1][j][k]
```

```

// - 选择第 i 个工作: dp[i][j][k] += dp[i-1][j-group[i-1]][Math.max(0, k-profit[i-1])]
//
// 时间复杂度: O(N * minProfit * n), 其中 N 是工作数量, n 是员工人数
// 空间复杂度: O(N * minProfit * n), 可以优化到 O(minProfit * n)

#define MOD 1000000007
#define MAX_N 101
#define MAX_P 101

// 获取两个数中的较大值
int max(int a, int b) {
 return a > b ? a : b;
}

// 获取两个数中的较小值
int min(int a, int b) {
 return a < b ? a : b;
}

/**
 * 计算盈利计划的数量
 *
 * 解题思路:
 * 这是一个三维费用的 01 背包问题。
 * 三维分别是: 员工人数、利润、可选的工作数量
 * 状态定义: dp[i][j][k] 表示考虑前 i 个工作, 使用 j 个员工, 获得至少 k 的利润的方案数
 * 状态转移方程:
 * - 不选择第 i 个工作: dp[i][j][k] = dp[i-1][j][k]
 * - 选择第 i 个工作: dp[i][j][k] += dp[i-1][j-group[i-1]][Math.max(0, k-profit[i-1])]
 *
 * 参数:
 * n: 员工总数
 * minProfit: 最小利润要求
 * group: 每个工作所需的员工数
 * groupSize: group 数组的大小
 * profit: 每个工作的利润
 * profitSize: profit 数组的大小
 *
 * 返回值:
 * 满足条件的计划数 (模 10^9+7)
 */
int profitableSchemes(int n, int minProfit, int* group, int groupSize, int* profit, int profitSize) {
 // 参数验证

```

```

if (n <= 0) {
 return minProfit <= 0 ? 1 : 0;
}

if (groupSize == 0 || profitSize == 0 || groupSize != profitSize) {
 return minProfit <= 0 ? 1 : 0;
}

int m = groupSize; // 工作数量

// 创建三维 DP 数组: dp[i][j][k] 表示考虑前 i 个工作, 使用 j 个员工, 获得至少 k 的利润的方案数
// 这是一个三维费用的 01 背包问题
int dp[101][MAX_N][MAX_P]; // 假设最多 100 个工作

// 初始化 DP 数组
for (int i = 0; i <= m; i++) {
 for (int j = 0; j <= n; j++) {
 for (int k = 0; k <= minProfit; k++) {
 dp[i][j][k] = 0;
 }
 }
}

// 初始状态: 没有选择任何工作时, 使用 0 个员工, 利润为 0 的方案数为 1
// 这是动态规划的边界条件
dp[0][0][0] = 1;

// 遍历每个工作 (物品)
for (int i = 1; i <= m; i++) {
 int g = group[i - 1]; // 当前工作所需的员工数 (第一个费用)
 int p = profit[i - 1]; // 当前工作的利润 (第二个费用)

 // 遍历员工数 (第一个费用维度)
 for (int j = 0; j <= n; j++) {
 // 遍历利润要求 (第二个费用维度)
 for (int k = 0; k <= minProfit; k++) {
 // 不选择当前工作的情况
 // 保持前一个状态的方案数
 dp[i][j][k] = dp[i - 1][j][k];

 // 选择当前工作的情况, 需要确保员工数足够
 if (j >= g) {
 // 计算选择当前工作后所需的最小利润
 // 如果 k < p, 则前一个状态需要的利润为 0 (因为负利润没有意义)
 }
 }
 }
}

```

```

 int prevProfit = max(0, k - p);
 // 更新方案数，注意取模
 // dp[i][j][k] = 不选择当前工作的方案数 + 选择当前工作的方案数
 long long newValue = (long long)dp[i][j][k] + dp[i - 1][j - g][prevProfit];
 dp[i][j][k] = newValue % MOD;
 }
}
}

// 计算所有可能的方案数：员工数不超过 n，利润至少为 minProfit
// 遍历所有可能的员工数，累加方案数
int result = 0;
for (int j = 0; j <= n; j++) {
 result = ((long long)result + dp[m][j][minProfit]) % MOD;
}

return result;
}

/***
 * 空间优化版本：使用二维 DP 数组
 *
 * 参数：
 * n: 员工总数
 * minProfit: 最小利润要求
 * group: 每个工作所需的员工数
 * groupSize: group 数组的大小
 * profit: 每个工作的利润
 * profitSize: profit 数组的大小
 *
 * 返回值：
 * 满足条件的计划数（模 10^9+7）
 */
int profitableSchemesOptimized(int n, int minProfit, int* group, int groupSize, int* profit, int profitSize) {
 // 参数验证
 if (n <= 0) {
 return minProfit <= 0 ? 1 : 0;
 }
 if (groupSize == 0 || profitSize == 0 || groupSize != profitSize) {
 return minProfit <= 0 ? 1 : 0;
 }
}

```

```

int m = groupSize; // 工作数量

// 创建二维 DP 数组: dp[j][k] 表示使用 j 个员工, 获得至少 k 的利润的方案数
// 使用滚动数组优化空间复杂度, 相当于将 dp[i][j][k] 压缩为 dp[j][k]
int dp[MAX_N][MAX_P];

// 初始化 DP 数组
for (int j = 0; j <= n; j++) {
 for (int k = 0; k <= minProfit; k++) {
 dp[j][k] = 0;
 }
}

// 初始状态: 使用 0 个员工, 利润为 0 的方案数为 1
dp[0][0] = 1;

// 遍历每个工作 (物品)
for (int i = 0; i < m; i++) {
 int g = group[i]; // 当前工作所需的员工数
 int p = profit[i]; // 当前工作的利润

 // 逆序遍历员工数, 避免重复计算
 // 倒序遍历确保每个物品只使用一次
 for (int j = n; j >= g; j--) {
 // 逆序遍历利润要求
 for (int k = minProfit; k >= 0; k--) {
 // 计算选择当前工作后所需的最小利润
 int prevProfit = max(0, k - p);
 // 更新方案数, 注意取模
 long long newValue = (long long)dp[j][k] + dp[j - g][prevProfit];
 dp[j][k] = newValue % MOD;
 }
 }
}

// 计算所有可能的方案数
int result = 0;
for (int j = 0; j <= n; j++) {
 result = ((long long)result + dp[j][minProfit]) % MOD;
}

return result;
}

```

```

/*
 * 示例：
 * 输入：n = 5, minProfit = 3, group = [2, 2], profit = [2, 3]
 * 输出：2
 * 解释：至少产生 3 利润的计划有 2 种：完成工作 1 和工作 2，或仅完成工作 2。
 *
 * 输入：n = 10, minProfit = 5, group = [2, 3, 5], profit = [6, 7, 8]
 * 输出：7
 * 解释：至少产生 5 利润的计划有 7 种。
 *
 * 时间复杂度：O(N * minProfit * n)
 * - 外层循环遍历所有工作：O(N)
 * - 中层循环遍历员工数：O(n)
 * - 内层循环遍历利润：O(minProfit)
 * 空间复杂度：O(minProfit * n)
 * - 二维 DP 数组的空间消耗
*/

```

=====

文件：Code18\_ProfitableSchemes.java

```

package class073;

// LeetCode 879. 盈利计划
// 题目描述：集团里有 n 名员工，他们可以完成各种各样的工作创造利润。
// 第 i 种工作会产生 profit[i] 的利润，它要求 group[i] 名成员共同参与。
// 如果成员参与了其中一项工作，就不能参与另一项工作。
// 工作的任何至少产生 minProfit 利润的子集称为 盈利计划 。
// 并且工作的成员总数最多为 n 。
// 有多少种计划可以选择？因为答案很大，所以返回结果模 10^9 + 7 的值。
// 链接：https://leetcode.cn/problems/profitable-schemes/
//
// 解题思路：
// 这是一个三维费用的 01 背包问题。
// 三维分别是：员工人数、利润、可选的工作数量
// 状态定义：dp[i][j][k] 表示考虑前 i 个工作，使用 j 个员工，获得至少 k 的利润的方案数
// 状态转移方程：
// - 不选择第 i 个工作：dp[i][j][k] = dp[i-1][j][k]
// - 选择第 i 个工作：dp[i][j][k] += dp[i-1][j-group[i-1]][Math.max(0, k-profit[i-1])]
//
// 时间复杂度：O(N * minProfit * n)，其中 N 是工作数量，n 是员工人数

```

```

// 空间复杂度: O(N * minProfit * n), 可以优化到 O(minProfit * n)

public class Code18_ProfitableSchemes {
 // 模数
 private static final int MOD = 1000000007;

 // 主方法, 用于测试
 public static void main(String[] args) {
 // 测试用例 1
 int n1 = 5, minProfit1 = 3;
 int[] group1 = {2, 2};
 int[] profit1 = {2, 3};
 System.out.println("测试用例 1 结果: " + profitableSchemes(n1, minProfit1, group1,
profit1)); // 预期输出: 2

 // 测试用例 2
 int n2 = 10, minProfit2 = 5;
 int[] group2 = {2, 3, 5};
 int[] profit2 = {6, 7, 8};
 System.out.println("测试用例 2 结果: " + profitableSchemes(n2, minProfit2, group2,
profit2)); // 预期输出: 7
 }

 /**
 * 计算盈利计划的数量
 *
 * 解题思路:
 * 这是一个三维费用的 01 背包问题。
 * 三维分别是: 员工人数、利润、可选的工作数量
 * 状态定义: dp[i][j][k] 表示考虑前 i 个工作, 使用 j 个员工, 获得至少 k 的利润的方案数
 * 状态转移方程:
 * - 不选择第 i 个工作: dp[i][j][k] = dp[i-1][j][k]
 * - 选择第 i 个工作: dp[i][j][k] += dp[i-1][j-group[i-1]][Math.max(0, k-profit[i-1])]
 *
 * @param n 员工总数
 * @param minProfit 最小利润要求
 * @param group 每个工作所需的员工数
 * @param profit 每个工作的利润
 * @return 满足条件的计划数 (模 10^9+7)
 */
 public static int profitableSchemes(int n, int minProfit, int[] group, int[] profit) {
 // 参数验证
 if (n <= 0) {

```

```

 return minProfit <= 0 ? 1 : 0;
}

if (group == null || profit == null || group.length == 0 || group.length != profit.length) {
 return minProfit <= 0 ? 1 : 0;
}

int m = group.length; // 工作数量

// 创建三维 DP 数组: dp[i][j][k] 表示考虑前 i 个工作, 使用 j 个员工, 获得至少 k 的利润的方案数
// 这是一个三维费用的 01 背包问题
int[][][] dp = new int[m + 1][n + 1][minProfit + 1];

// 初始状态: 没有选择任何工作时, 使用 0 个员工, 利润为 0 的方案数为 1
// 这是动态规划的边界条件
dp[0][0][0] = 1;

// 遍历每个工作 (物品)
for (int i = 1; i <= m; i++) {
 int g = group[i - 1]; // 当前工作所需的员工数 (第一个费用)
 int p = profit[i - 1]; // 当前工作的利润 (第二个费用)

 // 遍历员工数 (第一个费用维度)
 for (int j = 0; j <= n; j++) {
 // 遍历利润要求 (第二个费用维度)
 for (int k = 0; k <= minProfit; k++) {
 // 不选择当前工作的情况
 // 保持前一个状态的方案数
 dp[i][j][k] = dp[i - 1][j][k];

 // 选择当前工作的情况, 需要确保员工数足够
 if (j >= g) {
 // 计算选择当前工作后所需的最小利润
 // 如果 k < p, 则前一个状态需要的利润为 0 (因为负利润没有意义)
 int prevProfit = Math.max(0, k - p);
 // 更新方案数, 注意取模
 // dp[i][j][k] = 不选择当前工作的方案数 + 选择当前工作的方案数
 dp[i][j][k] = (dp[i][j][k] + dp[i - 1][j - g][prevProfit]) % MOD;
 }
 }
 }
}
}

```

```

// 计算所有可能的方案数：员工数不超过 n，利润至少为 minProfit
// 遍历所有可能的员工数，累加方案数
int result = 0;
for (int j = 0; j <= n; j++) {
 result = (result + dp[m][j][minProfit]) % MOD;
}

return result;
}

/**
 * 空间优化版本：使用二维 DP 数组
 *
 * @param n 员工总数
 * @param minProfit 最小利润要求
 * @param group 每个工作所需的员工数
 * @param profit 每个工作的利润
 * @return 满足条件的计划数（模 10^9+7）
 */
public static int profitableSchemesOptimized(int n, int minProfit, int[] group, int[] profit)
{
 // 参数验证
 if (n <= 0) {
 return minProfit <= 0 ? 1 : 0;
 }
 if (group == null || profit == null || group.length == 0 || group.length != profit.length) {
 return minProfit <= 0 ? 1 : 0;
 }

 int m = group.length; // 工作数量

 // 创建二维 DP 数组：dp[j][k] 表示使用 j 个员工，获得至少 k 的利润的方案数
 // 使用滚动数组优化空间复杂度，相当于将 dp[i][j][k] 压缩为 dp[j][k]
 int[][] dp = new int[n + 1][minProfit + 1];

 // 初始状态：使用 0 个员工，利润为 0 的方案数为 1
 dp[0][0] = 1;

 // 遍历每个工作（物品）
 for (int i = 0; i < m; i++) {
 int g = group[i]; // 当前工作所需的员工数
 int p = profit[i]; // 当前工作的利润

```

```

// 逆序遍历员工数，避免重复计算
// 倒序遍历确保每个物品只使用一次
for (int j = n; j >= g; j--) {
 // 逆序遍历利润要求
 for (int k = minProfit; k >= 0; k--) {
 // 计算选择当前工作后所需的最小利润
 int prevProfit = Math.max(0, k - p);
 // 更新方案数，注意取模
 dp[j][k] = (dp[j][k] + dp[j - g][prevProfit]) % MOD;
 }
}
}

// 计算所有可能的方案数
int result = 0;
for (int j = 0; j <= n; j++) {
 result = (result + dp[j][minProfit]) % MOD;
}

return result;
}

/**
 * 进一步优化：针对利润维度进行优化，当利润超过 minProfit 时，可以将其视为等于 minProfit
 *
 * @param n 员工总数
 * @param minProfit 最小利润要求
 * @param group 每个工作所需的员工数
 * @param profit 每个工作的利润
 * @return 满足条件的计划数（模 10^9+7）
 */
public static int profitableSchemesFurtherOptimized(int n, int minProfit, int[] group, int[]
profit) {
 // 参数验证
 if (n <= 0) {
 return minProfit <= 0 ? 1 : 0;
 }
 if (group == null || profit == null || group.length == 0 || group.length != profit.length) {
 return minProfit <= 0 ? 1 : 0;
 }
}

```

```

// 创建二维 DP 数组
int[][] dp = new int[n + 1][minProfit + 1];
dp[0][0] = 1;

// 遍历每个工作
for (int i = 0; i < group.length; i++) {
 int g = group[i];
 int p = profit[i];

 // 剪枝：如果工作所需员工数超过总员工数，跳过
 // 因为当前工作本身就无法完成
 if (g > n) {
 continue;
 }

 // 逆序遍历员工数
 for (int j = n; j >= g; j--) {
 // 计算当前工作的实际利润贡献（不超过 minProfit）
 // 当利润超过 minProfit 时，可以将其视为等于 minProfit
 int actualProfit = Math.min(p, minProfit);

 // 遍历利润要求
 for (int k = minProfit; k >= 0; k--) {
 // 计算选择当前工作后所需的最小利润
 int prevProfit = Math.max(0, k - actualProfit);
 // 更新方案数，注意取模
 dp[j][k] = (dp[j][k] + dp[j - g][prevProfit]) % MOD;
 }
 }
}

// 计算所有可能的方案数
int result = 0;
for (int j = 0; j <= n; j++) {
 result = (result + dp[j][minProfit]) % MOD;
}

return result;
}

/*
 * 示例：
 * 输入：n = 5, minProfit = 3, group = [2, 2], profit = [2, 3]
 */

```

```

* 输出: 2
* 解释: 至少产生 3 利润的计划有 2 种: 完成工作 1 和工作 2, 或仅完成工作 2。
*
* 输入: n = 10, minProfit = 5, group = [2, 3, 5], profit = [6, 7, 8]
* 输出: 7
* 解释: 至少产生 5 利润的计划有 7 种。
*
* 时间复杂度: O(N * minProfit * n)
* - 外层循环遍历所有工作: O(N)
* - 中层循环遍历员工数: O(n)
* - 内层循环遍历利润: O(minProfit)
* 空间复杂度: O(minProfit * n)
* - 二维 DP 数组的空间消耗
*/
}

=====
```

文件: Code18\_ProfitableSchemes.py

```

LeetCode 879. 盈利计划
题目描述: 集团里有 n 名员工, 他们可以完成各种各样的工作创造利润。
第 i 种工作会产生 profit[i] 的利润, 它要求 group[i] 名成员共同参与。
如果成员参与了其中一项工作, 就不能参与另一项工作。
工作的任何至少产生 minProfit 利润的子集称为 盈利计划 。
并且工作的成员总数最多为 n 。
有多少种计划可以选择? 因为答案很大, 所以返回结果模 $10^9 + 7$ 的值。
链接: https://leetcode.cn/problems/profitable-schemes/
#
解题思路:
这是一个三维费用的 01 背包问题。
三维分别是: 员工人数、利润、可选的工作数量
状态定义: dp[i][j][k] 表示考虑前 i 个工作, 使用 j 个员工, 获得至少 k 的利润的方案数
状态转移方程:
- 不选择第 i 个工作: dp[i][j][k] = dp[i-1][j][k]
- 选择第 i 个工作: dp[i][j][k] += dp[i-1][j-group[i-1]][max(0, k-profit[i-1])]
#
时间复杂度: O(N * minProfit * n), 其中 N 是工作数量, n 是员工人数
空间复杂度: O(N * minProfit * n), 可以优化到 O(minProfit * n)
```

MOD =  $10^{**}9 + 7$

```
def profitable_schemes(n, min_profit, group, profit):
```

```
"""
```

计算盈利计划的数量

解题思路：

这是一个三维费用的 01 背包问题。

三维分别是：员工人数、利润、可选的工作数量

状态定义： $dp[i][j][k]$  表示考虑前  $i$  个工作，使用  $j$  个员工，获得至少  $k$  的利润的方案数

状态转移方程：

- 不选择第  $i$  个工作： $dp[i][j][k] = dp[i-1][j][k]$

- 选择第  $i$  个工作： $dp[i][j][k] += dp[i-1][j-group[i-1]][\max(0, k-profit[i-1])]$

Args:

$n$ : 员工总数

$min\_profit$ : 最小利润要求

$group$ : 每个工作所需的员工数

$profit$ : 每个工作的利润

Returns:

$int$ : 满足条件的计划数（模  $10^{9+7}$ ）

```
"""
```

# 参数验证

if  $n \leq 0$ :

return 1 if  $min\_profit \leq 0$  else 0

if not group or not profit or len(group) != len(profit):

return 1 if  $min\_profit \leq 0$  else 0

$m = len(group)$  # 工作数量

# 创建三维 DP 数组： $dp[i][j][k]$  表示考虑前  $i$  个工作，使用  $j$  个员工，获得至少  $k$  的利润的方案数

# 这是一个三维费用的 01 背包问题

$dp = [[[0] * (min_profit + 1) for _ in range(n + 1)] for __ in range(m + 1)]$

# 初始状态：没有选择任何工作时，使用 0 个员工，利润为 0 的方案数为 1

# 这是动态规划的边界条件

$dp[0][0][0] = 1$

# 遍历每个工作（物品）

for  $i$  in range(1,  $m + 1$ ):

$g = group[i - 1]$  # 当前工作所需的员工数（第一个费用）

$p = profit[i - 1]$  # 当前工作的利润（第二个费用）

# 遍历员工数（第一个费用维度）

for  $j$  in range( $n + 1$ ):

```

遍历利润要求 (第二个费用维度)
for k in range(min_profit + 1):
 # 不选择当前工作的情况
 # 保持前一个状态的方案数
 dp[i][j][k] = dp[i - 1][j][k]

 # 选择当前工作的情况, 需要确保员工数足够
 if j >= g:
 # 计算选择当前工作后所需的最小利润
 # 如果 k < p, 则前一个状态需要的利润为 0 (因为负利润没有意义)
 prev_profit = max(0, k - p)
 # 更新方案数, 注意取模
 # dp[i][j][k] = 不选择当前工作的方案数 + 选择当前工作的方案数
 dp[i][j][k] = (dp[i][j][k] + dp[i - 1][j - g][prev_profit]) % MOD

计算所有可能的方案数: 员工数不超过 n, 利润至少为 min_profit
遍历所有可能的员工数, 累加方案数
result = 0
for j in range(n + 1):
 result = (result + dp[m][j][min_profit]) % MOD

return result

```

```

def profitable_schemes_optimized(n, min_profit, group, profit):
 """

```

空间优化版本: 使用二维 DP 数组

Args:

- n: 员工总数
- min\_profit: 最小利润要求
- group: 每个工作所需的员工数
- profit: 每个工作的利润

Returns:

- int: 满足条件的计划数 (模  $10^{9+7}$ )

```

"""

```

# 参数验证

```

if n <= 0:
 return 1 if min_profit <= 0 else 0
if not group or not profit or len(group) != len(profit):
 return 1 if min_profit <= 0 else 0

```

```

m = len(group) # 工作数量

```

```

创建二维 DP 数组: dp[j][k] 表示使用 j 个员工, 获得至少 k 的利润的方案数
使用滚动数组优化空间复杂度, 相当于将 dp[i][j][k] 压缩为 dp[j][k]
dp = [[0] * (min_profit + 1) for _ in range(n + 1)]

初始状态: 使用 0 个员工, 利润为 0 的方案数为 1
dp[0][0] = 1

遍历每个工作 (物品)
for i in range(m):
 g = group[i] # 当前工作所需的员工数
 p = profit[i] # 当前工作的利润

 # 逆序遍历员工数, 避免重复计算
 # 倒序遍历确保每个物品只使用一次
 for j in range(n, g - 1, -1):
 # 逆序遍历利润要求
 for k in range(min_profit, -1, -1):
 # 计算选择当前工作后所需的最小利润
 prev_profit = max(0, k - p)
 # 更新方案数, 注意取模
 dp[j][k] = (dp[j][k] + dp[j - g][prev_profit]) % MOD

计算所有可能的方案数
result = 0
for j in range(n + 1):
 result = (result + dp[j][min_profit]) % MOD

return result

```

```
def profitable_schemes_further_optimized(n, min_profit, group, profit):
 """

```

进一步优化: 针对利润维度进行优化, 当利润超过 `min_profit` 时, 可以将其视为等于 `min_profit`

Args:

- n: 员工总数
- `min_profit`: 最小利润要求
- group: 每个工作所需的员工数
- profit: 每个工作的利润

Returns:

- int: 满足条件的计划数 (模  $10^{9+7}$ )

```
"""

```

```

参数验证
if n <= 0:
 return 1 if min_profit <= 0 else 0
if not group or not profit or len(group) != len(profit):
 return 1 if min_profit <= 0 else 0

创建二维 DP 数组
dp = [[0] * (min_profit + 1) for _ in range(n + 1)]
dp[0][0] = 1

遍历每个工作
for i in range(len(group)):
 g = group[i]
 p = profit[i]

 # 剪枝: 如果工作所需员工数超过总员工数, 跳过
 # 因为当前工作本身就无法完成
 if g > n:
 continue

 # 逆序遍历员工数
 for j in range(n, g - 1, -1):
 # 计算当前工作的实际利润贡献 (不超过 min_profit)
 # 当利润超过 min_profit 时, 可以将其视为等于 min_profit
 actual_profit = min(p, min_profit)

 # 遍历利润要求
 for k in range(min_profit, -1, -1):
 # 计算选择当前工作后所需的最小利润
 prev_profit = max(0, k - actual_profit)
 # 更新方案数, 注意取模
 dp[j][k] = (dp[j][k] + dp[j - g][prev_profit]) % MOD

 # 计算所有可能的方案数
 result = 0
 for j in range(n + 1):
 result = (result + dp[j][min_profit]) % MOD

return result

测试用例
if __name__ == "__main__":
 # 测试用例 1

```

```
n1, min_profit1 = 5, 3
group1 = [2, 2]
profit1 = [2, 3]
print(f"测试用例 1 结果: {profitable_schemes(n1, min_profit1, group1, profit1)}") # 预期输出:
2
```

```
测试用例 2
n2, min_profit2 = 10, 5
group2 = [2, 3, 5]
profit2 = [6, 7, 8]
print(f"测试用例 2 结果: {profitable_schemes(n2, min_profit2, group2, profit2)}") # 预期输出:
7
```

,,

示例:

输入: n = 5, minProfit = 3, group = [2, 2], profit = [2, 3]

输出: 2

解释: 至少产生 3 利润的计划有 2 种: 完成工作 1 和工作 2, 或仅完成工作 2。

输入: n = 10, minProfit = 5, group = [2, 3, 5], profit = [6, 7, 8]

输出: 7

解释: 至少产生 5 利润的计划有 7 种。

时间复杂度:  $O(N * \text{minProfit} * n)$

- 外层循环遍历所有工作:  $O(N)$
- 中层循环遍历员工数:  $O(n)$
- 内层循环遍历利润:  $O(\text{minProfit})$

空间复杂度:  $O(\text{minProfit} * n)$

- 二维 DP 数组的空间消耗

,,

---

文件: Code19\_CoinChange.cpp

---

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <queue>
#include <climits>
```

```
// LeetCode 322. 零钱兑换
```

```
// 题目描述: 给你一个整数数组 coins , 表示不同面额的硬币; 以及一个整数 amount , 表示总金额。
```

```
// 计算并返回可以凑成总金额所需的 最少的硬币个数 。如果没有任何一种硬币组合能组成总金额，返回 -1 。
// 你可以认为每种硬币的数量是无限的。
// 链接: https://leetcode.cn/problems/coin-change/
//
// 解题思路:
// 这是一个典型的完全背包问题，因为每种硬币可以使用无限次。
// 状态定义: dp[i] 表示凑成金额 i 所需的最少硬币个数
// 状态转移方程: dp[i] = min(dp[i], dp[i - coins[j]] + 1)，其中 j 遍历所有硬币
// 初始状态: dp[0] = 0 (凑成金额 0 需要 0 个硬币)，其他初始化为一个较大值 (如 amount+1)
//
// 时间复杂度: O(amount * n)，其中 n 是硬币种类数
// 空间复杂度: O(amount)，使用一维 DP 数组
```

```
using namespace std;
```

```
/***
 * 计算凑成总金额所需的最少硬币个数
 * @param coins 不同面额的硬币数组
 * @param amount 总金额
 * @return 最少硬币个数，如果无法凑成则返回-1
 */
```

```
int coinChange(vector<int>& coins, int amount) {
 // 参数验证
 if (amount == 0) {
 return 0;
 }
 if (coins.empty()) {
 return -1;
 }
```

```
// 创建 DP 数组，dp[i] 表示凑成金额 i 所需的最少硬币个数
vector<int> dp(amount + 1, amount + 1);
```

```
// 基础情况：凑成金额 0 需要 0 个硬币
dp[0] = 0;

// 遍历每种硬币（物品）
for (int coin : coins) {
 // 正序遍历金额（容量），因为完全背包允许重复使用物品
 for (int i = coin; i <= amount; i++) {
 // 状态转移：选择当前硬币或不选择当前硬币
 dp[i] = min(dp[i], dp[i - coin] + 1);
 }
}
```

```
 }

 }

// 如果 dp[amount] 仍为初始值，说明无法凑成
return dp[amount] > amount ? -1 : dp[amount];
}

/***
 * 优化版本：提前剪枝和优化循环范围
 */
int coinChangeOptimized(vector<int>& coins, int amount) {
 // 参数验证和快速返回
 if (amount == 0) {
 return 0;
 }
 if (coins.empty()) {
 return -1;
 }

 // 对硬币进行排序，从小到大
 sort(coins.begin(), coins.end());

 // 创建 DP 数组
 vector<int> dp(amount + 1, amount + 1);
 dp[0] = 0;

 // 遍历金额
 for (int i = 1; i <= amount; i++) {
 // 遍历每种硬币
 for (int coin : coins) {
 // 剪枝：如果当前硬币大于金额 i，直接跳过
 if (coin > i) {
 break;
 }
 // 状态转移
 if (dp[i - coin] != amount + 1) {
 dp[i] = min(dp[i], dp[i - coin] + 1);
 }
 }
 }

 return dp[amount] > amount ? -1 : dp[amount];
}
```

```
/**
 * BFS 优化版本：对于找最少硬币个数的问题，BFS 可能更快找到解
 */

int coinChangeBFS(vector<int>& coins, int amount) {
 // 参数验证
 if (amount == 0) {
 return 0;
 }
 if (coins.empty()) {
 return -1;
 }

 // 对硬币进行排序，有助于提前剪枝
 sort(coins.begin(), coins.end());

 // 使用 BFS，每个节点表示当前的金额和已使用的硬币个数
 // 使用一个布尔数组记录已经访问过的金额，避免重复计算
 vector<bool> visited(amount + 1, false);
 queue<int> q;
 q.push(0);
 visited[0] = true;
 int level = 0; // 当前层数，表示已使用的硬币个数

 while (!q.empty()) {
 int size = q.size();
 level++;

 for (int i = 0; i < size; i++) {
 int current = q.front();
 q.pop();

 // 尝试每种硬币
 for (int coin : coins) {
 int next = current + coin;

 // 如果找到目标金额，返回当前层数
 if (next == amount) {
 return level;
 }

 // 剪枝：如果超过目标金额或已经访问过，跳过
 if (next > amount || visited[next]) {
```

```

 continue;
 }

 visited[next] = true;
 q.push(next);
}

}

// 无法凑成目标金额
return -1;
}

/***
 * 贪心+DFS 优化版本：对于某些情况（如硬币是倍数关系时）效率更高
 */
int coinChangeGreedyDFS(vector<int>& coins, int amount) {
 // 参数验证
 if (amount == 0) {
 return 0;
 }
 if (coins.empty()) {
 return -1;
 }

 // 对硬币进行排序，从大到小
 sort(coins.begin(), coins.end(), greater<int>());

 // 记录最小硬币个数
 int minCount = INT_MAX;

 // DFS 搜索
 function<void(int, int, int)> dfs = [&](int index, int remaining, int count) {
 // 已经找到一个解，或者当前硬币个数已经超过已知的最小硬币个数，直接返回
 if (remaining == 0) {
 minCount = min(minCount, count);
 return;
 }

 if (index == coins.size() || count >= minCount - 1) {
 return;
 }

 for (int i = index; i < coins.size(); ++i) {
 if (coins[i] > remaining) {
 break;
 }
 dfs(i + 1, remaining - coins[i], count + 1);
 }
 };
 dfs(0, amount, 0);
 return minCount;
}

```

```

// 贪心策略：尽可能多地使用当前面值的硬币
int maxUse = remaining / coins[index];
for (int i = maxUse; i >= 0; i--) {
 int newRemaining = remaining - i * coins[index];
 int newCount = count + i;

 // 剪枝：如果剩余金额为 0 或者当前硬币个数加上剩余金额的最小可能个数
 // 仍然小于已知的最小硬币个数，才继续搜索
 if (newRemaining == 0 || newCount + 1 < minCount) {
 dfs(index + 1, newRemaining, newCount);
 }
}
};

dfs(0, amount, 0);

return minCount == INT_MAX ? -1 : minCount;
}

int main() {
// 测试用例 1
vector<int> coins1 = {1, 2, 5};
int amount1 = 11;
cout << "测试用例 1 结果：" << coinChange(coins1, amount1) << endl; // 预期输出: 3 (5+5+1)

// 测试用例 2
vector<int> coins2 = {2};
int amount2 = 3;
cout << "测试用例 2 结果：" << coinChange(coins2, amount2) << endl; // 预期输出: -1

// 测试用例 3
vector<int> coins3 = {1};
int amount3 = 0;
cout << "测试用例 3 结果：" << coinChange(coins3, amount3) << endl; // 预期输出: 0

// 测试用例 4
vector<int> coins4 = {1, 2, 5, 10, 20, 50, 100};
int amount4 = 489;
cout << "测试用例 4 结果：" << coinChange(coins4, amount4) << endl; // 预期输出: 9

return 0;
}

```

文件: Code19\_CoinChange.java

```
=====
package class073;
```

```
import java.util.Arrays;
```

```
// LeetCode 322. 零钱兑换
```

```
// 题目描述: 给你一个整数数组 coins , 表示不同面额的硬币; 以及一个整数 amount , 表示总金额。
// 计算并返回可以凑成总金额所需的 最少的硬币个数 。如果没有任何一种硬币组合能组成总金额, 返回 -1 。
```

```
// 你可以认为每种硬币的数量是无限的。
```

```
// 链接: https://leetcode.cn/problems/coin-change/
```

```
///
```

```
// 解题思路:
```

```
// 这是一个典型的完全背包问题, 因为每种硬币可以使用无限次。
```

```
// 状态定义: dp[i] 表示凑成金额 i 所需的最少硬币个数
```

```
// 状态转移方程: dp[i] = min(dp[i], dp[i - coins[j]] + 1), 其中 j 遍历所有硬币
```

```
// 初始状态: dp[0] = 0 (凑成金额 0 需要 0 个硬币), 其他初始化为一个较大值 (如 amount+1)
```

```
///
```

```
// 时间复杂度: O(amount * n), 其中 n 是硬币种类数
```

```
// 空间复杂度: O(amount), 使用一维 DP 数组
```

```
public class Code19_CoinChange {
```

```
 // 主方法, 用于测试
```

```
 public static void main(String[] args) {
```

```
 // 测试用例 1
```

```
 int[] coins1 = {1, 2, 5};
```

```
 int amount1 = 11;
```

```
 System.out.println("测试用例 1 结果: " + coinChange(coins1, amount1)); // 预期输出: 3
```

```
(5+5+1)
```

```
 // 测试用例 2
```

```
 int[] coins2 = {2};
```

```
 int amount2 = 3;
```

```
 System.out.println("测试用例 2 结果: " + coinChange(coins2, amount2)); // 预期输出: -1
```

```
 // 测试用例 3
```

```
 int[] coins3 = {1};
```

```
 int amount3 = 0;
```

```
 System.out.println("测试用例 3 结果: " + coinChange(coins3, amount3)); // 预期输出: 0
```

```

// 测试用例 4
int[] coins4 = {1, 2, 5, 10, 20, 50, 100};
int amount4 = 489;
System.out.println("测试用例 4 结果: " + coinChange(coins4, amount4)); // 预期输出: 9
(200+200+50+20+10+5+2+1+1)
}

/**
 * 计算凑成总金额所需的最少硬币个数
 * @param coins 不同面额的硬币数组
 * @param amount 总金额
 * @return 最少硬币个数, 如果无法凑成则返回-1
 */
public static int coinChange(int[] coins, int amount) {
 // 参数验证
 if (amount == 0) {
 return 0;
 }
 if (coins == null || coins.length == 0) {
 return -1;
 }

 // 创建 DP 数组, dp[i] 表示凑成金额 i 所需的最少硬币个数
 int[] dp = new int[amount + 1];

 // 初始化 DP 数组, 设置为一个较大的值 (amount+1 一定大于可能的硬币个数)
 Arrays.fill(dp, amount + 1);
 // 基础情况: 凑成金额 0 需要 0 个硬币
 dp[0] = 0;

 // 遍历每种硬币 (物品)
 for (int coin : coins) {
 // 正序遍历金额 (容量), 因为完全背包允许重复使用物品
 for (int i = coin; i <= amount; i++) {
 // 状态转移: 选择当前硬币或不选择当前硬币
 dp[i] = Math.min(dp[i], dp[i - coin] + 1);
 }
 }

 // 如果 dp[amount] 仍为初始值, 说明无法凑成
 return dp[amount] > amount ? -1 : dp[amount];
}

```

```
/**
 * 优化版本：提前剪枝和优化循环范围
 */
public static int coinChangeOptimized(int[] coins, int amount) {
 // 参数验证和快速返回
 if (amount == 0) {
 return 0;
 }
 if (coins == null || coins.length == 0) {
 return -1;
 }

 // 对硬币进行排序，从小到大
 Arrays.sort(coins);

 // 创建 DP 数组
 int[] dp = new int[amount + 1];
 Arrays.fill(dp, amount + 1);
 dp[0] = 0;

 // 遍历金额
 for (int i = 1; i <= amount; i++) {
 // 遍历每种硬币
 for (int coin : coins) {
 // 剪枝：如果当前硬币大于金额 i，直接跳过
 if (coin > i) {
 break;
 }
 // 状态转移
 if (dp[i - coin] != amount + 1) {
 dp[i] = Math.min(dp[i], dp[i - coin] + 1);
 }
 }
 }

 return dp[amount] > amount ? -1 : dp[amount];
}

/**
 * BFS 优化版本：对于找最少硬币个数的问题，BFS 可能更快找到解
 */
public static int coinChangeBFS(int[] coins, int amount) {
```

```
// 参数验证
if (amount == 0) {
 return 0;
}
if (coins == null || coins.length == 0) {
 return -1;
}

// 对硬币进行排序，有助于提前剪枝
Arrays.sort(coins);

// 使用 BFS，每个节点表示当前的金额和已使用的硬币个数
// 使用一个布尔数组记录已经访问过的金额，避免重复计算
boolean[] visited = new boolean[amount + 1];
java.util.Queue<Integer> queue = new java.util.LinkedList<>();
queue.offer(0);
visited[0] = true;
int level = 0; // 当前层数，表示已使用的硬币个数

while (!queue.isEmpty()) {
 int size = queue.size();
 level++;

 for (int i = 0; i < size; i++) {
 int current = queue.poll();

 // 尝试每种硬币
 for (int coin : coins) {
 int next = current + coin;

 // 如果找到目标金额，返回当前层数
 if (next == amount) {
 return level;
 }

 // 剪枝：如果超过目标金额或已经访问过，跳过
 if (next > amount || visited[next]) {
 continue;
 }

 visited[next] = true;
 queue.offer(next);
 }
 }
}
```

```

 }
 }

 // 无法凑成目标金额
 return -1;
}

/***
 * 贪心+DFS 优化版本：对于某些情况（如硬币是倍数关系时）效率更高
 */
public static int coinChangeGreedyDFS(int[] coins, int amount) {
 // 参数验证
 if (amount == 0) {
 return 0;
 }
 if (coins == null || coins.length == 0) {
 return -1;
 }

 // 对硬币进行排序，从大到小
 Arrays.sort(coins);
 reverse(coins);

 // 记录最小硬币个数
 int[] result = {Integer.MAX_VALUE};

 // DFS 搜索
 dfs(coins, 0, amount, 0, result);

 return result[0] == Integer.MAX_VALUE ? -1 : result[0];
}

/***
 * DFS 辅助函数
 * @param coins 硬币数组
 * @param index 当前尝试的硬币索引
 * @param amount 剩余金额
 * @param count 当前已使用的硬币个数
 * @param result 存储最小硬币个数的数组
 */
private static void dfs(int[] coins, int index, int amount, int count, int[] result) {
 // 已经找到一个解，或者当前硬币个数已经超过已知的最小硬币个数，直接返回
 if (amount == 0) {

```

```

 result[0] = Math.min(result[0], count);
 return;
 }

 if (index == coins.length || count >= result[0] - 1) {
 return;
 }

 // 贪心策略：尽可能多地使用当前面值的硬币
 int maxUse = amount / coins[index];
 for (int i = maxUse; i >= 0; i--) {
 int remaining = amount - i * coins[index];
 int newCount = count + i;

 // 剪枝：如果剩余金额为 0 或者当前硬币个数加上剩余金额的最小可能个数（每个硬币面值为
 1)
 // 仍然小于已知的最小硬币个数，才继续搜索
 if (remaining == 0 || newCount + 1 < result[0]) {
 dfs(coins, index + 1, remaining, newCount, result);
 }
 }
}

/**
 * 反转数组
 */
private static void reverse(int[] array) {
 int left = 0;
 int right = array.length - 1;
 while (left < right) {
 int temp = array[left];
 array[left] = array[right];
 array[right] = temp;
 left++;
 right--;
 }
}
}
=====

文件: Code19_CoinChange.py
=====
```

```
LeetCode 322. 零钱兑换
题目描述：给你一个整数数组 coins，表示不同面额的硬币；以及一个整数 amount，表示总金额。
计算并返回可以凑成总金额所需的 最少的硬币个数。如果没有任何一种硬币组合能组成总金额，返回 -1。
你可以认为每种硬币的数量是无限的。
链接: https://leetcode.cn/problems/coin-change/
#
解题思路：
这是一个典型的完全背包问题，因为每种硬币可以使用无限次。
状态定义：dp[i] 表示凑成金额 i 所需的最少硬币个数
状态转移方程：dp[i] = min(dp[i], dp[i - coins[j]] + 1)，其中 j 遍历所有硬币
初始状态：dp[0] = 0 (凑成金额 0 需要 0 个硬币)，其他初始化为一个较大值（如 amount+1）
#
时间复杂度：O(amount * n)，其中 n 是硬币种类数
空间复杂度：O(amount)，使用一维 DP 数组
```

```
def coin_change(coins, amount):
```

```
 """

```

```
 计算凑成总金额所需的最少硬币个数

```

```
Args:
```

```
 coins: 不同面额的硬币数组

```

```
 amount: 总金额

```

```
Returns:

```

```
 int: 最少硬币个数，如果无法凑成则返回-1

```

```
 """

```

```
参数验证

```

```
if amount == 0:

```

```
 return 0

```

```
if not coins:

```

```
 return -1

```

```
创建 DP 数组，dp[i] 表示凑成金额 i 所需的最少硬币个数

```

```
dp = [amount + 1] * (amount + 1)

```

```
基础情况：凑成金额 0 需要 0 个硬币

```

```
dp[0] = 0

```

```
遍历每种硬币（物品）

```

```
for coin in coins:

```

```
 # 正序遍历金额（容量），因为完全背包允许重复使用物品

```

```
 for i in range(coin, amount + 1):

```

```
状态转移：选择当前硬币或不选择当前硬币
dp[i] = min(dp[i], dp[i - coin] + 1)
```

```
如果 dp[amount] 仍为初始值，说明无法凑成
return dp[amount] if dp[amount] <= amount else -1
```

```
def coin_change_optimized(coins, amount):
```

```
 """
```

```
 优化版本：提前剪枝和优化循环范围
```

```
Args:
```

```
 coins: 不同面额的硬币数组
```

```
 amount: 总金额
```

```
Returns:
```

```
 int: 最少硬币个数，如果无法凑成则返回-1
```

```
 """
```

```
参数验证和快速返回
```

```
if amount == 0:
```

```
 return 0
```

```
if not coins:
```

```
 return -1
```

```
对硬币进行排序，从小到大
```

```
coins.sort()
```

```
创建 DP 数组
```

```
dp = [amount + 1] * (amount + 1)
```

```
dp[0] = 0
```

```
遍历金额
```

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

```
 # 遍历每种硬币
```

```
 for coin in coins:
```

```
 # 剪枝：如果当前硬币大于金额 i，直接跳过
```

```
 if coin > i:
```

```
 break
```

```
 # 状态转移
```

```
 if dp[i - coin] != amount + 1:
```

```
 dp[i] = min(dp[i], dp[i - coin] + 1)
```

```
return dp[amount] if dp[amount] <= amount else -1
```

```
def coin_change_bfs(coins, amount):
 """
 BFS 优化版本：对于找最少硬币个数的问题，BFS 可能更快找到解
 """

 Args:
 coins: 不同面额的硬币数组
 amount: 总金额

 Returns:
 int: 最少硬币个数，如果无法凑成则返回-1
 """

 # 参数验证
 if amount == 0:
 return 0
 if not coins:
 return -1

 # 对硬币进行排序，有助于提前剪枝
 coins.sort()

 # 使用 BFS，每个节点表示当前的金额和已使用的硬币个数
 # 使用一个集合记录已经访问过的金额，避免重复计算
 visited = set()
 from collections import deque
 queue = deque([0])
 visited.add(0)
 level = 0 # 当前层数，表示已使用的硬币个数

 while queue:
 size = len(queue)
 level += 1

 for _ in range(size):
 current = queue.popleft()

 # 尝试每种硬币
 for coin in coins:
 next_amount = current + coin

 # 如果找到目标金额，返回当前层数
 if next_amount == amount:
 return level

 return -1
```

```
剪枝：如果超过目标金额或已经访问过，跳过
if next_amount > amount or next_amount in visited:
 continue

visited.add(next_amount)
queue.append(next_amount)

无法凑成目标金额
return -1
```

```
def coin_change_greedy_dfs(coins, amount):
 """
```

贪心+DFS 优化版本：对于某些情况（如硬币是倍数关系时）效率更高

Args:

coins: 不同面额的硬币数组  
amount: 总金额

Returns:

int: 最少硬币个数，如果无法凑成则返回-1

"""

# 参数验证

```
if amount == 0:
 return 0
if not coins:
 return -1
```

# 对硬币进行排序，从大到小

```
coins.sort(reverse=True)
```

# 记录最小硬币个数

```
min_count = float('inf')
```

```
def dfs(index, remaining, count):
```

```
 nonlocal min_count
```

# 已经找到一个解，或者当前硬币个数已经超过已知的最小硬币个数，直接返回

```
 if remaining == 0:
 min_count = min(min_count, count)
 return
```

```
 if index == len(coins) or count >= min_count - 1:
 return
```

```

贪心策略：尽可能多地使用当前面值的硬币
max_use = remaining // coins[index]
for i in range(max_use, -1, -1):
 new_remaining = remaining - i * coins[index]
 new_count = count + i

 # 剪枝：如果剩余金额为 0 或者当前硬币个数加上剩余金额的最小可能个数
 # 仍然小于已知的最小硬币个数，才继续搜索
 if new_remaining == 0 or new_count + 1 < min_count:
 dfs(index + 1, new_remaining, new_count)

dfs(0, amount, 0)

return min_count if min_count != float('inf') else -1

测试用例
if __name__ == "__main__":
 # 测试用例 1
 coins1 = [1, 2, 5]
 amount1 = 11
 print(f"测试用例 1 结果: {coin_change(coins1, amount1)}") # 预期输出: 3 (5+5+1)

 # 测试用例 2
 coins2 = [2]
 amount2 = 3
 print(f"测试用例 2 结果: {coin_change(coins2, amount2)}") # 预期输出: -1

 # 测试用例 3
 coins3 = [1]
 amount3 = 0
 print(f"测试用例 3 结果: {coin_change(coins3, amount3)}") # 预期输出: 0

 # 测试用例 4
 coins4 = [1, 2, 5, 10, 20, 50, 100]
 amount4 = 489
 print(f"测试用例 4 结果: {coin_change(coins4, amount4)}") # 预期输出: 9

```

---

文件: Code20\_CoinChange2.cpp

---

```
#include <iostream>
```

```

#include <vector>
#include <algorithm>

// LeetCode 518. 零钱兑换 II
// 题目描述：给你一个整数数组 coins 表示不同面额的硬币，另给一个整数 amount 表示总金额。
// 请你计算并返回可以凑成总金额的硬币组合数。如果没有任何一种硬币组合能组成总金额，返回 0 。
// 假设每一种面额的硬币有无限个。
// 链接: https://leetcode.cn/problems/coin-change-ii/
//
// 解题思路：
// 这是一个典型的完全背包问题的计数变种，因为每种硬币可以使用无限次，且我们需要计算组合数而非最大值。
// 状态定义：dp[i] 表示凑成金额 i 的硬币组合数
// 状态转移方程：dp[i] += dp[i - coins[j]]，其中 j 遍历所有硬币，且 i >= coins[j]
// 初始状态：dp[0] = 1 (表示凑成金额 0 有一种方式，即不选任何硬币)
//
// 注意事项：
// 1. 为了确保计算的是组合数而非排列数，我们需要先遍历硬币，再遍历金额
// 2. 这样可以保证每个硬币只使用一次（在组合中的顺序无关）
//
// 时间复杂度：O(amount * n)，其中 n 是硬币种类数
// 空间复杂度：O(amount)，使用一维 DP 数组

using namespace std;

/**
 * 计算凑成总金额的硬币组合数
 * @param amount 总金额
 * @param coins 不同面额的硬币数组
 * @return 硬币组合数
 */
int change(int amount, vector<int>& coins) {
 // 参数验证
 if (amount == 0) {
 return 1; // 凑成金额 0 有一种方式，即不选任何硬币
 }
 if (coins.empty()) {
 return 0;
 }

 // 创建 DP 数组，dp[i] 表示凑成金额 i 的硬币组合数
 vector<int> dp(amount + 1, 0);

```

```

// 初始状态: 求成金额 0 有一种方式
dp[0] = 1;

// 遍历每种硬币
for (int coin : coins) {
 // 正序遍历金额, 因为完全背包允许重复使用物品
 // 先遍历硬币再遍历金额, 确保计算的是组合数而非排列数
 for (int i = coin; i <= amount; i++) {
 // 状态转移: 加上使用当前硬币的组合数
 dp[i] += dp[i - coin];
 }
}

return dp[amount];
}

```

```

/**
 * 优化版本: 添加一些剪枝和提前终止的条件
 */
int changeOptimized(int amount, vector<int>& coins) {
 // 参数验证
 if (amount == 0) {
 return 1;
 }
 if (coins.empty()) {
 return 0;
 }

 // 排序有助于提前剪枝
 sort(coins.begin(), coins.end());

 // 创建 DP 数组
 vector<int> dp(amount + 1, 0);
 dp[0] = 1;

 // 遍历每种硬币
 for (int coin : coins) {
 // 如果当前硬币面值已经大于目标金额, 可以跳过
 if (coin > amount) {
 continue;
 }

 // 正序遍历金额

```

```

 for (int i = coin; i <= amount; i++) {
 dp[i] += dp[i - coin];
 }
 }

 return dp[amount];
}

/***
 * 二维 DP 实现，虽然空间复杂度更高，但更容易理解
 */
int change2D(int amount, vector<int>& coins) {
 // 参数验证
 if (amount == 0) {
 return 1;
 }
 if (coins.empty()) {
 return 0;
 }

 int n = coins.size();
 // dp[i][j] 表示使用前 i 种硬币凑成金额 j 的组合数
 vector<vector<int>> dp(n + 1, vector<int>(amount + 1, 0));

 // 初始化：不使用任何硬币，只能凑成金额 0
 dp[0][0] = 1;

 // 遍历每种硬币
 for (int i = 1; i <= n; i++) {
 int coin = coins[i - 1];
 // 遍历每种金额
 for (int j = 0; j <= amount; j++) {
 // 不使用当前硬币的情况
 dp[i][j] = dp[i - 1][j];

 // 使用当前硬币的情况（可以使用多次）
 if (j >= coin) {
 dp[i][j] += dp[i][j - coin];
 }
 }
 }

 return dp[n][amount];
}

```

```

}

/***
 * 计算排列数的版本（与本题要求不同，仅作对比）
 * 注意：先遍历金额再遍历硬币，这样会计算排列数
 */
int permutationChange(int amount, vector<int>& coins) {
 // 参数验证
 if (amount == 0) {
 return 1;
 }
 if (coins.empty()) {
 return 0;
 }

 // 创建 DP 数组
 vector<int> dp(amount + 1, 0);
 dp[0] = 1;

 // 先遍历金额再遍历硬币，计算的是排列数
 for (int i = 1; i <= amount; i++) {
 for (int coin : coins) {
 if (i >= coin) {
 dp[i] += dp[i - coin];
 }
 }
 }

 return dp[amount];
}

/***
 * 递归+记忆化搜索实现
 */
int changeDFS(int amount, vector<int>& coins) {
 // 参数验证
 if (amount == 0) {
 return 1;
 }
 if (coins.empty()) {
 return 0;
 }
}

```

```
// 排序有助于剪枝
sort(coins.begin(), coins.end());

// 使用记忆化搜索
vector<vector<int>> memo(coins.size(), vector<int>(amount + 1, -1));

// 定义 DFS 函数
function<int(int, int)> dfs = [&](int index, int remaining) -> int {
 // 基础情况：凑成金额 0，找到一种方式
 if (remaining == 0) {
 return 1;
 }

 // 基础情况：无法凑成
 if (index == coins.size() || remaining < 0) {
 return 0;
 }

 // 如果已经计算过，直接返回结果
 if (memo[index][remaining] != -1) {
 return memo[index][remaining];
 }

 // 不使用当前硬币的情况
 int notUse = dfs(index + 1, remaining);

 // 使用当前硬币的情况（如果可以使用）
 int use = 0;
 if (remaining >= coins[index]) {
 // 注意这里 index 不变，表示可以重复使用当前硬币
 use = dfs(index, remaining - coins[index]);
 }

 // 计算结果并记忆化
 memo[index][remaining] = notUse + use;
 return memo[index][remaining];
};

return dfs(0, amount);
}

int main() {
 // 测试用例 1
```

```

vector<int> coins1 = {1, 2, 5};
int amount1 = 5;
cout << "测试用例 1 结果: " << change(amount1, coins1) << endl; // 预期输出: 4

// 测试用例 2
vector<int> coins2 = {2};
int amount2 = 3;
cout << "测试用例 2 结果: " << change(amount2, coins2) << endl; // 预期输出: 0

// 测试用例 3
vector<int> coins3 = {10};
int amount3 = 10;
cout << "测试用例 3 结果: " << change(amount3, coins3) << endl; // 预期输出: 1

// 测试用例 4
vector<int> coins4 = {1, 2, 5, 10};
int amount4 = 10;
cout << "测试用例 4 结果: " << change(amount4, coins4) << endl; // 预期输出: 11

return 0;
}

```

=====

文件: Code20\_CoinChange2.java

=====

```

package class073;

// LeetCode 518. 零钱兑换 II
// 题目描述: 给你一个整数数组 coins 表示不同面额的硬币，另给一个整数 amount 表示总金额。
// 请你计算并返回可以凑成总金额的硬币组合数。如果没有任何一种硬币组合能组成总金额，返回 0 。
// 假设每一种面额的硬币有无限个。
// 链接: https://leetcode.cn/problems/coin-change-ii/
//
// 解题思路:
// 这是一个典型的完全背包问题的计数变种，因为每种硬币可以使用无限次，且我们需要计算组合数而非最大值。
// 状态定义: dp[i] 表示凑成金额 i 的硬币组合数
// 状态转移方程: dp[i] += dp[i - coins[j]]，其中 j 遍历所有硬币，且 i >= coins[j]
// 初始状态: dp[0] = 1 (表示凑成金额 0 有一种方式，即不选任何硬币)
//
// 注意事项:
// 1. 为了确保计算的是组合数而非排列数，我们需要先遍历硬币，再遍历金额

```

```

// 2. 这样可以保证每个硬币只使用一次（在组合中的顺序无关）
//
// 时间复杂度: O(amount * n)，其中 n 是硬币种类数
// 空间复杂度: O(amount)，使用一维 DP 数组

public class Code20_CoinChange2 {

 // 主方法，用于测试
 public static void main(String[] args) {
 // 测试用例 1
 int[] coins1 = {1, 2, 5};
 int amount1 = 5;
 System.out.println("测试用例 1 结果: " + change(amount1, coins1)); // 预期输出: 4
 // 解释：有四种方式可以凑成总金额：
 // 5=5
 // 5=2+2+1
 // 5=2+1+1+1
 // 5=1+1+1+1+1

 // 测试用例 2
 int[] coins2 = {2};
 int amount2 = 3;
 System.out.println("测试用例 2 结果: " + change(amount2, coins2)); // 预期输出: 0

 // 测试用例 3
 int[] coins3 = {10};
 int amount3 = 10;
 System.out.println("测试用例 3 结果: " + change(amount3, coins3)); // 预期输出: 1

 // 测试用例 4
 int[] coins4 = {1, 2, 5, 10};
 int amount4 = 10;
 System.out.println("测试用例 4 结果: " + change(amount4, coins4)); // 预期输出: 11
 }

 /**
 * 计算凑成总金额的硬币组合数
 * @param amount 总金额
 * @param coins 不同面额的硬币数组
 * @return 硬币组合数
 */
 public static int change(int amount, int[] coins) {
 // 参数验证

```

```

if (amount == 0) {
 return 1; // 溉成金额 0 有一种方式，即不选任何硬币
}
if (coins == null || coins.length == 0) {
 return 0;
}

// 创建 DP 数组，dp[i] 表示湉成金额 i 的硬币组合数
int[] dp = new int[amount + 1];

// 初始状态：湉成金额 0 有一种方式
dp[0] = 1;

// 遍历每种硬币
for (int coin : coins) {
 // 正序遍历金额，因为完全背包允许重复使用物品
 // 先遍历硬币再遍历金额，确保计算的是组合数而非排列数
 for (int i = coin; i <= amount; i++) {
 // 状态转移：加上使用当前硬币的组合数
 dp[i] += dp[i - coin];
 }
}

return dp[amount];
}

/***
 * 优化版本：添加一些剪枝和提前终止的条件
 */
public static int changeOptimized(int amount, int[] coins) {
 // 参数验证
 if (amount == 0) {
 return 1;
 }
 if (coins == null || coins.length == 0) {
 return 0;
 }

 // 排序有助于提前剪枝
 java.util.Arrays.sort(coins);

 // 创建 DP 数组
 int[] dp = new int[amount + 1];

```

```

dp[0] = 1;

// 遍历每种硬币
for (int coin : coins) {
 // 如果当前硬币面值已经大于目标金额，可以跳过
 if (coin > amount) {
 continue;
 }

 // 正序遍历金额
 for (int i = coin; i <= amount; i++) {
 dp[i] += dp[i - coin];
 }
}

return dp[amount];
}

/***
 * 二维 DP 实现，虽然空间复杂度更高，但更容易理解
 */
public static int change2D(int amount, int[] coins) {
 // 参数验证
 if (amount == 0) {
 return 1;
 }
 if (coins == null || coins.length == 0) {
 return 0;
 }

 int n = coins.length;
 // dp[i][j] 表示使用前 i 种硬币凑成金额 j 的组合数
 int[][] dp = new int[n + 1][amount + 1];

 // 初始化：不使用任何硬币，只能凑成金额 0
 dp[0][0] = 1;

 // 遍历每种硬币
 for (int i = 1; i <= n; i++) {
 int coin = coins[i - 1];
 // 遍历每种金额
 for (int j = 0; j <= amount; j++) {
 // 不使用当前硬币的情况

```

```

 dp[i][j] = dp[i - 1][j];

 // 使用当前硬币的情况（可以使用多次）
 if (j >= coin) {
 dp[i][j] += dp[i][j - coin];
 }
 }

 return dp[n][amount];
}

/***
 * 计算排列数的版本（与本题要求不同，仅作对比）
 * 注意：先遍历金额再遍历硬币，这样会计算排列数
 */
public static int permutationChange(int amount, int[] coins) {
 // 参数验证
 if (amount == 0) {
 return 1;
 }
 if (coins == null || coins.length == 0) {
 return 0;
 }

 // 创建 DP 数组
 int[] dp = new int[amount + 1];
 dp[0] = 1;

 // 先遍历金额再遍历硬币，计算的是排列数
 for (int i = 1; i <= amount; i++) {
 for (int coin : coins) {
 if (i >= coin) {
 dp[i] += dp[i - coin];
 }
 }
 }

 return dp[amount];
}

/***
 * 递归+记忆化搜索实现

```

```
/*
public static int changeDFS(int amount, int[] coins) {
 // 参数验证
 if (amount == 0) {
 return 1;
 }
 if (coins == null || coins.length == 0) {
 return 0;
 }

 // 排序有助于剪枝
 java.util.Arrays.sort(coins);

 // 使用记忆化搜索
 int[][] memo = new int[coins.length][amount + 1];
 // 初始化 memo 为-1，表示未计算过
 for (int[] row : memo) {
 java.util.Arrays.fill(row, -1);
 }

 return dfs(coins, 0, amount, memo);
}

/**
 * 递归辅助函数
 * @param coins 硬币数组
 * @param index 当前考虑的硬币索引
 * @param amount 剩余金额
 * @param memo 记忆化数组
 * @return 组合数
*/
private static int dfs(int[] coins, int index, int amount, int[][] memo) {
 // 基础情况：凑成金额 0，找到一种方式
 if (amount == 0) {
 return 1;
 }

 // 基础情况：无法凑成
 if (index == coins.length || amount < 0) {
 return 0;
 }

 // 如果已经计算过，直接返回结果
 if (memo[index][amount] != -1) {
 return memo[index][amount];
 }

 int count = 0;
 for (int i = index; i < coins.length; i++) {
 if (coins[i] > amount) {
 break;
 }
 count += dfs(coins, i + 1, amount - coins[i], memo);
 }

 memo[index][amount] = count;
 return count;
}
```

```

 if (memo[index][amount] != -1) {
 return memo[index][amount];
 }

 // 不使用当前硬币的情况
 int notUse = dfs(coins, index + 1, amount, memo);

 // 使用当前硬币的情况（如果可以使用）
 int use = 0;
 if (amount >= coins[index]) {
 // 注意这里 index 不变，表示可以重复使用当前硬币
 use = dfs(coins, index, amount - coins[index], memo);
 }

 // 计算结果并记忆化
 memo[index][amount] = notUse + use;
 return memo[index][amount];
}

```

=====

文件: Code20\_CoinChange2.py

=====

```

LeetCode 518. 零钱兑换 II
题目描述: 给你一个整数数组 coins 表示不同面额的硬币，另给一个整数 amount 表示总金额。
请你计算并返回可以凑成总金额的硬币组合数。如果没有任何一种硬币组合能组成总金额，返回 0 。
假设每一种面额的硬币有无限个。
链接: https://leetcode.cn/problems/coin-change-ii/
#
解题思路:
这是一个典型的完全背包问题的计数变种，因为每种硬币可以使用无限次，且我们需要计算组合数而非最大值。
状态定义: dp[i] 表示凑成金额 i 的硬币组合数
状态转移方程: dp[i] += dp[i - coins[j]]，其中 j 遍历所有硬币，且 i >= coins[j]
初始状态: dp[0] = 1 (表示凑成金额 0 有一种方式，即不选任何硬币)
#
注意事项:
1. 为了确保计算的是组合数而非排列数，我们需要先遍历硬币，再遍历金额
2. 这样可以保证每个硬币只使用一次（在组合中的顺序无关）
#
时间复杂度: O(amount * n)，其中 n 是硬币种类数
空间复杂度: O(amount)，使用一维 DP 数组

```

```
def change(amount, coins):
 """
 计算凑成总金额的硬币组合数

 Args:
 amount: 总金额
 coins: 不同面额的硬币数组

 Returns:
 int: 硬币组合数
 """

 # 参数验证
 if amount == 0:
 return 1 # 凑成金额 0 有一种方式, 即不选任何硬币
 if not coins:
 return 0

 # 创建 DP 数组, dp[i] 表示凑成金额 i 的硬币组合数
 dp = [0] * (amount + 1)

 # 初始状态: 凑成金额 0 有一种方式
 dp[0] = 1

 # 遍历每种硬币
 for coin in coins:
 # 正序遍历金额, 因为完全背包允许重复使用物品
 # 先遍历硬币再遍历金额, 确保计算的是组合数而非排列数
 for i in range(coin, amount + 1):
 # 状态转移: 加上使用当前硬币的组合数
 dp[i] += dp[i - coin]

 return dp[amount]
```

```
def change_optimized(amount, coins):
 """
 优化版本: 添加一些剪枝和提前终止的条件

 Args:
```

amount: 总金额  
 coins: 不同面额的硬币数组

Returns:

```
int: 硬币组合数
"""
参数验证
if amount == 0:
 return 1
if not coins:
 return 0

排序有助于提前剪枝
coins.sort()

创建 DP 数组
dp = [0] * (amount + 1)
dp[0] = 1

遍历每种硬币
for coin in coins:
 # 如果当前硬币面值已经大于目标金额，可以跳过
 if coin > amount:
 continue

 # 正序遍历金额
 for i in range(coin, amount + 1):
 dp[i] += dp[i - coin]

return dp[amount]
```

```
def change_2d(amount, coins):
```

```
"""
二维 DP 实现，虽然空间复杂度更高，但更容易理解
```

Args:

```
 amount: 总金额
 coins: 不同面额的硬币数组
```

Returns:

```
 int: 硬币组合数
"""

参数验证
if amount == 0:
 return 1
if not coins:
 return 0
```

```

n = len(coins)
dp[i][j] 表示使用前 i 种硬币凑成金额 j 的组合数
dp = [[0] * (amount + 1) for _ in range(n + 1)]

初始化：不使用任何硬币，只能凑成金额 0
dp[0][0] = 1

遍历每种硬币
for i in range(1, n + 1):
 coin = coins[i - 1]
 # 遍历每种金额
 for j in range(amount + 1):
 # 不使用当前硬币的情况
 dp[i][j] = dp[i - 1][j]

 # 使用当前硬币的情况（可以使用多次）
 if j >= coin:
 dp[i][j] += dp[i][j - coin]

return dp[n][amount]

```

```

def permutation_change(amount, coins):
 """
 计算排列数的版本（与本题要求不同，仅作对比）
 注意：先遍历金额再遍历硬币，这样会计算排列数

```

Args:

amount: 总金额  
coins: 不同面额的硬币数组

Returns:

int: 硬币排列数

"""

```

参数验证
if amount == 0:
 return 1
if not coins:
 return 0

创建 DP 数组
dp = [0] * (amount + 1)
dp[0] = 1

```

```
先遍历金额再遍历硬币，计算的是排列数
for i in range(1, amount + 1):
 for coin in coins:
 if i >= coin:
 dp[i] += dp[i - coin]

return dp[amount]
```

```
def change_dfs(amount, coins):
```

```
"""
```

```
递归+记忆化搜索实现
```

```
Args:
```

```
 amount: 总金额
```

```
 coins: 不同面额的硬币数组
```

```
Returns:
```

```
 int: 硬币组合数
```

```
"""
```

```
参数验证
```

```
if amount == 0:
 return 1
if not coins:
 return 0
```

```
排序有助于剪枝
```

```
coins.sort()
```

```
使用记忆化搜索
```

```
memo = {}
```

```
def dfs(index, remaining):
 # 基础情况：凑成金额 0，找到一种方式
 if remaining == 0:
 return 1

 # 基础情况：无法凑成
 if index == len(coins) or remaining < 0:
 return 0

 # 生成记忆化键
 key = (index, remaining)
```

```
if key in memo:
 return memo[key]

不使用当前硬币的情况
not_use = dfs(index + 1, remaining)

使用当前硬币的情况（如果可以使用）
use = 0
if remaining >= coins[index]:
 # 注意这里 index 不变，表示可以重复使用当前硬币
 use = dfs(index, remaining - coins[index])

计算结果并记忆化
memo[key] = not_use + use
return memo[key]

return dfs(0, amount)

测试用例
if __name__ == "__main__":
 # 测试用例 1
 coins1 = [1, 2, 5]
 amount1 = 5
 print(f"测试用例 1 结果: {change(amount1, coins1)}") # 预期输出: 4

 # 测试用例 2
 coins2 = [2]
 amount2 = 3
 print(f"测试用例 2 结果: {change(amount2, coins2)}") # 预期输出: 0

 # 测试用例 3
 coins3 = [10]
 amount3 = 10
 print(f"测试用例 3 结果: {change(amount3, coins3)}") # 预期输出: 1

 # 测试用例 4
 coins4 = [1, 2, 5, 10]
 amount4 = 10
 print(f"测试用例 4 结果: {change(amount4, coins4)}") # 预期输出: 11
```

```
=====
#include <iostream>
#include <vector>
#include <numeric>
#include <algorithm>

// LeetCode 416. 分割等和子集
// 题目描述：给你一个 只包含正整数 的 非空 数组 nums 。请你判断是否可以将这个数组分割成两个子集，使得两个子集的元素和相等。
// 链接: https://leetcode.cn/problems/partition-equal-subset-sum/
//
// 解题思路：
// 这是一个典型的 01 背包问题的变种。我们可以将问题转化为：
// 1. 计算数组的总和 sum
// 2. 如果 sum 是奇数，直接返回 false (无法分成两个和相等的子集)
// 3. 如果 sum 是偶数，问题转化为：是否存在一个子集，使得其和为 sum/2
// 4. 这相当于 01 背包问题，容量为 sum/2，物品价值和重量都是 nums[i]，问是否能恰好装满背包
//
// 状态定义：dp[i] 表示是否可以选择一些数字，使得它们的和恰好为 i
// 状态转移方程：dp[i] = dp[i] || dp[i - nums[j]]，其中 j 遍历所有数字，且 i >= nums[j]
// 初始状态：dp[0] = true (表示和为 0 可以通过不选任何数字来实现)
//
// 时间复杂度：O(n * target)，其中 n 是数组长度，target 是 sum/2
// 空间复杂度：O(target)，使用一维 DP 数组
```

```
using namespace std;
```

```
/**
```

```
* 判断是否可以将数组分割成两个和相等的子集
```

```
* @param nums 非空正整数数组
```

```
* @return 是否可以分割
```

```
*/
```

```
bool canPartition(vector<int>& nums) {
```

```
 // 参数验证
```

```
 if (nums.size() <= 1) {
```

```
 return false;
```

```
}
```

```
 // 计算数组总和
```

```
 int sum = accumulate(nums.begin(), nums.end(), 0);
```

```
 // 如果总和是奇数，无法分成两个和相等的子集
```

```
 if (sum % 2 != 0) {
```

```

 return false;
 }

// 目标和为总和的一半
int target = sum / 2;

// 创建 DP 数组, dp[i] 表示是否可以选择一些数字, 使得它们的和恰好为 i
vector<bool> dp(target + 1, false);

// 初始状态: 和为 0 可以通过不选任何数字来实现
dp[0] = true;

// 遍历每个数字 (物品)
for (int num : nums) {
 // 逆序遍历目标和 (容量), 防止重复使用同一个数字
 for (int i = target; i >= num; i--) {
 // 状态转移: 选择当前数字或不选择当前数字
 dp[i] = dp[i] || dp[i - num];
 }
}

// 提前终止: 如果已经找到可以组成 target 的子集, 直接返回 true
if (dp[target]) {
 return true;
}
}

return dp[target];
}

/***
 * 优化版本: 添加一些剪枝条件
 */
bool canPartitionOptimized(vector<int>& nums) {
 // 参数验证
 if (nums.size() <= 1) {
 return false;
 }

 // 计算数组总和, 并找出最大值
 int sum = 0;
 int maxNum = 0;
 for (int num : nums) {
 sum += num;

```

```

 maxNum = max(maxNum, num);
 }

// 如果总和是奇数，无法分成两个和相等的子集
if (sum % 2 != 0) {
 return false;
}

int target = sum / 2;

// 如果数组中最大值大于 target，那么这个数字必须单独在一个子集，但剩下的数字之和无法等于 target
if (maxNum > target) {
 return false;
}

// 如果数组中有元素等于 target，直接返回 true
for (int num : nums) {
 if (num == target) {
 return true;
 }
}

// 创建 DP 数组
vector<bool> dp(target + 1, false);
dp[0] = true;

for (int num : nums) {
 for (int i = target; i >= num; i--) {
 dp[i] = dp[i] || dp[i - num];
 if (dp[target]) {
 return true;
 }
 }
}

return dp[target];
}

/***
 * 二维 DP 实现，更容易理解
 */
bool canPartition2D(vector<int>& nums) {

```

```
// 参数验证
if (nums.size() <= 1) {
 return false;
}

int sum = accumulate(nums.begin(), nums.end(), 0);

if (sum % 2 != 0) {
 return false;
}

int target = sum / 2;
int n = nums.size();

// dp[i][j]表示前 i 个数字是否可以组成和为 j 的子集
vector<vector<bool>> dp(n + 1, vector<bool>(target + 1, false));

// 初始化: 前 0 个数字只能组成和为 0 的子集
dp[0][0] = true;

// 遍历每个数字
for (int i = 1; i <= n; i++) {
 int num = nums[i - 1];
 // 遍历每个可能的和
 for (int j = 0; j <= target; j++) {
 // 不选择当前数字
 dp[i][j] = dp[i - 1][j];

 // 选择当前数字 (如果 j >= num)
 if (j >= num) {
 dp[i][j] = dp[i][j] || dp[i - 1][j - num];
 }
 }
}

// 提前终止
if (dp[i][target]) {
 return true;
}

return dp[n][target];
```

```

/**
 * 递归+记忆化搜索实现
 */
bool canPartitionDFS(vector<int>& nums) {
 // 参数验证
 if (nums.size() <= 1) {
 return false;
 }

 int sum = accumulate(nums.begin(), nums.end(), 0);

 if (sum % 2 != 0) {
 return false;
 }

 int target = sum / 2;

 // 使用记忆化搜索, memo[i][j]表示从第 i 个数字开始, 是否可以找到和为 j 的子集
 vector<vector<int>> memo(nums.size(), vector<int>(target + 1, -1)); // -1 表示未计算, 0 表示
false, 1 表示 true

 function<bool(int, int)> dfs = [&](int index, int currentTarget) -> bool {
 // 基础情况: 找到目标和
 if (currentTarget == 0) {
 return true;
 }

 // 基础情况: 超出数组范围或目标和为负
 if (index >= nums.size() || currentTarget < 0) {
 return false;
 }

 // 如果已经计算过, 直接返回结果
 if (memo[index][currentTarget] != -1) {
 return memo[index][currentTarget] == 1;
 }

 // 选择当前数字或不选择当前数字
 bool result = dfs(index + 1, currentTarget - nums[index]) ||
 dfs(index + 1, currentTarget);

 // 记忆化结果
 memo[index][currentTarget] = result ? 1 : 0;
 };
}

```

```

 return result;
 };

 return dfs(0, target);
}

/***
 * 位运算优化版本
 * 使用位图记录所有可能的和
 */
bool canPartitionBitwise(vector<int>& nums) {
 // 参数验证
 if (nums.size() <= 1) {
 return false;
 }

 int sum = accumulate(nums.begin(), nums.end(), 0);

 if (sum % 2 != 0) {
 return false;
 }

 int target = sum / 2;

 // 使用位掩码表示所有可能的和
 // bit[i] = 1 表示可以组成和为 i 的子集
 // 注意：这里使用 long long 来避免溢出
 // 如果 target 很大，可能需要使用 bitset
 long long dp = 1; // 初始状态：可以组成和为 0 的子集

 for (int num : nums) {
 // 对于每个数字，更新可能的和集合
 dp |= dp << num;
 }

 // 检查是否可以组成和为 target 的子集
 return (dp & (1LL << target)) != 0;
}

int main() {
 // 测试用例 1
 vector<int> nums1 = {1, 5, 11, 5};
 cout << "测试用例 1 结果：" << (canPartition(nums1) ? "true" : "false") << endl; // 预期输出：
}

```

```

true

// 测试用例 2
vector<int> nums2 = {1, 2, 3, 5};
cout << "测试用例 2 结果: " << (canPartition(nums2) ? "true" : "false") << endl; // 预期输出:
false

// 测试用例 3
vector<int> nums3 = {1, 2, 5};
cout << "测试用例 3 结果: " << (canPartition(nums3) ? "true" : "false") << endl; // 预期输出:
false

// 测试用例 4
vector<int> nums4 = {2, 2, 3, 5};
cout << "测试用例 4 结果: " << (canPartition(nums4) ? "true" : "false") << endl; // 预期输出:
false

// 测试用例 5
vector<int> nums5 = {1, 2, 3, 4, 5, 6, 7};
cout << "测试用例 5 结果: " << (canPartition(nums5) ? "true" : "false") << endl; // 预期输出:
true

return 0;
}

```

=====

文件: Code21\_PartitionEqualSubsetSum.java

=====

```

package class073;

// LeetCode 416. 分割等和子集
// 题目描述: 给你一个 只包含正整数 的 非空 数组 nums 。请你判断是否可以将这个数组分割成两个子集,
使得两个子集的元素和相等。
// 链接: https://leetcode.cn/problems/partition-equal-subset-sum/
//
// 解题思路:
// 这是一个典型的 01 背包问题的变种。我们可以将问题转化为:
// 1. 计算数组的总和 sum
// 2. 如果 sum 是奇数, 直接返回 false (无法分成两个和相等的子集)
// 3. 如果 sum 是偶数, 问题转化为: 是否存在一个子集, 使得其和为 sum/2
// 4. 这相当于 01 背包问题, 容量为 sum/2, 物品价值和重量都是 nums[i], 问是否能恰好装满背包
//
```

```

// 状态定义: dp[i] 表示是否可以选择一些数字, 使得它们的和恰好为 i
// 状态转移方程: dp[i] = dp[i] || dp[i - nums[j]], 其中 j 遍历所有数字, 且 i >= nums[j]
// 初始状态: dp[0] = true (表示和为 0 可以通过不选任何数字来实现)
//
// 时间复杂度: O(n * target), 其中 n 是数组长度, target 是 sum/2
// 空间复杂度: O(target), 使用一维 DP 数组

public class Code21_PartitionEqualSubsetSum {

 // 主方法, 用于测试
 public static void main(String[] args) {
 // 测试用例 1
 int[] nums1 = {1, 5, 11, 5};
 System.out.println("测试用例 1 结果: " + canPartition(nums1)); // 预期输出: true (分割为 [1, 5, 5] 和 [11])

 // 测试用例 2
 int[] nums2 = {1, 2, 3, 5};
 System.out.println("测试用例 2 结果: " + canPartition(nums2)); // 预期输出: false

 // 测试用例 3
 int[] nums3 = {1, 2, 5};
 System.out.println("测试用例 3 结果: " + canPartition(nums3)); // 预期输出: false

 // 测试用例 4
 int[] nums4 = {2, 2, 3, 5};
 System.out.println("测试用例 4 结果: " + canPartition(nums4)); // 预期输出: false

 // 测试用例 5
 int[] nums5 = {1, 2, 3, 4, 5, 6, 7};
 System.out.println("测试用例 5 结果: " + canPartition(nums5)); // 预期输出: true (分割为 [2, 3, 5, 7] 和 [1, 4, 6])
 }

 /**
 * 判断是否可以将数组分割成两个和相等的子集
 * @param nums 非空正整数数组
 * @return 是否可以分割
 */
 public static boolean canPartition(int[] nums) {
 // 参数验证
 if (nums == null || nums.length <= 1) {
 return false;
 }

```

```
}

// 计算数组总和
int sum = 0;
for (int num : nums) {
 sum += num;
}

// 如果总和是奇数，无法分成两个和相等的子集
if (sum % 2 != 0) {
 return false;
}

// 目标和为总和的一半
int target = sum / 2;

// 创建 DP 数组，dp[i] 表示是否可以选择一些数字，使得它们的和恰好为 i
boolean[] dp = new boolean[target + 1];

// 初始状态：和为 0 可以通过不选任何数字来实现
dp[0] = true;

// 遍历每个数字（物品）
for (int num : nums) {
 // 逆序遍历目标和（容量），防止重复使用同一个数字
 for (int i = target; i >= num; i--) {
 // 状态转移：选择当前数字或不选择当前数字
 // 如果不选择当前数字，dp[i] 保持不变
 // 如果选择当前数字，则要看 dp[i - num] 是否为 true
 dp[i] = dp[i] || dp[i - num];
 }
}

// 提前终止：如果已经找到可以组成 target 的子集，直接返回 true
if (dp[target]) {
 return true;
}

return dp[target];
}

/***
 * 优化版本：添加一些剪枝条件
 */
```

```
/*
public static boolean canPartitionOptimized(int[] nums) {
 // 参数验证
 if (nums == null || nums.length <= 1) {
 return false;
 }

 // 计算数组总和，并找出最大值
 int sum = 0;
 int maxNum = 0;
 for (int num : nums) {
 sum += num;
 maxNum = Math.max(maxNum, num);
 }

 // 如果总和是奇数，无法分成两个和相等的子集
 if (sum % 2 != 0) {
 return false;
 }

 int target = sum / 2;

 // 如果数组中最大值大于 target，那么这个数字必须单独在一个子集，但剩下的数字之和无法等于 target
 if (maxNum > target) {
 return false;
 }

 // 如果数组中有元素等于 target，直接返回 true
 for (int num : nums) {
 if (num == target) {
 return true;
 }
 }

 // 创建 DP 数组
 boolean[] dp = new boolean[target + 1];
 dp[0] = true;

 for (int num : nums) {
 for (int i = target; i >= num; i--) {
 dp[i] = dp[i] || dp[i - num];
 if (dp[target]) {

```

```

 return true;
 }
}

}

return dp[target];
}

/***
 * 二维 DP 实现，更容易理解
 */
public static boolean canPartition2D(int[] nums) {
 // 参数验证
 if (nums == null || nums.length <= 1) {
 return false;
 }

 int sum = 0;
 for (int num : nums) {
 sum += num;
 }

 if (sum % 2 != 0) {
 return false;
 }

 int target = sum / 2;
 int n = nums.length;

 // dp[i][j] 表示前 i 个数字是否可以组成和为 j 的子集
 boolean[][] dp = new boolean[n + 1][target + 1];

 // 初始化：前 0 个数字只能组成和为 0 的子集
 dp[0][0] = true;

 // 遍历每个数字
 for (int i = 1; i <= n; i++) {
 int num = nums[i - 1];
 // 遍历每个可能的和
 for (int j = 0; j <= target; j++) {
 // 不选择当前数字
 dp[i][j] = dp[i - 1][j];
 if (j >= num) {
 dp[i][j] |= dp[i - 1][j - num];
 }
 }
 }
}

```

```

 // 选择当前数字 (如果 j >= num)
 if (j >= num) {
 dp[i][j] = dp[i][j] || dp[i - 1][j - num];
 }
 }

 // 提前终止
 if (dp[i][target]) {
 return true;
 }
}

return dp[n][target];
}

/***
 * 递归+记忆化搜索实现
 */
public static boolean canPartitionDFS(int[] nums) {
 // 参数验证
 if (nums == null || nums.length <= 1) {
 return false;
 }

 int sum = 0;
 for (int num : nums) {
 sum += num;
 }

 if (sum % 2 != 0) {
 return false;
 }

 int target = sum / 2;

 // 使用记忆化搜索, memo[i][j] 表示从第 i 个数字开始, 是否可以找到和为 j 的子集
 Boolean[][] memo = new Boolean[nums.length][target + 1];

 return dfs(nums, 0, target, memo);
}

/***
 * 递归辅助函数

```

```

* @param nums 数组
* @param index 当前考虑的索引
* @param target 目标和
* @param memo 记忆化数组
* @return 是否可以找到和为 target 的子集
*/
private static boolean dfs(int[] nums, int index, int target, Boolean[][] memo) {
 // 基础情况：找到目标和
 if (target == 0) {
 return true;
 }

 // 基础情况：超出数组范围或目标和为负
 if (index >= nums.length || target < 0) {
 return false;
 }

 // 如果已经计算过，直接返回结果
 if (memo[index][target] != null) {
 return memo[index][target];
 }

 // 选择当前数字或不选择当前数字
 boolean result = dfs(nums, index + 1, target - nums[index], memo) ||
 dfs(nums, index + 1, target, memo);

 // 记忆化结果
 memo[index][target] = result;
 return result;
}

/**
 * 位运算优化版本
 * 使用位图记录所有可能的和
 */
public static boolean canPartitionBitwise(int[] nums) {
 // 参数验证
 if (nums == null || nums.length <= 1) {
 return false;
 }

 int sum = 0;
 for (int num : nums) {

```

```

 sum += num;
 }

 if (sum % 2 != 0) {
 return false;
 }

 int target = sum / 2;

 // 使用位掩码表示所有可能的和
 // bit[i] = 1 表示可以组成和为 i 的子集
 int dp = 1; // 初始状态: 可以组成和为 0 的子集

 for (int num : nums) {
 // 对于每个数字, 更新可能的和集合
 // dp |= dp << num 表示当前数字可以与之前的每个和相加, 产生新的和
 dp |= dp << num;
 }

 // 检查是否可以组成和为 target 的子集
 return (dp & (1 << target)) != 0;
}
}

```

---

文件: Code21\_PartitionEqualSubsetSum.py

---

```

LeetCode 416. 分割等和子集
题目描述: 给你一个 只包含正整数 的 非空 数组 nums 。请你判断是否可以将这个数组分割成两个子集,
使得两个子集的元素和相等。
链接: https://leetcode.cn/problems/partition-equal-subset-sum/
#
解题思路:
这是一个典型的 01 背包问题的变种。我们可以将问题转化为:
1. 计算数组的总和 sum
2. 如果 sum 是奇数, 直接返回 false (无法分成两个和相等的子集)
3. 如果 sum 是偶数, 问题转化为: 是否存在一个子集, 使得其和为 sum/2
4. 这相当于 01 背包问题, 容量为 sum/2, 物品价值和重量都是 nums[i], 问是否能恰好装满背包
#
状态定义: dp[i] 表示是否可以选择一些数字, 使得它们的和恰好为 i
状态转移方程: dp[i] = dp[i] || dp[i - nums[j]], 其中 j 遍历所有数字, 且 i >= nums[j]
初始状态: dp[0] = True (表示和为 0 可以通过不选任何数字来实现)

```

```

时间复杂度: O(n * target), 其中 n 是数组长度, target 是 sum/2
空间复杂度: O(target), 使用一维 DP 数组

def can_partition(nums):
 """
 判断是否可以将数组分割成两个和相等的子集

 Args:
 nums: 非空正整数数组

 Returns:
 bool: 是否可以分割
 """

 # 参数验证
 if len(nums) <= 1:
 return False

 # 计算数组总和
 total_sum = sum(nums)

 # 如果总和是奇数, 无法分成两个和相等的子集
 if total_sum % 2 != 0:
 return False

 # 目标和为总和的一半
 target = total_sum // 2

 # 创建 DP 数组, dp[i] 表示是否可以选择一些数字, 使得它们的和恰好为 i
 dp = [False] * (target + 1)

 # 初始状态: 和为 0 可以通过不选任何数字来实现
 dp[0] = True

 # 遍历每个数字 (物品)
 for num in nums:
 # 逆序遍历目标和 (容量), 防止重复使用同一个数字
 for i in range(target, num - 1, -1):
 # 状态转移: 选择当前数字或不选择当前数字
 dp[i] = dp[i] or dp[i - num]

 # 提前终止: 如果已经找到可以组成 target 的子集, 直接返回 True
 if dp[target]:
```

```
 return True

 return dp[target]

def can_partition_optimized(nums):
 """
 优化版本：添加一些剪枝条件

 Args:
 nums: 非空正整数数组

 Returns:
 bool: 是否可以分割
 """
 # 参数验证
 if len(nums) <= 1:
 return False

 # 计算数组总和，并找出最大值
 total_sum = sum(nums)
 max_num = max(nums)

 # 如果总和是奇数，无法分成两个和相等的子集
 if total_sum % 2 != 0:
 return False

 target = total_sum // 2

 # 如果数组中最大值大于 target，那么这个数字必须单独在一个子集，但剩下的数字之和无法等于 target
 if max_num > target:
 return False

 # 如果数组中有元素等于 target，直接返回 True
 if target in nums:
 return True

 # 创建 DP 数组
 dp = [False] * (target + 1)
 dp[0] = True

 for num in nums:
 for i in range(target, num - 1, -1):
 dp[i] = dp[i] or dp[i - num]
```

```

 if dp[target]:
 return True

 return dp[target]

def can_partition_2d(nums):
 """
 二维 DP 实现，更容易理解

 Args:
 nums: 非空正整数数组

 Returns:
 bool: 是否可以分割
 """

 # 参数验证
 if len(nums) <= 1:
 return False

 total_sum = sum(nums)

 if total_sum % 2 != 0:
 return False

 target = total_sum // 2
 n = len(nums)

 # dp[i][j] 表示前 i 个数字是否可以组成和为 j 的子集
 dp = [[False] * (target + 1) for _ in range(n + 1)]

 # 初始化：前 0 个数字只能组成和为 0 的子集
 dp[0][0] = True

 # 遍历每个数字
 for i in range(1, n + 1):
 num = nums[i - 1]
 # 遍历每个可能的和
 for j in range(target + 1):
 # 不选择当前数字
 dp[i][j] = dp[i - 1][j]

 # 选择当前数字（如果 j >= num）
 if j >= num:

```

```
dp[i][j] = dp[i][j] or dp[i - 1][j - num]
```

```
提前终止
if dp[i][target]:
 return True

return dp[n][target]
```

```
def can_partition_dfs(nums):
```

```
"""
```

```
递归+记忆化搜索实现
```

```
Args:
```

```
 nums: 非空正整数数组
```

```
Returns:
```

```
 bool: 是否可以分割
```

```
"""
```

```
参数验证
```

```
if len(nums) <= 1:
 return False
```

```
total_sum = sum(nums)
```

```
if total_sum % 2 != 0:
 return False
```

```
target = total_sum // 2
```

```
使用记忆化搜索
```

```
memo = {}
```

```
def dfs(index, current_target):
```

```
 # 基础情况: 找到目标和
```

```
 if current_target == 0:
 return True
```

```
 # 基础情况: 超出数组范围或目标和为负
```

```
 if index >= len(nums) or current_target < 0:
 return False
```

```
 # 生成记忆化键
```

```
 key = (index, current_target)
```

```

 if key in memo:
 return memo[key]

 # 选择当前数字或不选择当前数字
 result = (dfs(index + 1, current_target - nums[index]) or
 dfs(index + 1, current_target))

 # 记忆化结果
 memo[key] = result
 return result

return dfs(0, target)

```

```

def can_partition_bitwise(nums):
 """

```

位运算优化版本

使用位图记录所有可能的和

Args:

nums: 非空正整数数组

Returns:

bool: 是否可以分割

"""

# 参数验证

if len(nums) <= 1:

return False

total\_sum = sum(nums)

if total\_sum % 2 != 0:

return False

target = total\_sum // 2

# 使用位掩码表示所有可能的和

# bit[i] = 1 表示可以组成和为 i 的子集

dp = 1 # 初始状态: 可以组成和为 0 的子集

for num in nums:

# 对于每个数字, 更新可能的和集合

dp |= dp << num

```

检查是否可以组成和为 target 的子集
return (dp & (1 << target)) != 0

测试用例
if __name__ == "__main__":
 # 测试用例 1
 nums1 = [1, 5, 11, 5]
 print(f"测试用例 1 结果: {can_partition(nums1)}") # 预期输出: True

 # 测试用例 2
 nums2 = [1, 2, 3, 5]
 print(f"测试用例 2 结果: {can_partition(nums2)}") # 预期输出: False

 # 测试用例 3
 nums3 = [1, 2, 5]
 print(f"测试用例 3 结果: {can_partition(nums3)}") # 预期输出: False

 # 测试用例 4
 nums4 = [2, 2, 3, 5]
 print(f"测试用例 4 结果: {can_partition(nums4)}") # 预期输出: False

 # 测试用例 5
 nums5 = [1, 2, 3, 4, 5, 6, 7]
 print(f"测试用例 5 结果: {can_partition(nums5)}") # 预期输出: True

```

---

文件: Code22\_LastStoneWeightII.cpp

---

```

// LeetCode 1049. 最后一块石头的重量 II
// 题目描述: 有一堆石头，每块石头的重量都是正整数。
// 每一回合，从中选出任意两块石头，然后将它们一起粉碎。假设石头的重量分别为 x 和 y，且 x <= y。
// 那么粉碎的可能结果如下：
// - 如果 x == y，那么两块石头都会被完全粉碎；
// - 如果 x != y，那么重量为 x 的石头将会完全粉碎，而重量为 y 的石头新重量为 y-x。
// 最后，最多只会剩下一块石头。返回此石头最小的可能重量。如果没有石头剩下，就返回 0。
// 链接: https://leetcode.cn/problems/last-stone-weight-ii/
//
// 解题思路:
// 这是一个可以转化为 01 背包问题的变种。我们的目标是将石头分成两堆，使得它们的重量尽可能接近，
// 这样最后剩下的石头重量就会最小（等于两堆重量之差）。
//
// 具体分析:

```

```

// 1. 如果我们能将石头分成两堆，重量分别为 sum1 和 sum2，那么最终剩下的石头重量为|sum1 - sum2|
// 2. 由于 sum1 + sum2 = totalSum (石头总重量)，所以剩下的重量为|totalSum - 2*sum1|
// 3. 为了最小化这个值，我们需要让 sum1 尽可能接近 totalSum/2
// 4. 这转化为：在石头中选择一些，使得它们的总重量不超过 totalSum/2，且尽可能接近 totalSum/2
// 5. 这正是一个 01 背包问题，背包容量为 totalSum/2，物品重量为石头重量，目标是最大化能装入的重量
//
// 状态定义：dp[i] 表示是否可以组成和为 i 的石头堆
// 状态转移方程：dp[i] = dp[i] || dp[i - stones[j]]，其中 j 遍历所有石头，且 i >= stones[j]
// 初始状态：dp[0] = true (表示和为 0 可以通过不选任何石头来实现)
//
// 时间复杂度：O(n * target)，其中 n 是石头数量，target 是总重量的一半
// 空间复杂度：O(target)，使用一维 DP 数组

/**
 * 计算最后一块石头的最小可能重量
 *
 * 解题思路：
 * 1. 这道题可以转化为将石头分为两堆，使得两堆重量差最小
 * 2. 假设两堆分别为 A 和 B，A >= B
 * 3. 最终剩下的石头重量就是 A - B
 * 4. 要使 A - B 最小，就要使 B 尽可能接近 sum/2
 * 5. 问题转化为：在不超过 sum/2 的前提下，背包最多能装多少重量的石头
 * 6. 这就是一个标准的 01 背包问题
 *
 * @param stones 石头重量数组
 * @param stonesSize 石头数量
 * @return 最后一块石头的最小可能重量
 */
int lastStoneWeightII(int* stones, int stonesSize) {
 // 参数验证
 if (stones == 0 || stonesSize == 0) {
 return 0;
 }
 if (stonesSize == 1) {
 return stones[0];
 }

 // 计算石头总重量
 int totalSum = 0;
 for (int i = 0; i < stonesSize; i++) {
 totalSum += stones[i];
 }

```

```

// 目标是找到不超过 totalSum/2 的最大子集和
int target = totalSum / 2;

// 创建 DP 数组, dp[i] 表示是否可以组成和为 i 的石头堆
// 由于不能使用动态内存分配, 我们使用固定大小的数组
// 假设 target 最大不超过 10000
bool dp[10001] = {false};

// 初始状态: 和为 0 可以通过不选任何石头来实现
dp[0] = true;

// 遍历每个石头 (物品)
for (int i = 0; i < stonesSize; i++) {
 int stone = stones[i];
 // 逆序遍历目标和 (容量), 防止重复使用同一个石头
 for (int j = target; j >= stone; j--) {
 // 状态转移: 选择当前石头或不选择当前石头
 dp[j] = dp[j] || dp[j - stone];
 }
}

// 找到最大的 i, 使得 dp[i] 为 true
int maxSum = 0;
for (int i = target; i >= 0; i--) {
 if (dp[i]) {
 maxSum = i;
 break;
 }
}

// 最后一块石头的最小可能重量为总重量减去两倍的最大子集和
return totalSum - 2 * maxSum;
}

/***
 * 优化版本: 使用一维 DP 数组记录可以达到的最大和
 *
 * 解题思路:
 * 在基础版本的基础上进行优化, 通过记录当前可以达到的最大和来提前结束循环
 *
 * @param stones 石头重量数组
 * @param stonesSize 石头数量
 * @return 最后一块石头的最小可能重量
*/

```

```

*/
int lastStoneWeightIIOptimized(int* stones, int stonesSize) {
 // 参数验证
 if (stones == 0 || stonesSize == 0) {
 return 0;
 }
 if (stonesSize == 1) {
 return stones[0];
 }

 // 计算石头总重量
 int totalSum = 0;
 for (int i = 0; i < stonesSize; i++) {
 totalSum += stones[i];
 }

 // 目标是找到不超过 totalSum/2 的最大子集和
 int target = totalSum / 2;

 // 创建 DP 数组, dp[i] 表示是否可以组成和为 i 的石头堆
 bool dp[10001] = {false};
 dp[0] = true;

 // 记录当前可以达到的最大和
 int currentMax = 0;

 for (int i = 0; i < stonesSize; i++) {
 int stone = stones[i];
 // 逆序遍历, 但只遍历到当前可能的最大和+stone
 int limit = (target < (currentMax + stone)) ? target : (currentMax + stone);
 for (int j = limit; j >= stone; j--) {
 if (dp[j - stone]) {
 dp[j] = true;
 // 更新当前可以达到的最大和
 currentMax = (currentMax > j) ? currentMax : j;
 // 如果已经可以达到 target, 提前结束
 if (currentMax == target) {
 return totalSum - 2 * target;
 }
 }
 }
 }
}

```

```

 return totalSum - 2 * currentMax;
}

/***
 * 二维 DP 实现，更容易理解
 *
 * 解题思路：
 * 使用二维 DP 数组，dp[i][j] 表示前 i 个石头是否可以组成和为 j 的石头堆
 * 状态转移方程：dp[i][j] = dp[i-1][j] || dp[i-1][j-stones[i-1]]
 *
 * @param stones 石头重量数组
 * @param stonesSize 石头数量
 * @return 最后一块石头的最小可能重量
 */
int lastStoneWeightII2D(int* stones, int stonesSize) {
 // 参数验证
 if (stones == 0 || stonesSize == 0) {
 return 0;
 }
 if (stonesSize == 1) {
 return stones[0];
 }

 int totalSum = 0;
 for (int i = 0; i < stonesSize; i++) {
 totalSum += stones[i];
 }

 int target = totalSum / 2;

 // dp[i][j] 表示前 i 个石头是否可以组成和为 j 的石头堆
 // 由于不能使用动态内存分配，我们使用固定大小的数组
 bool dp[51][10001] = {{false}}; // 假设石头数量不超过 50，目标和不超过 10000

 // 初始化：前 0 个石头只能组成和为 0 的石头堆
 dp[0][0] = true;

 // 遍历每个石头
 for (int i = 1; i <= stonesSize; i++) {
 int stone = stones[i - 1];
 // 遍历每个可能的和
 for (int j = 0; j <= target; j++) {
 // 不选择当前石头
 dp[i][j] = dp[i-1][j] || dp[i-1][j-stone];
 }
 }
}

```

```

dp[i][j] = dp[i - 1][j];

// 选择当前石头 (如果 j >= stone)
if (j >= stone) {
 dp[i][j] = dp[i][j] || dp[i - 1][j - stone];
}

}

// 找到最大的 j, 使得 dp[stonesSize][j] 为 true
int maxSum = 0;
for (int j = target; j >= 0; j--) {
 if (dp[stonesSize][j]) {
 maxSum = j;
 break;
 }
}

return totalSum - 2 * maxSum;
}

```

=====

文件: Code22\_LastStoneWeightII.java

=====

```

package class073;

// LeetCode 1049. 最后一块石头的重量 II
// 题目描述: 有一堆石头, 每块石头的重量都是正整数。
// 每一回合, 从中选出任意两块石头, 然后将它们一起粉碎。假设石头的重量分别为 x 和 y, 且 x <= y。
// 那么粉碎的可能结果如下:
// - 如果 x == y, 那么两块石头都会被完全粉碎;
// - 如果 x != y, 那么重量为 x 的石头将会完全粉碎, 而重量为 y 的石头新重量为 y-x。
// 最后, 最多只会剩下一块石头。返回此石头最小的可能重量。如果没有石头剩下, 就返回 0。
// 链接: https://leetcode.cn/problems/last-stone-weight-ii/
//
// 解题思路:
// 这是一个可以转化为 01 背包问题的变种。我们的目标是将石头分成两堆, 使得它们的重量尽可能接近,
// 这样最后剩下的石头重量就会最小 (等于两堆重量之差)。
//
// 具体分析:
// 1. 如果我们能将石头分成两堆, 重量分别为 sum1 和 sum2, 那么最终剩下的石头重量为 |sum1 - sum2|
// 2. 由于 sum1 + sum2 = totalSum (石头总重量), 所以剩下的重量为 |totalSum - 2*sum1|

```

```

// 3. 为了最小化这个值，我们需要让 sum1 尽可能接近 totalSum/2
// 4. 这转化为：在石头中选择一些，使得它们的总重量不超过 totalSum/2，且尽可能接近 totalSum/2
// 5. 这正是一个 01 背包问题，背包容量为 totalSum/2，物品重量为石头重量，目标是最大化能装入的重量
//
// 状态定义：dp[i] 表示是否可以组成和为 i 的石头堆
// 状态转移方程：dp[i] = dp[i] || dp[i - stones[j]]，其中 j 遍历所有石头，且 i >= stones[j]
// 初始状态：dp[0] = true（表示和为 0 可以通过不选任何石头来实现）
//
// 时间复杂度：O(n * target)，其中 n 是石头数量，target 是总重量的一半
// 空间复杂度：O(target)，使用一维 DP 数组

```

```
public class Code22_LastStoneWeightII {
```

```
// 主方法，用于测试
```

```
public static void main(String[] args) {
```

```
 // 测试用例 1
```

```
 int[] stones1 = {2, 7, 4, 1, 8, 1};
```

```
 System.out.println("测试用例 1 结果：" + lastStoneWeightII(stones1)); // 预期输出：1
```

```
 // 解释：
```

```
 // 组合 2 和 4，得到 2，所以数组转化为 [2, 7, 1, 8, 1]
```

```
 // 组合 7 和 8，得到 1，所以数组转化为 [2, 1, 1, 1]
```

```
 // 组合 2 和 1，得到 1，所以数组转化为 [1, 1]
```

```
 // 组合 1 和 1，得到 0，所以最终只剩下 0
```

```
 // 测试用例 2
```

```
 int[] stones2 = {31, 26, 33, 21, 40};
```

```
 System.out.println("测试用例 2 结果：" + lastStoneWeightII(stones2)); // 预期输出：5
```

```
 // 测试用例 3
```

```
 int[] stones3 = {1, 2};
```

```
 System.out.println("测试用例 3 结果：" + lastStoneWeightII(stones3)); // 预期输出：1
```

```
 // 测试用例 4
```

```
 int[] stones4 = {1};
```

```
 System.out.println("测试用例 4 结果：" + lastStoneWeightII(stones4)); // 预期输出：1
```

```
}
```

```
/**
```

```
* 计算最后一块石头的最小可能重量
```

```
* @param stones 石头重量数组
```

```
* @return 最后一块石头的最小可能重量
```

```
*/
```

```
public static int lastStoneWeightII(int[] stones) {
```

```
// 参数验证
if (stones == null || stones.length == 0) {
 return 0;
}
if (stones.length == 1) {
 return stones[0];
}

// 计算石头总重量
int totalSum = 0;
for (int stone : stones) {
 totalSum += stone;
}

// 目标是找到不超过 totalSum/2 的最大子集和
int target = totalSum / 2;

// 创建 DP 数组, dp[i] 表示是否可以组成和为 i 的石头堆
boolean[] dp = new boolean[target + 1];

// 初始状态: 和为 0 可以通过不选任何石头来实现
dp[0] = true;

// 遍历每个石头 (物品)
for (int stone : stones) {
 // 逆序遍历目标和 (容量), 防止重复使用同一个石头
 for (int i = target; i >= stone; i--) {
 // 状态转移: 选择当前石头或不选择当前石头
 dp[i] = dp[i] || dp[i - stone];
 }
}

// 找到最大的 i, 使得 dp[i] 为 true
int maxSum = 0;
for (int i = target; i >= 0; i--) {
 if (dp[i]) {
 maxSum = i;
 break;
 }
}

// 最后一块石头的最小可能重量为总重量减去两倍的最大子集和
return totalSum - 2 * maxSum;
```

```
}

/**
 * 优化版本：使用一维 DP 数组记录可以达到的最大和
 */
public static int lastStoneWeightIIOptimized(int[] stones) {
 // 参数验证
 if (stones == null || stones.length == 0) {
 return 0;
 }
 if (stones.length == 1) {
 return stones[0];
 }

 // 计算石头总重量
 int totalSum = 0;
 for (int stone : stones) {
 totalSum += stone;
 }

 // 目标是找到不超过 totalSum/2 的最大子集和
 int target = totalSum / 2;

 // 创建 DP 数组，dp[i] 表示是否可以组成和为 i 的石头堆
 boolean[] dp = new boolean[target + 1];
 dp[0] = true;

 // 记录当前可以达到的最大和
 int currentMax = 0;

 for (int stone : stones) {
 // 逆序遍历
 for (int i = Math.min(target, currentMax + stone); i >= stone; i--) {
 if (dp[i - stone]) {
 dp[i] = true;
 // 更新当前可以达到的最大和
 currentMax = Math.max(currentMax, i);
 // 如果已经可以达到 target，提前结束
 if (currentMax == target) {
 return totalSum - 2 * target;
 }
 }
 }
 }
}
```

```

}

return totalSum - 2 * currentMax;
}

/***
 * 二维 DP 实现，更容易理解
 */
public static int lastStoneWeightII2D(int[] stones) {
 // 参数验证
 if (stones == null || stones.length == 0) {
 return 0;
 }
 if (stones.length == 1) {
 return stones[0];
 }

 int totalSum = 0;
 for (int stone : stones) {
 totalSum += stone;
 }

 int target = totalSum / 2;
 int n = stones.length;

 // dp[i][j] 表示前 i 个石头是否可以组成和为 j 的石头堆
 boolean[][] dp = new boolean[n + 1][target + 1];

 // 初始化：前 0 个石头只能组成和为 0 的石头堆
 dp[0][0] = true;

 // 遍历每个石头
 for (int i = 1; i <= n; i++) {
 int stone = stones[i - 1];
 // 遍历每个可能的和
 for (int j = 0; j <= target; j++) {
 // 不选择当前石头
 dp[i][j] = dp[i - 1][j];

 // 选择当前石头（如果 j >= stone）
 if (j >= stone) {
 dp[i][j] = dp[i][j] || dp[i - 1][j - stone];
 }
 }
 }
}

```

```

 }
 }

 // 找到最大的 j, 使得 dp[n][j] 为 true
 int maxSum = 0;
 for (int j = target; j >= 0; j--) {
 if (dp[n][j]) {
 maxSum = j;
 break;
 }
 }

 return totalSum - 2 * maxSum;
}

/***
 * 递归+记忆化搜索实现
 */
public static int lastStoneWeightIIDFS(int[] stones) {
 // 参数验证
 if (stones == null || stones.length == 0) {
 return 0;
 }
 if (stones.length == 1) {
 return stones[0];
 }

 // 计算石头总重量
 int totalSum = 0;
 for (int stone : stones) {
 totalSum += stone;
 }

 // 目标是找到不超过 totalSum/2 的最大子集和
 int target = totalSum / 2;

 // 使用记忆化搜索, memo[i][j] 表示从第 i 个石头开始, 当前和为 j 时是否可以继续选择石头
 // 由于只需要知道是否可达, 我们可以使用布尔型数组
 Boolean[][] memo = new Boolean[stones.length][target + 1];

 // 尝试找到最大的不超过 target 的子集和
 int maxSum = dfs(stones, 0, 0, target, memo);
}

```

```

 return totalSum - 2 * maxSum;
 }

/***
 * 递归辅助函数，返回从 index 开始，当前和为 currentSum 时，可以达到的不超过 target 的最大子集和
 */
private static int dfs(int[] stones, int index, int currentSum, int target, Boolean[][] memo)
{
 // 基础情况：已经处理完所有石头，或者当前和已经达到目标
 if (index == stones.length || currentSum == target) {
 return currentSum;
 }

 // 如果已经计算过，直接返回结果
 if (memo[index][currentSum] != null) {
 // 如果返回 false，表示从这个状态无法达到更大的和，返回当前和
 return currentSum;
 }

 // 不选择当前石头
 int notTake = dfs(stones, index + 1, currentSum, target, memo);

 // 选择当前石头（如果当前和加上石头重量不超过目标）
 int take = currentSum;
 if (currentSum + stones[index] <= target) {
 take = dfs(stones, index + 1, currentSum + stones[index], target, memo);
 }

 // 记录这个状态可以到达更大的和
 memo[index][currentSum] = (notTake > currentSum || take > currentSum);

 // 返回较大的那个结果
 return Math.max(notTake, take);
}

/***
 * 位运算优化版本
 * 使用位图记录所有可能的子集和
 */
public static int lastStoneWeightIIBitwise(int[] stones) {
 // 参数验证
 if (stones == null || stones.length == 0) {
 return 0;
 }
}

```

```

 }

 if (stones.length == 1) {
 return stones[0];
 }

 // 计算石头总重量
 int totalSum = 0;
 for (int stone : stones) {
 totalSum += stone;
 }

 int target = totalSum / 2;

 // 使用位掩码表示所有可能的子集和
 // bit[i] = 1 表示可以组成和为 i 的子集
 int dp = 1; // 初始状态：可以组成和为 0 的子集

 for (int stone : stones) {
 // 对于每个石头，更新可能的子集和集合
 dp |= dp << stone;
 }

 // 找到最大的 i <= target，使得 dp 的第 i 位为 1
 int maxSum = 0;
 for (int i = target; i >= 0; i--) {
 if ((dp & (1 << i)) != 0) {
 maxSum = i;
 break;
 }
 }

 return totalSum - 2 * maxSum;
}
}

```

=====

文件: Code22\_LastStoneWeightII.py

=====

```

LeetCode 1049. 最后一块石头的重量 II
题目描述：有一堆石头，每块石头的重量都是正整数。
每一回合，从中选出任意两块石头，然后将它们一起粉碎。假设石头的重量分别为 x 和 y，且 x <= y。
那么粉碎的可能结果如下：

```

```
- 如果 x == y, 那么两块石头都会被完全粉碎;
- 如果 x != y, 那么重量为 x 的石头将会完全粉碎, 而重量为 y 的石头新重量为 y-x。
最后, 最多只会剩下一块石头。返回此石头最小的可能重量。如果没有石头剩下, 就返回 0。
链接: https://leetcode.cn/problems/last-stone-weight-ii/
#
解题思路:
这是一个可以转化为 01 背包问题的变种。我们的目标是将石头分成两堆, 使得它们的重量尽可能接近,
这样最后剩下的石头重量就会最小(等于两堆重量之差)。
#
具体分析:
1. 如果我们能将石头分成两堆, 重量分别为 sum1 和 sum2, 那么最终剩下的石头重量为 |sum1 - sum2|
2. 由于 $sum1 + sum2 = totalSum$ (石头总重量), 所以剩下的重量为 $|totalSum - 2*sum1|$
3. 为了最小化这个值, 我们需要让 sum1 尽可能接近 $totalSum/2$
4. 这转化为: 在石头中选择一些, 使得它们的总重量不超过 $totalSum/2$, 且尽可能接近 $totalSum/2$
5. 这正是一个 01 背包问题, 背包容量为 $totalSum/2$, 物品重量为石头重量, 目标是最大化能装入的重量
#
状态定义: dp[i] 表示是否可以组成和为 i 的石头堆
状态转移方程: dp[i] = dp[i] || dp[i - stones[j]], 其中 j 遍历所有石头, 且 $i \geq stones[j]$
初始状态: dp[0] = True (表示和为 0 可以通过不选任何石头来实现)
#
时间复杂度: O(n * target), 其中 n 是石头数量, target 是总重量的一半
空间复杂度: O(target), 使用一维 DP 数组
```

```
def last_stone_weight_ii(stones):
```

```
 """

```

```
 计算最后一块石头的最小可能重量

```

```
Args:

```

```
 stones: 石头重量数组

```

```
Returns:

```

```
 int: 最后一块石头的最小可能重量
"""

```

```
参数验证
if not stones:
 return 0
if len(stones) == 1:
 return stones[0]

```

```
计算石头总重量
total_sum = sum(stones)

```

```
目标是找到不超过 total_sum/2 的最大子集和

```

```
target = total_sum // 2

创建 DP 数组, dp[i] 表示是否可以组成和为 i 的石头堆
dp = [False] * (target + 1)

初始状态: 和为 0 可以通过不选任何石头来实现
dp[0] = True

遍历每个石头 (物品)
for stone in stones:
 # 逆序遍历目标和 (容量), 防止重复使用同一个石头
 for i in range(target, stone - 1, -1):
 # 状态转移: 选择当前石头或不选择当前石头
 dp[i] = dp[i] or dp[i - stone]

找到最大的 i, 使得 dp[i] 为 True
max_sum = 0
for i in range(target, -1, -1):
 if dp[i]:
 max_sum = i
 break

最后一块石头的最小可能重量为总重量减去两倍的最大子集和
return total_sum - 2 * max_sum
```

```
def last_stone_weight_ii_optimized(stones):
```

```
 """
```

```
 优化版本: 使用一维 DP 数组记录可以达到的最大和
```

```
Args:
```

```
 stones: 石头重量数组
```

```
Returns:
```

```
 int: 最后一块石头的最小可能重量
```

```
 """
```

```
参数验证
```

```
if not stones:
```

```
 return 0
```

```
if len(stones) == 1:
```

```
 return stones[0]
```

```
计算石头总重量
```

```
total_sum = sum(stones)
```

```

目标是找到不超过 total_sum/2 的最大子集和
target = total_sum // 2

创建 DP 数组, dp[i] 表示是否可以组成和为 i 的石头堆
dp = [False] * (target + 1)
dp[0] = True

记录当前可以达到的最大和
current_max = 0

for stone in stones:
 # 逆序遍历, 但只遍历到当前可能的最大和+stone
 for i in range(min(target, current_max + stone), stone - 1, -1):
 if dp[i - stone]:
 dp[i] = True
 # 更新当前可以达到的最大和
 current_max = max(current_max, i)
 # 如果已经可以达到 target, 提前结束
 if current_max == target:
 return total_sum - 2 * target

return total_sum - 2 * current_max

```

```
def last_stone_weight_ii_2d(stones):
```

```
"""
```

二维 DP 实现, 更容易理解

Args:

stones: 石头重量数组

Returns:

int: 最后一块石头的最小可能重量

```
"""
```

# 参数验证

if not stones:

return 0

if len(stones) == 1:

return stones[0]

total\_sum = sum(stones)

target = total\_sum // 2

n = len(stones)

```

dp[i][j]表示前 i 个石头是否可以组成和为 j 的石头堆
dp = [[False] * (target + 1) for _ in range(n + 1)]

初始化: 前 0 个石头只能组成和为 0 的石头堆
dp[0][0] = True

遍历每个石头
for i in range(1, n + 1):
 stone = stones[i - 1]
 # 遍历每个可能的和
 for j in range(target + 1):
 # 不选择当前石头
 dp[i][j] = dp[i - 1][j]

 # 选择当前石头 (如果 j >= stone)
 if j >= stone:
 dp[i][j] = dp[i][j] or dp[i - 1][j - stone]

找到最大的 j, 使得 dp[n][j] 为 True
max_sum = 0
for j in range(target, -1, -1):
 if dp[n][j]:
 max_sum = j
 break

return total_sum - 2 * max_sum

```

```

def last_stone_weight_ii_dfs(stones):
"""
递归+记忆化搜索实现

```

Args:

stones: 石头重量数组

Returns:

int: 最后一块石头的最小可能重量

"""

# 参数验证

if not stones:

return 0

if len(stones) == 1:

return stones[0]

```

计算石头总重量
total_sum = sum(stones)

目标是找到不超过 total_sum/2 的最大子集和
target = total_sum // 2

使用记忆化搜索
memo = {}

def dfs(index, current_sum):
 # 基础情况：已经处理完所有石头，或者当前和已经达到目标
 if index == len(stones) or current_sum == target:
 return current_sum

 # 生成记忆化键
 key = (index, current_sum)
 if key in memo:
 return memo[key]

 # 不选择当前石头
 not_take = dfs(index + 1, current_sum)

 # 选择当前石头（如果当前和加上石头重量不超过目标）
 take = current_sum
 if current_sum + stones[index] <= target:
 take = dfs(index + 1, current_sum + stones[index])

 # 记录结果
 memo[key] = max(not_take, take)
 return memo[key]

尝试找到最大的不超过 target 的子集和
max_sum = dfs(0, 0)

return total_sum - 2 * max_sum

```

```

def last_stone_weight_i_i_bitwise(stones):
"""

```

位运算优化版本  
使用位图记录所有可能的子集和

Args:

```
stones: 石头重量数组
```

```
Returns:
```

```
int: 最后一块石头的最小可能重量
```

```
"""
```

```
参数验证
```

```
if not stones:
```

```
 return 0
```

```
if len(stones) == 1:
```

```
 return stones[0]
```

```
计算石头总重量
```

```
total_sum = sum(stones)
```

```
target = total_sum // 2
```

```
使用位掩码表示所有可能的子集和
```

```
bit[i] = 1 表示可以组成和为 i 的子集
```

```
dp = 1 # 初始状态: 可以组成和为 0 的子集
```

```
for stone in stones:
```

```
 # 对于每个石头, 更新可能的子集和集合
```

```
 dp |= dp << stone
```

```
找到最大的 i <= target, 使得 dp 的第 i 位为 1
```

```
max_sum = 0
```

```
for i in range(target, -1, -1):
```

```
 if (dp & (1 << i)) != 0:
```

```
 max_sum = i
```

```
 break
```

```
return total_sum - 2 * max_sum
```

```
测试用例
```

```
if __name__ == "__main__":
```

```
 # 测试用例 1
```

```
 stones1 = [2, 7, 4, 1, 8, 1]
```

```
 print(f"测试用例 1 结果: {last_stone_weight_ii(stones1)}") # 预期输出: 1
```

```
测试用例 2
```

```
stones2 = [31, 26, 33, 21, 40]
```

```
print(f"测试用例 2 结果: {last_stone_weight_ii(stones2)}") # 预期输出: 5
```

```
测试用例 3
```

```
stones3 = [1, 2]
print(f"测试用例 3 结果: {last_stone_weight_ii(stones3)}") # 预期输出: 1

测试用例 4
stones4 = [1]
print(f"测试用例 4 结果: {last_stone_weight_ii(stones4)}") # 预期输出: 1
```

---

文件: Code23\_OnesAndZeros.cpp

```
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
#include <unordered_map>
#include <tuple>

// LeetCode 474. 一和零
// 题目描述: 给你一个二进制字符串数组 strs 和两个整数 m 和 n。
// 请你找出并返回 strs 的最大子集的大小，该子集中 最多 有 m 个 0 和 n 个 1。
// 如果 x 的所有元素也是 y 的元素，集合 x 是集合 y 的 子集。
// 链接: https://leetcode.cn/problems/ones-and-zeroes/
//
// 解题思路:
// 这是一个多维背包问题，我们需要同时考虑两种资源限制：0 的数量和 1 的数量。
// 每个字符串相当于一个物品，占用的空间是它包含的 0 的数量和 1 的数量，价值为 1（因为我们想最大化子集的大小）。
// 目标是在不超过 m 个 0 和 n 个 1 的限制下，选择尽可能多的字符串。
//
// 状态定义: dp[i][j] 表示使用 i 个 0 和 j 个 1 时，可以选择的最大字符串数量
// 状态转移方程: 对于每个字符串 s，其中有 zeros 个 0 和 ones 个 1,
// dp[i][j] = max(dp[i][j], dp[i-zeros][j-ones] + 1)，当 i >= zeros 且 j >= ones 时
// 初始状态: dp[0][0] = 0，表示不使用任何 0 和 1 时，可以选择 0 个字符串
// 其他初始值也为 0，表示还没有选择任何字符串
//
// 时间复杂度: O(l * m * n)，其中 l 是字符串数组的长度，m 和 n 是给定的整数
// 空间复杂度: O(m * n)，使用二维 DP 数组

using namespace std;

/***
 * 统计字符串中 0 和 1 的数量
 */
```

```

*/
pair<int, int> countZerosOnes(const string& s) {
 int zeros = 0, ones = 0;
 for (char c : s) {
 if (c == '0') {
 zeros++;
 } else {
 ones++;
 }
 }
 return {zeros, ones};
}

/**
 * 找出最大子集的大小，该子集中最多有 m 个 0 和 n 个 1
 * @param strs 二进制字符串数组
 * @param m 最大 0 的数量
 * @param n 最大 1 的数量
 * @return 最大子集的大小
*/
int findMaxForm(vector<string>& strs, int m, int n) {
 // 参数验证
 if (strs.empty()) {
 return 0;
 }

 // 创建二维 DP 数组，dp[i][j] 表示使用 i 个 0 和 j 个 1 时，可以选择的最大字符串数量
 vector<vector<int>> dp(m + 1, vector<int>(n + 1, 0));

 // 遍历每个字符串（物品）
 for (const string& s : strs) {
 // 统计当前字符串中 0 和 1 的数量
 auto [zeros, ones] = countZerosOnes(s);

 // 逆序遍历 m 和 n，避免重复使用同一个字符串
 // 从大到小遍历 0 的数量
 for (int i = m; i >= zeros; i--) {
 // 从大到小遍历 1 的数量
 for (int j = n; j >= ones; j--) {
 // 状态转移：选择当前字符串或不选择当前字符串
 dp[i][j] = max(dp[i][j], dp[i - zeros][j - ones] + 1);
 }
 }
 }
}

```

```

}

// 返回结果：使用最多 m 个 0 和 n 个 1 时，可以选择的最大字符串数量
return dp[m][n];
}

/***
 * 优化版本：预处理字符串的 0 和 1 数量，避免重复计算
 */
int findMaxFormOptimized(vector<string>& strs, int m, int n) {
 // 参数验证
 if (strs.empty()) {
 return 0;
 }

 // 预处理：统计每个字符串中 0 和 1 的数量
 vector<pair<int, int>> counts;
 for (const string& s : strs) {
 int zeros = 0, ones = 0;
 for (char c : s) {
 if (c == '0') {
 zeros++;
 } else {
 ones++;
 }
 }
 counts.push_back({zeros, ones});
 }

 // 创建二维 DP 数组
 vector<vector<int>> dp(m + 1, vector<int>(n + 1, 0));

 // 遍历每个字符串（物品）
 for (const auto& count : counts) {
 int zeros = count.first;
 int ones = count.second;

 // 剪枝：如果当前字符串需要的 0 或 1 超过限制，则跳过
 if (zeros > m || ones > n) {
 continue;
 }

 // 逆序遍历

```

```

 for (int i = m; i >= zeros; i--) {
 for (int j = n; j >= ones; j--) {
 dp[i][j] = max(dp[i][j], dp[i - zeros][j - ones] + 1);
 }
 }
 }

 return dp[m][n];
}

```

```

/***
 * 另一种实现方式，使用滚动数组优化空间
 */
int findMaxFormWithRollingArray(vector<string>& strs, int m, int n) {
 // 参数验证
 if (strs.empty()) {
 return 0;
 }

 // 创建 DP 数组
 vector<vector<int>> dp(m + 1, vector<int>(n + 1, 0));

 for (const string& s : strs) {
 auto [zeros, ones] = countZerosOnes(s);

 // 使用临时数组保存上一状态
 vector<vector<int>> temp = dp;

 for (int i = zeros; i <= m; i++) {
 for (int j = ones; j <= n; j++) {
 dp[i][j] = max(dp[i][j], temp[i - zeros][j - ones] + 1);
 }
 }
 }

 return dp[m][n];
}

```

```

/***
 * 递归+记忆化搜索实现
 * 使用三元组作为键的哈希表来缓存结果
 */
int findMaxFormDFS(vector<string>& strs, int m, int n) {

```

```

// 参数验证
if (strs.empty()) {
 return 0;
}

// 预处理：统计每个字符串中 0 和 1 的数量
vector<pair<int, int>> counts;
for (const string& s : strs) {
 auto [zeros, ones] = countZerosOnes(s);
 // 只保留可能被选中的字符串
 if (zeros <= m && ones <= n) {
 counts.push_back({zeros, ones});
 }
}

// 使用哈希表作为缓存，键为(index, m, n)的组合，值为对应的最大子集大小
// 为了使用自定义键，我们可以将三元组转换为字符串或使用 tuple 作为键
unordered_map<string, int> memo;

// 定义 DFS 函数
function<int(int, int, int)> dfs = [&](int index, int remainingM, int remainingN) -> int {
 // 基础情况：已经处理完所有字符串
 if (index == counts.size()) {
 return 0;
 }

 // 生成缓存键
 string key = to_string(index) + "," + to_string(remainingM) + "," +
 to_string(remainingN);
 if (memo.find(key) != memo.end()) {
 return memo[key];
 }

 // 获取当前字符串的 0 和 1 数量
 int zeros = counts[index].first;
 int ones = counts[index].second;

 // 选择不使用当前字符串
 int notTake = dfs(index + 1, remainingM, remainingN);

 // 选择使用当前字符串（如果有足够的 0 和 1）
 int take = 0;
 if (zeros <= remainingM && ones <= remainingN) {

```

```

 take = 1 + dfs(index + 1, remainingM - zeros, remainingN - ones);
 }

 // 记录结果
 memo[key] = max(notTake, take);

 return memo[key];
};

// 调用递归函数
return dfs(0, m, n);
}

/***
 * 使用 tuple 作为缓存键的版本（需要 C++11 或更高版本）
 */
int findMaxFormDFSWithTuple(vector<string>& strs, int m, int n) {
 // 参数验证
 if (strs.empty()) {
 return 0;
 }

 // 预处理：统计每个字符串中 0 和 1 的数量
 vector<pair<int, int>> counts;
 for (const string& s : strs) {
 auto [zeros, ones] = countZerosOnes(s);
 if (zeros <= m && ones <= n) {
 counts.push_back({zeros, ones});
 }
 }

 // 为 tuple<int, int, int> 创建哈希函数
 struct TupleHash {
 template <typename T1, typename T2, typename T3>
 size_t operator()(const tuple<T1, T2, T3>& t) const {
 auto hash1 = hash<T1>{}(get<0>(t));
 auto hash2 = hash<T2>{}(get<1>(t));
 auto hash3 = hash<T3>{}(get<2>(t));
 // 组合哈希值
 return hash1 ^ (hash2 << 1) ^ (hash3 << 2);
 }
 };
}
```

```

// 使用 tuple 作为键的哈希表
unordered_map<tuple<int, int, int>, int, TupleHash> memo;

// 定义 DFS 函数
function<int(int, int, int)> dfs = [&](int index, int remainingM, int remainingN) -> int {
 if (index == counts.size()) {
 return 0;
 }

 auto key = make_tuple(index, remainingM, remainingN);
 if (memo.find(key) != memo.end()) {
 return memo[key];
 }

 int zeros = counts[index].first;
 int ones = counts[index].second;

 int notTake = dfs(index + 1, remainingM, remainingN);
 int take = 0;
 if (zeros <= remainingM && ones <= remainingN) {
 take = 1 + dfs(index + 1, remainingM - zeros, remainingN - ones);
 }

 memo[key] = max(notTake, take);
 return memo[key];
};

return dfs(0, m, n);
}

/**
 * 贪心算法（仅供参考，不适用于所有情况）
 * 贪心无法保证得到正确结果，但在某些情况下可以作为启发式方法
 */
int findMaxFormGreedy(vector<string>& strs, int m, int n) {
 // 按照字符串长度排序，优先选择较短的字符串（因为它们可能占用更少的 0 和 1）
 sort(strs.begin(), strs.end(), [] (const string& a, const string& b) {
 return a.size() < b.size();
 });

 int result = 0;
 int usedM = 0, usedN = 0;

```

```

for (const string& s : strs) {
 auto [zeros, ones] = countZerosOnes(s);
 if (usedM + zeros <= m && usedN + ones <= n) {
 usedM += zeros;
 usedN += ones;
 result++;
 }
}

return result;
}

int main() {
 // 测试用例 1
 vector<string> strs1 = {"10", "0001", "111001", "1", "0"};
 int m1 = 5, n1 = 3;
 cout << "测试用例 1 结果: " << findMaxForm(strs1, m1, n1) << endl; // 预期输出: 4

 // 测试用例 2
 vector<string> strs2 = {"10", "0", "1"};
 int m2 = 1, n2 = 1;
 cout << "测试用例 2 结果: " << findMaxForm(strs2, m2, n2) << endl; // 预期输出: 2

 // 测试用例 3
 vector<string> strs3 = {"00", "000"};
 int m3 = 1, n3 = 0;
 cout << "测试用例 3 结果: " << findMaxForm(strs3, m3, n3) << endl; // 预期输出: 0

 // 测试用例 4
 vector<string> strs4 = {"111", "1000", "1000", "1000"};
 int m4 = 9, n4 = 3;
 cout << "测试用例 4 结果: " << findMaxForm(strs4, m4, n4) << endl; // 预期输出: 3

 return 0;
}

```

=====

文件: Code23\_OnesAndZeros.java

=====

```
package class073;
```

```
// LeetCode 474. 一和零
```

```

// 题目描述：给你一个二进制字符串数组 strs 和两个整数 m 和 n。
// 请你找出并返回 strs 的最大子集的大小，该子集中 最多 有 m 个 0 和 n 个 1。
// 如果 x 的所有元素也是 y 的元素，集合 x 是集合 y 的 子集 。
// 链接: https://leetcode.cn/problems/ones-and-zeroes/
//
// 解题思路：
// 这是一个多维背包问题，我们需要同时考虑两种资源限制：0 的数量和 1 的数量。
// 每个字符串相当于一个物品，占用的空间是它包含的 0 的数量和 1 的数量，价值为 1（因为我们想最大化子集的大小）。
// 目标是在不超过 m 个 0 和 n 个 1 的限制下，选择尽可能多的字符串。
//
// 状态定义：dp[i][j] 表示使用 i 个 0 和 j 个 1 时，可以选择的最大字符串数量
// 状态转移方程：对于每个字符串 s，其中有 zeros 个 0 和 ones 个 1，
// dp[i][j] = max(dp[i][j], dp[i-zeros][j-ones] + 1)，当 i >= zeros 且 j >= ones 时
// 初始状态：dp[0][0] = 0，表示不使用任何 0 和 1 时，可以选择 0 个字符串
// 其他初始值也为 0，表示还没有选择任何字符串
//
// 时间复杂度：O(1 * m * n)，其中 1 是字符串数组的长度，m 和 n 是给定的整数
// 空间复杂度：O(m * n)，使用二维 DP 数组

```

```

public class Code23_OnesAndZeros {

 // 主方法，用于测试
 public static void main(String[] args) {
 // 测试用例 1
 String[] strs1 = {"10", "0001", "111001", "1", "0"};
 int m1 = 5, n1 = 3;
 System.out.println("测试用例 1 结果：" + findMaxForm(strs1, m1, n1)); // 预期输出: 4
 // 解释：最多有 5 个 0 和 3 个 1 的最大子集是 {"10", "0001", "1", "0"}
 // 这个子集最多包含 4 个 0 (来自"0001") 和 3 个 1 (来自"10", "1", "0001")。

 // 测试用例 2
 String[] strs2 = {"10", "0", "1"};
 int m2 = 1, n2 = 1;
 System.out.println("测试用例 2 结果：" + findMaxForm(strs2, m2, n2)); // 预期输出: 2
 // 解释：最大的子集是 {"0", "1"}，这两个字符串最多有 1 个 0 和 1 个 1。

 // 测试用例 3
 String[] strs3 = {"00", "000"};
 int m3 = 1, n3 = 0;
 System.out.println("测试用例 3 结果：" + findMaxForm(strs3, m3, n3)); // 预期输出: 0

 // 测试用例 4

```

```

String[] strs4 = {"111", "1000", "1000", "1000"};
int m4 = 9, n4 = 3;
System.out.println("测试用例 4 结果: " + findMaxForm(strs4, m4, n4)); // 预期输出: 3
}

/***
 * 找出最大子集的大小，该子集中最多有 m 个 0 和 n 个 1
 * @param strs 二进制字符串数组
 * @param m 最大 0 的数量
 * @param n 最大 1 的数量
 * @return 最大子集的大小
*/
public static int findMaxForm(String[] strs, int m, int n) {
 // 参数验证
 if (strs == null || strs.length == 0) {
 return 0;
 }

 // 创建二维 DP 数组, dp[i][j] 表示使用 i 个 0 和 j 个 1 时, 可以选择的最大字符串数量
 int[][] dp = new int[m + 1][n + 1];

 // 遍历每个字符串 (物品)
 for (String s : strs) {
 // 统计当前字符串中 0 和 1 的数量
 int zeros = 0, ones = 0;
 for (char c : s.toCharArray()) {
 if (c == '0') {
 zeros++;
 } else {
 ones++;
 }
 }
 }

 // 逆序遍历 m 和 n, 避免重复使用同一个字符串
 // 从大到小遍历 0 的数量
 for (int i = m; i >= zeros; i--) {
 // 从大到小遍历 1 的数量
 for (int j = n; j >= ones; j--) {
 // 状态转移: 选择当前字符串或不选择当前字符串
 dp[i][j] = Math.max(dp[i][j], dp[i - zeros][j - ones] + 1);
 }
 }
}

```

```

// 返回结果：使用最多 m 个 0 和 n 个 1 时，可以选择的最大字符串数量
return dp[m][n];
}

/**
 * 优化版本：预处理字符串的 0 和 1 数量，避免重复计算
 */
public static int findMaxFormOptimized(String[] strs, int m, int n) {
 // 参数验证
 if (strs == null || strs.length == 0) {
 return 0;
 }

 // 预处理：统计每个字符串中 0 和 1 的数量
 int[][] counts = new int[strs.length][2];
 for (int i = 0; i < strs.length; i++) {
 String s = strs[i];
 int zeros = 0, ones = 0;
 for (char c : s.toCharArray()) {
 if (c == '0') {
 zeros++;
 } else {
 ones++;
 }
 }
 counts[i][0] = zeros;
 counts[i][1] = ones;
 }

 // 创建二维 DP 数组
 int[][] dp = new int[m + 1][n + 1];

 // 遍历每个字符串（物品）
 for (int[] count : counts) {
 int zeros = count[0];
 int ones = count[1];

 // 剪枝：如果当前字符串需要的 0 或 1 超过限制，则跳过
 if (zeros > m || ones > n) {
 continue;
 }
 }
}

```

```

// 逆序遍历
for (int i = m; i >= zeros; i--) {
 for (int j = n; j >= ones; j--) {
 dp[i][j] = Math.max(dp[i][j], dp[i - zeros][j - ones] + 1);
 }
}
}

return dp[m][n];
}

/***
 * 另一种实现方式，使用滚动数组优化空间（虽然在这里空间复杂度已经是 O(m*n)，但展示了一种优化思路）
 */
public static int findMaxFormWithRollingArray(String[] strs, int m, int n) {
 // 参数验证
 if (strs == null || strs.length == 0) {
 return 0;
 }

 // 创建 DP 数组
 int[][] dp = new int[m + 1][n + 1];

 for (String s : strs) {
 int zeros = 0, ones = 0;
 for (char c : s.toCharArray()) {
 if (c == '0') zeros++;
 else ones++;
 }

 // 使用临时数组保存上一状态
 // 实际上，逆序遍历就可以避免使用临时数组，但这里展示这种方法
 int[][] temp = new int[m + 1][n + 1];
 for (int i = 0; i <= m; i++) {
 System.arraycopy(dp[i], 0, temp[i], 0, n + 1);
 }

 for (int i = zeros; i <= m; i++) {
 for (int j = ones; j <= n; j++) {
 dp[i][j] = Math.max(dp[i][j], temp[i - zeros][j - ones] + 1);
 }
 }
 }
}

```

```

 }

 return dp[m][n];
}

/***
 * 递归+记忆化搜索实现
 */
public static int findMaxFormDFS(String[] strs, int m, int n) {
 // 参数验证
 if (strs == null || strs.length == 0) {
 return 0;
 }

 // 预处理：统计每个字符串中 0 和 1 的数量
 int[][] counts = new int[strs.length][2];
 for (int i = 0; i < strs.length; i++) {
 String s = strs[i];
 int zeros = 0, ones = 0;
 for (char c : s.toCharArray()) {
 if (c == '0') {
 zeros++;
 } else {
 ones++;
 }
 }
 counts[i][0] = zeros;
 counts[i][1] = ones;
 }

 // 使用三维记忆化数组：dp[index][i][j] 表示从 index 开始，使用 i 个 0 和 j 个 1 时，可以选择的最大字符串数量
 Integer[][][] memo = new Integer[strs.length][m + 1][n + 1];

 // 调用递归辅助函数
 return dfs(counts, 0, m, n, memo);
}

/***
 * 递归辅助函数
 */
private static int dfs(int[][] counts, int index, int m, int n, Integer[][][] memo) {
 // 基础情况：已经处理完所有字符串
}

```

```

 if (index == counts.length) {
 return 0;
 }

 // 如果已经计算过，直接返回结果
 if (memo[index][m][n] != null) {
 return memo[index][m][n];
 }

 // 获取当前字符串的 0 和 1 数量
 int zeros = counts[index][0];
 int ones = counts[index][1];

 // 选择不使用当前字符串
 int notTake = dfs(counts, index + 1, m, n, memo);

 // 选择使用当前字符串（如果有足够的 0 和 1）
 int take = 0;
 if (zeros <= m && ones <= n) {
 take = 1 + dfs(counts, index + 1, m - zeros, n - ones, memo);
 }

 // 记录结果
 memo[index][m][n] = Math.max(notTake, take);

 return memo[index][m][n];
}

/**
 * 优化的 DFS 实现，避免使用过大的三维数组
 */
public static int findMaxFormDFSOptimized(String[] strs, int m, int n) {
 // 预处理：统计每个字符串中 0 和 1 的数量，并过滤掉不可能选择的字符串
 java.util.List<int[]> validCounts = new java.util.ArrayList<>();
 for (String s : strs) {
 int zeros = 0, ones = 0;
 for (char c : s.toCharArray()) {
 if (c == '0') zeros++;
 else ones++;
 }
 // 只有当字符串需要的 0 和 1 都不超过限制时，才可能被选中
 if (zeros <= m && ones <= n) {
 validCounts.add(new int[] {zeros, ones});
 }
 }
}

```

```

 }
 }

 // 使用 Map 作为缓存, 键为(index, m, n)的组合, 值为对应的最大子集大小
 java.util.Map<String, Integer> memo = new java.util.HashMap<>();

 return dfsOptimized(validCounts, 0, m, n, memo);
}

private static int dfsOptimized(java.util.List<int[]> counts, int index, int m, int n,
 java.util.Map<String, Integer> memo) {
 if (index == counts.size()) {
 return 0;
 }

 // 生成缓存键
 String key = index + "," + m + "," + n;
 if (memo.containsKey(key)) {
 return memo.get(key);
 }

 int[] current = counts.get(index);
 int zeros = current[0];
 int ones = current[1];

 // 不选择当前字符串
 int notTake = dfsOptimized(counts, index + 1, m, n, memo);

 // 选择当前字符串
 int take = 0;
 if (zeros <= m && ones <= n) {
 take = 1 + dfsOptimized(counts, index + 1, m - zeros, n - ones, memo);
 }

 // 取最大值并缓存
 int result = Math.max(notTake, take);
 memo.put(key, result);

 return result;
}

```

=====

文件: Code23\_OnesAndZeros.py

```
=====
LeetCode 474. 一和零
题目描述: 给你一个二进制字符串数组 strs 和两个整数 m 和 n 。
请你找出并返回 strs 的最大子集的大小，该子集中 最多 有 m 个 0 和 n 个 1 。
如果 x 的所有元素也是 y 的元素，集合 x 是集合 y 的 子集 。
链接: https://leetcode.cn/problems/ones-and-zeroes/
#
解题思路:
这是一个多维背包问题，我们需要同时考虑两种资源限制：0 的数量和 1 的数量。
每个字符串相当于一个物品，占用的空间是它包含的 0 的数量和 1 的数量，价值为 1（因为我们想最大化子集的大小）。
目标是在不超过 m 个 0 和 n 个 1 的限制下，选择尽可能多的字符串。
#
状态定义: dp[i][j] 表示使用 i 个 0 和 j 个 1 时，可以选择的最大字符串数量
状态转移方程: 对于每个字符串 s，其中有 zeros 个 0 和 ones 个 1,
dp[i][j] = max(dp[i][j], dp[i-zeros][j-ones] + 1)，当 i >= zeros 且 j >= ones 时
初始状态: dp[0][0] = 0，表示不使用任何 0 和 1 时，可以选择 0 个字符串
其他初始值也为 0，表示还没有选择任何字符串
#
时间复杂度: O(l * m * n)，其中 l 是字符串数组的长度，m 和 n 是给定的整数
空间复杂度: O(m * n)，使用二维 DP 数组
```

```
def find_max_form(strs, m, n):
```

```
 """

```

找出最大子集的大小，该子集中最多有 m 个 0 和 n 个 1

Args:

strs: 二进制字符串数组

m: 最大 0 的数量

n: 最大 1 的数量

Returns:

int: 最大子集的大小

```
 """

```

# 参数验证

```
if not strs:
```

```
 return 0
```

```
创建二维 DP 数组，dp[i][j] 表示使用 i 个 0 和 j 个 1 时，可以选择的最大字符串数量
```

```
dp = [[0] * (n + 1) for _ in range(m + 1)]
```

```

遍历每个字符串（物品）
for s in strs:
 # 统计当前字符串中 0 和 1 的数量
 zeros = s.count('0')
 ones = len(s) - zeros

 # 逆序遍历 m 和 n，避免重复使用同一个字符串
 # 从大到小遍历 0 的数量
 for i in range(m, zeros - 1, -1):
 # 从大到小遍历 1 的数量
 for j in range(n, ones - 1, -1):
 # 状态转移：选择当前字符串或不选择当前字符串
 dp[i][j] = max(dp[i][j], dp[i - zeros][j - ones] + 1)

返回结果：使用最多 m 个 0 和 n 个 1 时，可以选择的最大字符串数量
return dp[m][n]

```

```
def find_max_form_optimized(strs, m, n):
```

```
"""

```

优化版本：预处理字符串的 0 和 1 数量，避免重复计算

Args:

strs: 二进制字符串数组

m: 最大 0 的数量

n: 最大 1 的数量

Returns:

int: 最大子集的大小

```
"""

```

# 参数验证

```
if not strs:
```

```
 return 0
```

# 预处理：统计每个字符串中 0 和 1 的数量

```
counts = []
```

```
for s in strs:
```

```
 zeros = s.count('0')
```

```
 ones = len(s) - zeros
```

```
 counts.append((zeros, ones))
```

# 创建二维 DP 数组

```
dp = [[0] * (n + 1) for _ in range(m + 1)]
```

```

遍历每个字符串（物品）
for zeros, ones in counts:
 # 剪枝：如果当前字符串需要的 0 或 1 超过限制，则跳过
 if zeros > m or ones > n:
 continue

 # 逆序遍历
 for i in range(m, zeros - 1, -1):
 for j in range(n, ones - 1, -1):
 dp[i][j] = max(dp[i][j], dp[i - zeros][j - ones] + 1)

return dp[m][n]

```

```
def find_max_form_with_rolling_array(strs, m, n):
```

```
 """
```

另一种实现方式，使用滚动数组优化空间

Args:

- strs: 二进制字符串数组
- m: 最大 0 的数量
- n: 最大 1 的数量

Returns:

- int: 最大子集的大小

```
 """
```

# 参数验证

```
if not strs:
 return 0
```

# 创建 DP 数组

```
dp = [[0] * (n + 1) for _ in range(m + 1)]
```

for s in strs:

```
 zeros = s.count('0')
 ones = len(s) - zeros
```

# 使用临时数组保存上一状态

```
temp = [row.copy() for row in dp]
```

```
for i in range(zeros, m + 1):
```

```
 for j in range(ones, n + 1):
```

```
 dp[i][j] = max(dp[i][j], temp[i - zeros][j - ones] + 1)
```

```
return dp[m][n]
```

```
def find_max_form_dfs(strs, m, n):
```

```
 """
```

```
 递归+记忆化搜索实现
```

```
Args:
```

```
 strs: 二进制字符串数组
```

```
 m: 最大 0 的数量
```

```
 n: 最大 1 的数量
```

```
Returns:
```

```
 int: 最大子集的大小
```

```
 """
```

```
参数验证
```

```
if not strs:
```

```
 return 0
```

```
预处理: 统计每个字符串中 0 和 1 的数量, 并过滤掉不可能选择的字符串
```

```
counts = []
```

```
for s in strs:
```

```
 zeros = s.count('0')
```

```
 ones = len(s) - zeros
```

```
只有当字符串需要的 0 和 1 都不超过限制时, 才可能被选中
```

```
if zeros <= m and ones <= n:
```

```
 counts.append((zeros, ones))
```

```
使用字典作为缓存, 键为(index, remaining_m, remaining_n)的元组
```

```
memo = {}
```

```
def dfs(index, remaining_m, remaining_n):
```

```
基础情况: 已经处理完所有字符串
```

```
if index == len(counts):
```

```
 return 0
```

```
生成缓存键
```

```
key = (index, remaining_m, remaining_n)
```

```
if key in memo:
```

```
 return memo[key]
```

```
获取当前字符串的 0 和 1 数量
```

```
zeros, ones = counts[index]
```

```
选择不使用当前字符串
not_take = dfs(index + 1, remaining_m, remaining_n)

选择使用当前字符串（如果有足够的 0 和 1）
take = 0
if zeros <= remaining_m and ones <= remaining_n:
 take = 1 + dfs(index + 1, remaining_m - zeros, remaining_n - ones)

记录结果
memo[key] = max(not_take, take)

return memo[key]

调用递归函数
return dfs(0, m, n)
```

```
def find_max_form_dp_compressed(strs, m, n):
 """
```

使用一维 DP 数组的优化版本（降维）

注意：由于这里是二维背包，降维后需要使用临时数组来保存上一状态

Args:

strs: 二进制字符串数组

m: 最大 0 的数量

n: 最大 1 的数量

Returns:

int: 最大子集的大小

```
"""
```

# 实际上，对于二维背包，使用二维数组更加直观

# 这里仅作为演示，实现方式与上面基本相同

```
return find_max_form_optimized(strs, m, n)
```

```
def find_max_form_greedy(strs, m, n):
 """
```

贪心算法（仅供参考，不适用于所有情况）

贪心无法保证得到正确结果，但在某些情况下可以作为启发式方法

Args:

strs: 二进制字符串数组

m: 最大 0 的数量

n: 最大 1 的数量

Returns:

int: 可能的子集大小 (不一定是最大的)

"""

# 按照字符串长度排序, 优先选择较短的字符串

strs\_sorted = sorted(strs, key=len)

result = 0

used\_m = 0

used\_n = 0

for s in strs\_sorted:

    zeros = s.count('0')

    ones = len(s) - zeros

    if used\_m + zeros <= m and used\_n + ones <= n:

        used\_m += zeros

        used\_n += ones

        result += 1

return result

# 测试用例

if \_\_name\_\_ == "\_\_main\_\_":

# 测试用例 1

strs1 = ["10", "0001", "111001", "1", "0"]

m1, n1 = 5, 3

print(f"测试用例 1 结果: {find\_max\_form(strs1, m1, n1)}") # 预期输出: 4

# 测试用例 2

strs2 = ["10", "0", "1"]

m2, n2 = 1, 1

print(f"测试用例 2 结果: {find\_max\_form(strs2, m2, n2)}") # 预期输出: 2

# 测试用例 3

strs3 = ["00", "000"]

m3, n3 = 1, 0

print(f"测试用例 3 结果: {find\_max\_form(strs3, m3, n3)}") # 预期输出: 0

# 测试用例 4

strs4 = ["111", "1000", "1000", "1000"]

m4, n4 = 9, 3

print(f"测试用例 4 结果: {find\_max\_form(strs4, m4, n4)}") # 预期输出: 3

=====

文件: Code24\_CombinationSum4. cpp

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <unordered_map>

// LeetCode 377. 组合总和 IV
// 题目描述: 给你一个由 不同 整数组成的数组 nums , 和一个目标整数 target 。
// 请你从 nums 中找出并返回总和为 target 的元素组合的个数。
// 注意: 顺序不同的序列被视作不同的组合。
// 链接: https://leetcode.cn/problems/combination-sum-iv/
//
// 解题思路:
// 这是一个完全背包问题的变种, 但是与传统的完全背包问题不同, 这里需要计算的是排列数而不是组合数。
// 对于排列数, 我们需要先遍历容量 (target), 再遍历物品 (nums 数组), 这样可以确保不同顺序的序列被视为不同的组合。
//
// 状态定义: dp[i] 表示总和为 i 的元素组合的个数
// 状态转移方程: dp[i] += dp[i - num], 对于每个 num, 如果 i >= num
// 初始状态: dp[0] = 1, 表示总和为 0 的组合只有一种 (空组合)
//
// 时间复杂度: O(target * n), 其中 n 是 nums 数组的长度
// 空间复杂度: O(target), 使用一维 DP 数组

using namespace std;

/**
 * 找出总和为 target 的元素组合的个数
 * @param nums 不同整数组成的数组
 * @param target 目标整数
 * @return 总和为 target 的元素组合的个数
 */
int combinationSum4(vector<int>& nums, int target) {
 // 参数验证
 if (nums.empty()) {
 return 0;
 }

 // 创建一维 DP 数组, dp[i] 表示总和为 i 的元素组合的个数
 vector<long long> dp(target + 1, 0); // 使用 long long 防止溢出
 dp[0] = 1; // 初始状态: 总和为 0 的组合只有一种 (空组合)

 for (int i = 1; i <= target; ++i) {
 for (int num : nums) {
 if (i - num <= 0) {
 continue;
 }
 dp[i] += dp[i - num];
 }
 }

 return dp[target];
}
```

```

// 初始状态: 总和为 0 的组合只有一种 (空组合)
dp[0] = 1;

// 注意: 为了计算排列数, 我们先遍历容量 (target), 再遍历物品 (nums 数组)
// 这样可以确保不同顺序的序列被视为不同的组合
for (int i = 1; i <= target; i++) {
 for (int num : nums) {
 // 状态转移: 如果当前容量 i 大于等于物品重量 num
 if (i >= num && dp[i - num] <= INT_MAX - dp[i]) {
 dp[i] += dp[i - num];
 }
 }
}

// 返回结果: 总和为 target 的元素组合的个数
return dp[target] > INT_MAX ? INT_MAX : static_cast<int>(dp[target]);
}

```

```

/**
 * 优化版本: 剪枝处理
 */
int combinationSum4Optimized(vector<int>& nums, int target) {
 // 参数验证
 if (nums.empty()) {
 return 0;
 }

 // 排序 nums 数组, 方便后续剪枝
 sort(nums.begin(), nums.end());

 // 创建一维 DP 数组
 vector<long long> dp(target + 1, 0);
 dp[0] = 1;

 // 先遍历容量, 再遍历物品
 for (int i = 1; i <= target; i++) {
 for (int num : nums) {
 // 剪枝: 如果 num 大于 i, 后续的 num 会更大, 不需要继续遍历
 if (num > i) {
 break;
 }
 // 防止整数溢出
 if (dp[i - num] <= INT_MAX - dp[i]) {

```

```

 dp[i] += dp[i - num];
 }
}

return dp[target] > INT_MAX ? INT_MAX : static_cast<int>(dp[target]);
}

/***
 * 递归+记忆化搜索实现
 */
int combinationSum4DFS(vector<int>& nums, int target) {
 // 参数验证
 if (nums.empty()) {
 return 0;
 }

 // 使用 unordered_map 作为缓存
 unordered_map<int, long long> memo; // 键为剩余目标值，值为对应的组合数

 // 定义 DFS 函数
 function<long long(int)> dfs = [&](int remaining) {
 // 基础情况：剩余目标值为 0，返回 1（表示找到一种组合）
 if (remaining == 0) {
 return 1LL;
 }

 // 基础情况：剩余目标值小于 0，返回 0（表示无法找到组合）
 if (remaining < 0) {
 return 0LL;
 }

 // 检查缓存
 if (memo.find(remaining) != memo.end()) {
 return memo[remaining];
 }

 // 计算所有可能的组合数
 long long count = 0;
 for (int num : nums) {
 // 递归计算使用当前 num 后的组合数
 count += dfs(remaining - num);
 // 防止整数溢出
 }
 };
}

```

```

 if (count > INT_MAX) {
 break;
 }
 }

 // 缓存结果
 memo[remaining] = count;

 return count;
};

// 调用递归函数
long long result = dfs(target);
return result > INT_MAX ? INT_MAX : static_cast<int>(result);
}

/***
 * 使用数组作为缓存的优化 DFS 实现
 */
int combinationSum4DFSWithArrayCache(vector<int>& nums, int target) {
 // 参数验证
 if (nums.empty()) {
 return 0;
 }

 // 使用 vector 作为缓存，初始值为-1 表示未计算
 vector<long long> cache(target + 1, -1);

 // 定义 DFS 函数
 function<long long(int)> dfs = [&](int remaining) {
 // 基础情况
 if (remaining == 0) {
 return 1LL;
 }
 if (remaining < 0) {
 return 0LL;
 }

 // 检查缓存
 if (cache[remaining] != -1) {
 return cache[remaining];
 }

 long long result = 0;
 for (int num : nums) {
 result += dfs(remaining - num);
 }
 cache[remaining] = result;
 return result;
 };
}

```

```

// 计算组合数
long long count = 0;
for (int num : nums) {
 count += dfs(remaining - num);
 if (count > INT_MAX) {
 break;
 }
}

// 缓存结果
cache[remaining] = count;

return count;
};

long long result = dfs(target);
return result > INT_MAX ? INT_MAX : static_cast<int>(result);
}

/***
 * 回溯算法实现（注意：对于大数会超时，仅作为参考）
 */
int combinationSum4Backtracking(vector<int>& nums, int target) {
 // 参数验证
 if (nums.empty()) {
 return 0;
 }

 // 结果计数器
 long long count = 0;

 // 定义回溯函数
 function<void(int)> backtrack = [&](int remaining) {
 // 找到一个有效组合
 if (remaining == 0) {
 count++;
 return;
 }

 // 超过目标值，直接返回
 if (remaining < 0 || count > INT_MAX) {
 return;
 }

 for (int num : nums) {
 remaining -= num;
 backtrack(remaining);
 remaining += num;
 }
 };

 backtrack(target);
 return count;
}

```

```

// 尝试每个数字
for (int num : nums) {
 backtrack(remaining - num);
}
};

// 调用回溯函数
backtrack(target);

return count > INT_MAX ? INT_MAX : static_cast<int>(count);
}

/***
 * 使用 BFS 的实现方式
 */
int combinationSum4BFS(vector<int>& nums, int target) {
 if (nums.empty()) {
 return 0;
 }

 // dp[i] 表示总和为 i 的元素组合的个数
 vector<long long> dp(target + 1, 0);
 dp[0] = 1;

 // BFS 思想：从小到大计算每个值的组合数
 for (int i = 0; i <= target; i++) {
 // 如果当前值 i 无法达到，跳过
 if (dp[i] == 0) {
 continue;
 }

 // 尝试在当前值的基础上添加每个数字
 for (int num : nums) {
 // 确保不会超出 target
 if (i + num <= target) {
 // 防止溢出
 if (dp[i] <= INT_MAX - dp[i + num]) {
 dp[i + num] += dp[i];
 }
 }
 }
 }
}

```

```

 return dp[target] > INT_MAX ? INT_MAX : static_cast<int>(dp[target]);
}

int main() {
 // 测试用例 1
 vector<int> nums1 = {1, 2, 3};
 int target1 = 4;
 cout << "测试用例 1 结果: " << combinationSum4(nums1, target1) << endl; // 预期输出: 7

 // 测试用例 2
 vector<int> nums2 = {9};
 int target2 = 3;
 cout << "测试用例 2 结果: " << combinationSum4(nums2, target2) << endl; // 预期输出: 0

 // 测试用例 3
 vector<int> nums3 = {1, 2, 4};
 int target3 = 32;
 cout << "测试用例 3 结果: " << combinationSum4(nums3, target3) << endl; // 大数测试

 // 测试用例 4
 vector<int> nums4 = {1, 50};
 int target4 = 100;
 cout << "测试用例 4 结果: " << combinationSum4(nums4, target4) << endl; // 预期输出: 3

 return 0;
}

```

=====

文件: Code24\_CombinationSum4.java

=====

```

package class073;

// LeetCode 377. 组合总和 IV
// 题目描述: 给你一个由 不同 整数组成的数组 nums，和一个目标整数 target。
// 请你从 nums 中找出并返回总和为 target 的元素组合的个数。
// 注意: 顺序不同的序列被视作不同的组合。
// 链接: https://leetcode.cn/problems/combination-sum-iv/
//
// 解题思路:
// 这是一个完全背包问题的变种，但是与传统的完全背包问题不同，这里需要计算的是排列数而不是组合数。
// 对于排列数，我们需要先遍历容量 (target)，再遍历物品 (nums 数组)，这样可以确保不同顺序的序列被视

```

为不同的组合。

```
// 状态定义: dp[i] 表示总和为 i 的元素组合的个数
// 状态转移方程: dp[i] += dp[i - num], 对于每个 num, 如果 i >= num
// 初始状态: dp[0] = 1, 表示总和为 0 的组合只有一种 (空组合)
//
// 时间复杂度: O(target * n), 其中 n 是 nums 数组的长度
// 空间复杂度: O(target), 使用一维 DP 数组

public class Code24_CombinationSum4 {

 // 主方法, 用于测试
 public static void main(String[] args) {
 // 测试用例 1
 int[] nums1 = {1, 2, 3};
 int target1 = 4;
 System.out.println("测试用例 1 结果: " + combinationSum4(nums1, target1)); // 预期输出: 7
 // 解释: 所有可能的组合为:
 // (1, 1, 1, 1)
 // (1, 1, 2)
 // (1, 2, 1)
 // (1, 3)
 // (2, 1, 1)
 // (2, 2)
 // (3, 1)

 // 测试用例 2
 int[] nums2 = {9};
 int target2 = 3;
 System.out.println("测试用例 2 结果: " + combinationSum4(nums2, target2)); // 预期输出: 0

 // 测试用例 3
 int[] nums3 = {1, 2, 4};
 int target3 = 32;
 System.out.println("测试用例 3 结果: " + combinationSum4(nums3, target3)); // 大数测试

 // 测试用例 4
 int[] nums4 = {1, 50};
 int target4 = 100;
 System.out.println("测试用例 4 结果: " + combinationSum4(nums4, target4)); // 预期输出: 3
 }

 /**

```

```

* 找出总和为 target 的元素组合的个数
* @param nums 不同整数组成的数组
* @param target 目标整数
* @return 总和为 target 的元素组合的个数
*/
public static int combinationSum4(int[] nums, int target) {
 // 参数验证
 if (nums == null || nums.length == 0) {
 return 0;
 }

 // 创建一维 DP 数组, dp[i] 表示总和为 i 的元素组合的个数
 int[] dp = new int[target + 1];

 // 初始状态: 总和为 0 的组合只有一种 (空组合)
 dp[0] = 1;

 // 注意: 为了计算排列数, 我们先遍历容量 (target), 再遍历物品 (nums 数组)
 // 这样可以确保不同顺序的序列被视为不同的组合
 for (int i = 1; i <= target; i++) {
 for (int num : nums) {
 // 状态转移: 如果当前容量 i 大于等于物品重量 num
 if (i >= num) {
 // 防止整数溢出
 if (dp[i] > Integer.MAX_VALUE - dp[i - num]) {
 // 处理溢出情况
 continue;
 }
 dp[i] += dp[i - num];
 }
 }
 }

 // 返回结果: 总和为 target 的元素组合的个数
 return dp[target];
}

/**
 * 优化版本: 剪枝处理
 */
public static int combinationSum4Optimized(int[] nums, int target) {
 // 参数验证
 if (nums == null || nums.length == 0) {

```

```
 return 0;
 }

 // 排序 nums 数组，方便后续剪枝
 java.util.Arrays.sort(nums);

 // 创建一维 DP 数组
 int[] dp = new int[target + 1];
 dp[0] = 1;

 // 先遍历容量，再遍历物品
 for (int i = 1; i <= target; i++) {
 for (int num : nums) {
 // 剪枝：如果 num 大于 i，后续的 num 会更大，不需要继续遍历
 if (num > i) {
 break;
 }
 // 防止整数溢出
 if (dp[i] > Integer.MAX_VALUE - dp[i - num]) {
 continue;
 }
 dp[i] += dp[i - num];
 }
 }

 return dp[target];
}

/**
 * 递归+记忆化搜索实现
 */
public static int combinationSum4DFS(int[] nums, int target) {
 // 参数验证
 if (nums == null || nums.length == 0) {
 return 0;
 }

 // 使用 HashMap 作为缓存
 java.util.Map<Integer, Integer> memo = new java.util.HashMap<>();

 // 调用递归辅助函数
 return dfs(nums, target, memo);
}
```

```
/**
 * 递归辅助函数
 * @param nums 数组
 * @param remaining 剩余需要达到的目标值
 * @param memo 缓存，键为剩余目标值，值为对应的组合数
 * @return 达到剩余目标值的组合数
 */
private static int dfs(int[] nums, int remaining, java.util.Map<Integer, Integer> memo) {
 // 基础情况：剩余目标值为 0，返回 1（表示找到一种组合）
 if (remaining == 0) {
 return 1;
 }

 // 基础情况：剩余目标值小于 0，返回 0（表示无法找到组合）
 if (remaining < 0) {
 return 0;
 }

 // 检查缓存
 if (memo.containsKey(remaining)) {
 return memo.get(remaining);
 }

 // 计算所有可能的组合数
 int count = 0;
 for (int num : nums) {
 // 递归计算使用当前 num 后的组合数
 int result = dfs(nums, remaining - num, memo);
 // 防止整数溢出
 if (count > Integer.MAX_VALUE - result) {
 continue;
 }
 count += result;
 }

 // 缓存结果
 memo.put(remaining, count);

 return count;
}
/**
```

```
* 使用数组作为缓存的优化 DFS 实现
*/
public static int combinationSum4DFSWithArrayCache(int[] nums, int target) {
 // 参数验证
 if (nums == null || nums.length == 0) {
 return 0;
 }

 // 使用数组作为缓存
 Integer[] cache = new Integer[target + 1];

 // 调用递归辅助函数
 return dfsWithArrayCache(nums, target, cache);
}

private static int dfsWithArrayCache(int[] nums, int remaining, Integer[] cache) {
 // 基础情况
 if (remaining == 0) {
 return 1;
 }
 if (remaining < 0) {
 return 0;
 }

 // 检查缓存
 if (cache[remaining] != null) {
 return cache[remaining];
 }

 // 计算组合数
 int count = 0;
 for (int num : nums) {
 int result = dfsWithArrayCache(nums, remaining - num, cache);
 if (count > Integer.MAX_VALUE - result) {
 continue;
 }
 count += result;
 }

 // 缓存结果
 cache[remaining] = count;

 return count;
}
```

```
}

/**
 * 使用 long 类型防止溢出的版本
 * 注意：LeetCode 题目的测试用例可能会导致整数溢出
 */
public static int combinationSum4WithLong(int[] nums, int target) {
 if (nums == null || nums.length == 0) {
 return 0;
 }

 // 使用 long 类型的 DP 数组防止溢出
 long[] dp = new long[target + 1];
 dp[0] = 1;

 for (int i = 1; i <= target; i++) {
 for (int num : nums) {
 if (i >= num) {
 dp[i] += dp[i - num];
 }
 }
 // 如果结果超过 Integer.MAX_VALUE，可能会溢出
 if (dp[i] > Integer.MAX_VALUE) {
 // 根据题目要求处理
 // 这里简单返回 Integer.MAX_VALUE
 return Integer.MAX_VALUE;
 }
 }

 return (int) dp[target];
}

/**
 * 回溯算法实现（注意：对于大数会超时，仅作为参考）
 */
public static int combinationSum4Backtracking(int[] nums, int target) {
 // 参数验证
 if (nums == null || nums.length == 0) {
 return 0;
 }

 // 结果计数器
 java.util.concurrent.atomic.AtomicInteger count = new
```

```

java.util.concurrent.atomic.AtomicInteger(0);

 // 调用回溯函数
 backtrack(nums, target, count);

 return count.get();
}

private static void backtrack(int[] nums, int remaining,
java.util.concurrent.atomic.AtomicInteger count) {
 // 找到一个有效组合
 if (remaining == 0) {
 count.incrementAndGet();
 return;
 }

 // 超过目标值，直接返回
 if (remaining < 0) {
 return;
 }

 // 尝试每个数字
 for (int num : nums) {
 backtrack(nums, remaining - num, count);
 }
}

```

文件: Code24\_CombinationSum4.py

```

LeetCode 377. 组合总和 IV
题目描述: 给你一个由 不同 整数组成的数组 nums ，和一个目标整数 target 。
请你从 nums 中找出并返回总和为 target 的元素组合的个数。
注意: 顺序不同的序列被视作不同的组合。
链接: https://leetcode.cn/problems/combination-sum-iv/
#
解题思路:
这是一个完全背包问题的变种，但是与传统的完全背包问题不同，这里需要计算的是排列数而不是组合数。
对于排列数，我们需要先遍历容量 (target)，再遍历物品 (nums 数组)，这样可以确保不同顺序的序列被视为不同的组合。
#

```

```
状态定义: dp[i] 表示总和为 i 的元素组合的个数
状态转移方程: dp[i] += dp[i - num], 对于每个 num, 如果 i >= num
初始状态: dp[0] = 1, 表示总和为 0 的组合只有一种 (空组合)
#
时间复杂度: O(target * n), 其中 n 是 nums 数组的长度
空间复杂度: O(target), 使用一维 DP 数组
```

```
def combination_sum4(nums, target):
```

```
 """

```

```
 找出总和为 target 的元素组合的个数

```

```
Args:
```

```
 nums: 不同整数组成的数组

```

```
 target: 目标整数

```

```
Returns:
```

```
 int: 总和为 target 的元素组合的个数
"""

```

```
参数验证
if not nums:
 return 0

```

```
创建一维 DP 数组, dp[i] 表示总和为 i 的元素组合的个数
dp = [0] * (target + 1)
```

```
初始状态: 总和为 0 的组合只有一种 (空组合)
dp[0] = 1
```

```
注意: 为了计算排列数, 我们先遍历容量 (target), 再遍历物品 (nums 数组)
这样可以确保不同顺序的序列被视为不同的组合
for i in range(1, target + 1):
 for num in nums:
 # 状态转移: 如果当前容量 i 大于等于物品重量 num
 if i >= num:
 dp[i] += dp[i - num]

```

```
返回结果: 总和为 target 的元素组合的个数
return dp[target]
```

```
def combination_sum4_optimized(nums, target):
```

```
 """

```

```
 优化版本: 剪枝处理

```

Args:

    nums: 不同整数组成的数组

    target: 目标整数

Returns:

    int: 总和为 target 的元素组合的个数

"""

# 参数验证

if not nums:

    return 0

# 排序 nums 数组, 方便后续剪枝

nums.sort()

# 创建一维 DP 数组

dp = [0] \* (target + 1)

dp[0] = 1

# 先遍历容量, 再遍历物品

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

    for num in nums:

        # 剪枝: 如果 num 大于 i, 后续的 num 会更大, 不需要继续遍历

        if num > i:

            break

        dp[i] += dp[i - num]

return dp[target]

def combination\_sum4\_dfs(nums, target):

"""

递归+记忆化搜索实现

Args:

    nums: 不同整数组成的数组

    target: 目标整数

Returns:

    int: 总和为 target 的元素组合的个数

"""

# 参数验证

if not nums:

    return 0

```
使用字典作为缓存
memo = {}

def dfs(remaining):
 """
 递归辅助函数

 Args:
 remaining: 剩余需要达到的目标值

 Returns:
 int: 达到剩余目标值的组合数
 """
 # 基础情况: 剩余目标值为 0, 返回 1 (表示找到一种组合)
 if remaining == 0:
 return 1

 # 基础情况: 剩余目标值小于 0, 返回 0 (表示无法找到组合)
 if remaining < 0:
 return 0

 # 检查缓存
 if remaining in memo:
 return memo[remaining]

 # 计算所有可能的组合数
 count = 0
 for num in nums:
 # 递归计算使用当前 num 后的组合数
 count += dfs(remaining - num)

 # 缓存结果
 memo[remaining] = count

 return count

调用递归函数
return dfs(target)

def combination_sum4_dfs_with_lru_cache(nums, target):
 """
 使用 functools.lru_cache 的优化 DFS 实现
 """
```

Args:

nums: 不同整数组成的数组

target: 目标整数

Returns:

int: 总和为 target 的元素组合的个数

"""

```
from functools import lru_cache
```

# 参数验证

```
if not nums:
```

```
 return 0
```

# 使用 lru\_cache 装饰器缓存结果

```
@lru_cache(maxsize=None)
```

```
def dfs(remaining):
```

# 基础情况

```
 if remaining == 0:
```

```
 return 1
```

```
 if remaining < 0:
```

```
 return 0
```

# 计算组合数

```
count = 0
```

```
for num in nums:
```

```
 count += dfs(remaining - num)
```

```
return count
```

```
result = dfs(target)
```

# 清除缓存

```
dfs.cache_clear()
```

```
return result
```

```
def combination_sum4_backtracking(nums, target):
```

"""

回溯算法实现（注意：对于大数会超时，仅作为参考）

Args:

nums: 不同整数组成的数组

target: 目标整数

Returns:

```
int: 总和为 target 的元素组合的个数
"""
参数验证
if not nums:
 return 0

结果计数器
count = 0

def backtrack(remaining):
 nonlocal count
 # 找到一个有效组合
 if remaining == 0:
 count += 1
 return
 # 超过目标值，直接返回
 if remaining < 0:
 return
 # 尝试每个数字
 for num in nums:
 backtrack(remaining - num)

调用回溯函数
backtrack(target)

return count
```

```
def combination_sum4_bfs(nums, target):
```

```
"""
使用 BFS 的实现方式
```

Args:

nums: 不同整数组成的数组

target: 目标整数

Returns:

int: 总和为 target 的元素组合的个数
"""
if not nums:
 return 0

```
dp[i] 表示总和为 i 的元素组合的个数
dp = [0] * (target + 1)
dp[0] = 1

BFS 思想：从小到大计算每个值的组合数
for i in range(target + 1):
 # 如果当前值 i 无法达到，跳过
 if dp[i] == 0:
 continue

 # 尝试在当前值的基础上添加每个数字
 for num in nums:
 # 确保不会超出 target
 if i + num <= target:
 dp[i + num] += dp[i]

return dp[target]
```

```
def combination_sum4_with_overflow_check(nums, target):
 """
 包含溢出检查的版本

```

Args:

```
 nums: 不同整数组成的数组
 target: 目标整数
```

Returns:

```
 int: 总和为 target 的元素组合的个数
 """

```

```
import sys
```

```
if not nums:
 return 0
```

```
dp = [0] * (target + 1)
dp[0] = 1

for i in range(1, target + 1):
 for num in nums:
 if i >= num:
 # 检查溢出
 if dp[i] > sys.maxsize - dp[i - num]:
 return -1 # 表示溢出
```

```

dp[i] += dp[i - num]

return dp[target]

测试用例
if __name__ == "__main__":
 # 测试用例 1
 nums1 = [1, 2, 3]
 target1 = 4
 print(f"测试用例 1 结果: {combination_sum4(nums1, target1)}") # 预期输出: 7

 # 测试用例 2
 nums2 = [9]
 target2 = 3
 print(f"测试用例 2 结果: {combination_sum4(nums2, target2)}") # 预期输出: 0

 # 测试用例 3
 nums3 = [1, 2, 4]
 target3 = 32
 print(f"测试用例 3 结果: {combination_sum4(nums3, target3)}") # 大数测试

 # 测试用例 4
 nums4 = [1, 50]
 target4 = 100
 print(f"测试用例 4 结果: {combination_sum4(nums4, target4)}") # 预期输出: 3

```

=====

文件: Code25\_WordBreak.cpp

=====

```

#include <iostream>
#include <vector>
#include <string>
#include <unordered_set>
#include <unordered_map>
#include <queue>
#include <algorithm>

// LeetCode 139. 单词拆分
// 题目描述: 给你一个字符串 s 和一个字符串列表 wordDict 作为字典。请你判断是否可以利用字典中出现的单词拼接出 s。
// 注意: 不要求字典中出现的单词全部都使用，并且字典中的单词可以重复使用。
// 链接: https://leetcode.cn/problems/word-break/

```

```
//
// 解题思路:
// 这是一个完全背包问题的变种，其中：
// - 背包容量：字符串 s 的长度
// - 物品：字典中的单词
// - 问题转化为：是否可以选择一些单词（可重复），恰好拼接成字符串 s
//
// 状态定义：dp[i] 表示字符串 s 的前 i 个字符是否可以被拆分
// 状态转移方程：对于每个位置 i，遍历所有位置 j (j < i)，如果 dp[j] 为 true 且 s[j:i] 在字典中，则
dp[i] 为 true
// 初始状态：dp[0] = true，表示空字符串可以被拆分
//
// 时间复杂度：O(n^3)，其中 n 是字符串 s 的长度（需要两层循环，并且每次需要判断子字符串是否在字典
中）
// 空间复杂度：O(n)，使用一维 DP 数组
```

```
using namespace std;
```

```
/**
```

```
* 判断是否可以利用字典中出现的单词拼接出 s
* @param s 目标字符串
* @param wordDict 单词字典
* @return 是否可以拼接出 s
*/
```

```
bool wordBreak(string s, vector<string>& wordDict) {
```

```
 // 参数验证
 if (s.empty()) {
 return false; // 空字符串返回 false
 }
```

```
 // 将 wordDict 转换为 unordered_set，提高查找效率
 unordered_set<string> wordSet(wordDict.begin(), wordDict.end());
```

```
 // 获取字典中单词的最大长度，用于后续剪枝
```

```
 int maxWordLength = 0;
 for (const string& word : wordDict) {
 maxWordLength = max(maxWordLength, static_cast<int>(word.length()));
 }
```

```
 // 创建一维 DP 数组，dp[i] 表示字符串 s 的前 i 个字符是否可以被拆分
 vector<bool> dp(s.length() + 1, false);
```

```
 // 初始状态：空字符串可以被拆分
```

```

dp[0] = true;

// 遍历字符串 s 的每个位置 i
for (int i = 1; i <= s.length(); i++) {
 // 遍历之前的位置 j, 从 max(0, i - maxWordLength) 到 i-1
 // 这样可以避免检查过长的子字符串
 int start = max(0, i - maxWordLength);
 for (int j = start; j < i; j++) {
 // 状态转移: 如果前 j 个字符可以被拆分, 且子字符串 s[j:i] 在字典中, 则前 i 个字符可以被拆
 分
 if (dp[j] && wordSet.count(s.substr(j, i - j))) {
 dp[i] = true;
 break; // 只要找到一种拆分方式即可
 }
 }
}

```

```

// 返回结果: 整个字符串 s 是否可以被拆分
return dp[s.length()];
}

/***
 * 优化版本: 移除 maxWordLength 的计算, 直接使用普通的两层循环
 */
bool wordBreakOptimized(string s, vector<string>& wordDict) {
 // 参数验证
 if (s.empty()) {
 return false;
 }

 // 将 wordDict 转换为 unordered_set, 提高查找效率
 unordered_set<string> wordSet(wordDict.begin(), wordDict.end());

 // 创建一维 DP 数组
 vector<bool> dp(s.length() + 1, false);
 dp[0] = true;

 // 遍历字符串 s 的每个位置 i
 for (int i = 1; i <= s.length(); i++) {
 // 遍历之前的位置 j
 for (int j = 0; j < i; j++) {
 // 状态转移
 if (dp[j] && wordSet.count(s.substr(j, i - j))) {

```

```

 dp[i] = true;
 break; // 只要找到一种拆分方式即可
 }
}

return dp[s.length()];
}

/***
 * 另一种实现方式：先遍历字典中的单词，再遍历字符串位置
 * 这更符合完全背包问题的思路
 */
bool wordBreakKnapsackStyle(string s, vector<string>& wordDict) {
 // 参数验证
 if (s.empty()) {
 return false;
 }

 // 创建一维 DP 数组
 vector<bool> dp(s.length() + 1, false);
 dp[0] = true;

 // 先遍历容量（字符串长度）
 for (int i = 1; i <= s.length(); i++) {
 // 再遍历物品（字典中的单词）
 for (const string& word : wordDict) {
 int wordLength = word.length();
 // 如果当前位置 i 大于等于单词长度，并且前 i-wordLength 个字符可以被拆分
 // 并且子字符串 s[i-wordLength:i] 等于当前单词
 if (i >= wordLength && dp[i - wordLength] &&
 s.substr(i - wordLength, wordLength) == word) {
 dp[i] = true;
 break; // 只要找到一种拆分方式即可
 }
 }
 }

 return dp[s.length()];
}

/***
 * 递归+记忆化搜索实现

```

```
/*
bool wordBreakDFS(string s, vector<string>& wordDict) {
 // 参数验证
 if (s.empty()) {
 return false;
 }

 // 将 wordDict 转换为 unordered_set，提高查找效率
 unordered_set<string> wordSet(wordDict.begin(), wordDict.end());

 // 使用 unordered_map 作为缓存，键为子字符串的起始索引，值为该子字符串是否可以被拆分
 unordered_map<int, bool> memo;

 // 定义 DFS 函数
 function<bool(int)> dfs = [&](int start) {
 // 基础情况：如果已经到达字符串末尾，表示成功拆分
 if (start == s.length()) {
 return true;
 }

 // 检查缓存
 if (memo.find(start) != memo.end()) {
 return memo[start];
 }

 // 尝试从 start 位置开始的所有可能的子字符串
 for (int end = start + 1; end <= s.length(); end++) {
 // 如果子字符串在字典中，并且剩余部分也可以被拆分
 if (wordSet.count(s.substr(start, end - start)) && dfs(end)) {
 memo[start] = true;
 return true;
 }
 }

 // 所有可能的拆分方式都失败了
 memo[start] = false;
 return false;
 };

 // 调用递归函数
 return dfs(0);
}
```

```
/***
 * BFS 实现
 */
bool wordBreakBFS(string s, vector<string>& wordDict) {
 // 参数验证
 if (s.empty()) {
 return false;
 }

 // 将 wordDict 转换为 unordered_set，提高查找效率
 unordered_set<string> wordSet(wordDict.begin(), wordDict.end());

 // 创建队列，存储可以拆分到的位置
 queue<int> q;
 // 标记已经访问过的位置，避免重复处理
 vector<bool> visited(s.length(), false);

 // 初始位置为 0
 q.push(0);
 visited[0] = true;

 while (!q.empty()) {
 int start = q.front();
 q.pop();

 // 尝试从 start 位置开始的所有可能的子字符串
 for (int end = start + 1; end <= s.length(); end++) {
 // 如果子字符串在字典中
 if (wordSet.count(s.substr(start, end - start))) {
 // 如果已经到达字符串末尾，表示成功拆分
 if (end == s.length()) {
 return true;
 }
 // 如果该位置尚未访问过，将其加入队列
 if (!visited[end]) {
 q.push(end);
 visited[end] = true;
 }
 }
 }
 }

 // 队列为空仍未返回 true，表示无法拆分
}
```

```

 return false;
}

/***
 * Trie 树节点类
 */
class TrieNode {
public:
 bool isEnd;
 unordered_map<char, TrieNode*> children;

 TrieNode() : isEnd(false) {}

 ~TrieNode() {
 for (auto& pair : children) {
 delete pair.second;
 }
 }
};

/***
 * 构建 Trie 树
 */
TrieNode* buildTrie(const vector<string>& wordDict) {
 TrieNode* root = new TrieNode();
 for (const string& word : wordDict) {
 TrieNode* node = root;
 for (char c : word) {
 if (node->children.find(c) == node->children.end()) {
 node->children[c] = new TrieNode();
 }
 node = node->children[c];
 }
 node->isEnd = true;
 }
 return root;
}

/***
 * 使用 Trie 树优化查找效率
 */
bool wordBreakWithTrie(string s, vector<string>& wordDict) {
 // 参数验证

```

```

if (s.empty()) {
 return false;
}

// 构建 Trie 树
TrieNode* root = buildTrie(wordDict);

// 创建一维 DP 数组
vector<bool> dp(s.length() + 1, false);
dp[0] = true;

// 遍历字符串 s 的每个位置 i
for (int i = 0; i < s.length(); i++) {
 // 如果前 i 个字符无法被拆分，跳过
 if (!dp[i]) {
 continue;
 }

 // 从 i 位置开始，在 Trie 树中查找可能的单词
 TrieNode* node = root;
 for (int j = i; j < s.length(); j++) {
 char c = s[j];
 if (node->children.find(c) == node->children.end()) {
 break; // 无法继续匹配
 }
 node = node->children[c];
 // 如果找到一个单词，标记 dp[j+1] 为 true
 if (node->isEnd) {
 dp[j + 1] = true;
 }
 }
}

// 释放 Trie 树内存
delete root;

return dp[s.length()];
}

int main() {
 // 测试用例 1
 string s1 = "leetcode";
 vector<string> wordDict1 = {"leet", "code"};

```

```

cout << "测试用例 1 结果: " << (wordBreak(s1, wordDict1) ? "true" : "false") << endl; // 预期
输出: true

// 测试用例 2
string s2 = "applepenapple";
vector<string> wordDict2 = {"apple", "pen"};
cout << "测试用例 2 结果: " << (wordBreak(s2, wordDict2) ? "true" : "false") << endl; // 预期
输出: true

// 测试用例 3
string s3 = "catsandog";
vector<string> wordDict3 = {"cats", "dog", "sand", "and", "cat"};
cout << "测试用例 3 结果: " << (wordBreak(s3, wordDict3) ? "true" : "false") << endl; // 预期
输出: false

// 测试用例 4
string s4 = "";
vector<string> wordDict4 = {"a"};
cout << "测试用例 4 结果: " << (wordBreak(s4, wordDict4) ? "true" : "false") << endl; // 预期
输出: false

return 0;
}

```

=====

文件: Code25\_WordBreak.java

=====

```

package class073;

import java.util.*;

// LeetCode 139. 单词拆分
// 题目描述: 给你一个字符串 s 和一个字符串列表 wordDict 作为字典。请你判断是否可以利用字典中出现
// 的单词拼接出 s 。
// 注意: 不要求字典中出现的单词全部都使用, 并且字典中的单词可以重复使用。
// 链接: https://leetcode.cn/problems/word-break/
//
// 解题思路:
// 这是一个完全背包问题的变种, 其中:
// - 背包容量: 字符串 s 的长度
// - 物品: 字典中的单词
// - 问题转化为: 是否可以选择一些单词 (可重复), 恰好拼接成字符串 s

```

```

// 状态定义: dp[i] 表示字符串 s 的前 i 个字符是否可以被拆分
// 状态转移方程: 对于每个位置 i, 遍历所有位置 j (j < i), 如果 dp[j] 为 true 且 s[j:i] 在字典中, 则
dp[i] 为 true
// 初始状态: dp[0] = true, 表示空字符串可以被拆分
//
// 时间复杂度: O(n^3), 其中 n 是字符串 s 的长度 (需要两层循环, 并且每次需要判断子字符串是否在字典
中)
// 空间复杂度: O(n), 使用一维 DP 数组

public class Code25_WordBreak {

 // 主方法, 用于测试
 public static void main(String[] args) {
 // 测试用例 1
 String s1 = "leetcode";
 List<String> wordDict1 = Arrays.asList("leet", "code");
 System.out.println("测试用例 1 结果: " + wordBreak(s1, wordDict1)); // 预期输出: true

 // 测试用例 2
 String s2 = "applepenapple";
 List<String> wordDict2 = Arrays.asList("apple", "pen");
 System.out.println("测试用例 2 结果: " + wordBreak(s2, wordDict2)); // 预期输出: true

 // 测试用例 3
 String s3 = "catsandog";
 List<String> wordDict3 = Arrays.asList("cats", "dog", "sand", "and", "cat");
 System.out.println("测试用例 3 结果: " + wordBreak(s3, wordDict3)); // 预期输出: false

 // 测试用例 4
 String s4 = "";
 List<String> wordDict4 = Arrays.asList("a");
 System.out.println("测试用例 4 结果: " + wordBreak(s4, wordDict4)); // 预期输出: false
 }

 /**
 * 判断是否可以利用字典中出现的单词拼接出 s
 * @param s 目标字符串
 * @param wordDict 单词字典
 * @return 是否可以拼接出 s
 */
 public static boolean wordBreak(String s, List<String> wordDict) {
 // 参数验证

```

```

if (s == null || s.length() == 0) {
 return false; // 题目中说明 s 非空? 根据测试用例 4, 空字符串应该返回 false
}

// 将 wordDict 转换为 Set, 提高查找效率
Set<String> wordSet = new HashSet<>(wordDict);

// 获取字典中单词的最大长度, 用于后续剪枝
int maxWordLength = 0;
for (String word : wordDict) {
 maxWordLength = Math.max(maxWordLength, word.length());
}

// 创建一维 DP 数组, dp[i] 表示字符串 s 的前 i 个字符是否可以被拆分
boolean[] dp = new boolean[s.length() + 1];

// 初始状态: 空字符串可以被拆分
dp[0] = true;

// 遍历字符串 s 的每个位置 i
for (int i = 1; i <= s.length(); i++) {
 // 遍历之前的位置 j, 从 max(0, i - maxWordLength) 到 i-1
 // 这样可以避免检查过长的子字符串
 for (int j = Math.max(0, i - maxWordLength); j < i; j++) {
 // 状态转移: 如果前 j 个字符可以被拆分, 且子字符串 s[j:i] 在字典中, 则前 i 个字符可以
被拆分
 if (dp[j] && wordSet.contains(s.substring(j, i))) {
 dp[i] = true;
 break; // 只要找到一种拆分方式即可
 }
 }
}

// 返回结果: 整个字符串 s 是否可以被拆分
return dp[s.length()];
}

/**
 * 优化版本: 移除 maxWordLength 的计算, 直接使用普通的两层循环
 */
public static boolean wordBreakOptimized(String s, List<String> wordDict) {
 // 参数验证
 if (s == null || s.length() == 0) {

```

```

 return false;
 }

 // 将 wordDict 转换为 Set，提高查找效率
 Set<String> wordSet = new HashSet<>(wordDict);

 // 创建一维 DP 数组
 boolean[] dp = new boolean[s.length() + 1];
 dp[0] = true;

 // 遍历字符串 s 的每个位置 i
 for (int i = 1; i <= s.length(); i++) {
 // 遍历之前的位置 j
 for (int j = 0; j < i; j++) {
 // 状态转移
 if (dp[j] && wordSet.contains(s.substring(j, i))) {
 dp[i] = true;
 break; // 只要找到一种拆分方式即可
 }
 }
 }

 return dp[s.length()];
}

/***
 * 另一种实现方式：先遍历字典中的单词，再遍历字符串位置
 * 这更符合完全背包问题的思路
 */
public static boolean wordBreakKnapsackStyle(String s, List<String> wordDict) {
 // 参数验证
 if (s == null || s.length() == 0) {
 return false;
 }

 // 创建一维 DP 数组
 boolean[] dp = new boolean[s.length() + 1];
 dp[0] = true;

 // 先遍历容量（字符串长度）
 for (int i = 1; i <= s.length(); i++) {
 // 再遍历物品（字典中的单词）
 for (String word : wordDict) {

```

```

 int wordLength = word.length();
 // 如果当前位置 i 大于等于单词长度，并且前 i-wordLength 个字符可以被拆分
 // 并且子字符串 s[i-wordLength:i] 等于当前单词
 if (i >= wordLength && dp[i - wordLength] &&
 s.substring(i - wordLength, i).equals(word)) {
 dp[i] = true;
 break; // 只要找到一种拆分方式即可
 }
 }

}

return dp[s.length()];
}

/***
 * 递归+记忆化搜索实现
 */
public static boolean wordBreakDFS(String s, List<String> wordDict) {
 // 参数验证
 if (s == null || s.length() == 0) {
 return false;
 }

 // 将 wordDict 转换为 Set，提高查找效率
 Set<String> wordSet = new HashSet<>(wordDict);

 // 使用 HashMap 作为缓存，键为子字符串的起始索引，值为该子字符串是否可以被拆分
 Map<Integer, Boolean> memo = new HashMap<>();

 // 调用递归辅助函数
 return dfs(s, 0, wordSet, memo);
}

/***
 * 递归辅助函数
 * @param s 目标字符串
 * @param start 子字符串的起始索引
 * @param wordSet 单词集合
 * @param memo 缓存
 * @return 从 start 开始的子字符串是否可以被拆分
 */
private static boolean dfs(String s, int start, Set<String> wordSet, Map<Integer, Boolean> memo) {

```

```

// 基础情况：如果已经到达字符串末尾，表示成功拆分
if (start == s.length()) {
 return true;
}

// 检查缓存
if (memo.containsKey(start)) {
 return memo.get(start);
}

// 尝试从 start 位置开始的所有可能的子字符串
for (int end = start + 1; end <= s.length(); end++) {
 // 如果子字符串在字典中，并且剩余部分也可以被拆分
 if (wordSet.contains(s.substring(start, end)) && dfs(s, end, wordSet, memo)) {
 memo.put(start, true);
 return true;
 }
}

// 所有可能的拆分方式都失败了
memo.put(start, false);
return false;
}

/**
 * BFS 实现
 */
public static boolean wordBreakBFS(String s, List<String> wordDict) {
 // 参数验证
 if (s == null || s.length() == 0) {
 return false;
 }

 // 将 wordDict 转换为 Set，提高查找效率
 Set<String> wordSet = new HashSet<>(wordDict);

 // 创建队列，存储可以拆分到的位置
 Queue<Integer> queue = new LinkedList<>();
 // 标记已经访问过的位置，避免重复处理
 boolean[] visited = new boolean[s.length()];

 // 初始位置为 0
 queue.offer(0);

```

```

visited[0] = true;

while (!queue.isEmpty()) {
 int start = queue.poll();

 // 尝试从 start 位置开始的所有可能的子字符串
 for (int end = start + 1; end <= s.length(); end++) {
 // 如果子字符串在字典中
 if (wordSet.contains(s.substring(start, end))) {
 // 如果已经到达字符串末尾，表示成功拆分
 if (end == s.length()) {
 return true;
 }
 // 如果该位置尚未访问过，将其加入队列
 if (!visited[end]) {
 queue.offer(end);
 visited[end] = true;
 }
 }
 }
}

// 队列为空仍未返回 true，表示无法拆分
return false;
}

/**
 * 使用 Trie 树优化查找效率
 */
public static boolean wordBreakWithTrie(String s, List<String> wordDict) {
 // 参数验证
 if (s == null || s.length() == 0) {
 return false;
 }

 // 构建 Trie 树
 TrieNode root = buildTrie(wordDict);

 // 创建一维 DP 数组
 boolean[] dp = new boolean[s.length() + 1];
 dp[0] = true;

 // 遍历字符串 s 的每个位置 i

```

```

 for (int i = 0; i < s.length(); i++) {
 // 如果前 i 个字符无法被拆分，跳过
 if (!dp[i]) {
 continue;
 }

 // 从 i 位置开始，在 Trie 树中查找可能的单词
 TrieNode node = root;
 for (int j = i; j < s.length(); j++) {
 char c = s.charAt(j);
 if (node.children[c - 'a'] == null) {
 break; // 无法继续匹配
 }
 node = node.children[c - 'a'];
 // 如果找到一个单词，标记 dp[j+1] 为 true
 if (node.isEnd) {
 dp[j + 1] = true;
 }
 }
 }

 return dp[s.length()];
 }

 /**
 * Trie 树节点类
 */
 private static class TrieNode {
 boolean isEnd;
 TrieNode[] children;

 public TrieNode() {
 isEnd = false;
 children = new TrieNode[26]; // 假设只包含小写字母
 }
 }

 /**
 * 构建 Trie 树
 */
 private static TrieNode buildTrie(List<String> wordDict) {
 TrieNode root = new TrieNode();
 for (String word : wordDict) {

```

```

TrieNode node = root;
for (char c : word.toCharArray()) {
 int index = c - 'a';
 if (node.children[index] == null) {
 node.children[index] = new TrieNode();
 }
 node = node.children[index];
}
node.isEnd = true;
}
return root;
}
}

```

---

文件: Code25\_WordBreak.py

---

```

LeetCode 139. 单词拆分
题目描述: 给你一个字符串 s 和一个字符串列表 wordDict 作为字典。请你判断是否可以利用字典中出现的单词拼接出 s 。
注意: 不要求字典中出现的单词全部都使用, 并且字典中的单词可以重复使用。
链接: https://leetcode.cn/problems/word-break/
#
解题思路:
这是一个完全背包问题的变种, 其中:
- 背包容量: 字符串 s 的长度
- 物品: 字典中的单词
- 问题转化为: 是否可以选择一些单词 (可重复), 恰好拼接成字符串 s
#
状态定义: dp[i] 表示字符串 s 的前 i 个字符是否可以被拆分
状态转移方程: 对于每个位置 i, 遍历所有位置 j (j < i), 如果 dp[j] 为 true 且 s[j:i] 在字典中, 则 dp[i] 为 true
初始状态: dp[0] = true, 表示空字符串可以被拆分
#
时间复杂度: O(n^3), 其中 n 是字符串 s 的长度 (需要两层循环, 并且每次需要判断子字符串是否在字典中)
空间复杂度: O(n), 使用一维 DP 数组

```

```
from typing import List, Set, Dict, Optional
```

```
def word_break(s: str, word_dict: List[str]) -> bool:
```

```
"""
```

```
判断是否可以利用字典中出现的单词拼接出 s
```

```
Args:
```

```
 s: 目标字符串
```

```
 word_dict: 单词字典
```

```
Returns:
```

```
 bool: 是否可以拼接出 s
```

```
"""
```

```
参数验证
```

```
if not s:
```

```
 return False # 空字符串返回 false
```

```
将 word_dict 转换为 set, 提高查找效率
```

```
word_set = set(word_dict)
```

```
获取字典中单词的最大长度, 用于后续剪枝
```

```
max_word_length = 0
```

```
for word in word_dict:
```

```
 max_word_length = max(max_word_length, len(word))
```

```
创建一维 DP 数组, dp[i] 表示字符串 s 的前 i 个字符是否可以被拆分
```

```
dp = [False] * (len(s) + 1)
```

```
初始状态: 空字符串可以被拆分
```

```
dp[0] = True
```

```
遍历字符串 s 的每个位置 i
```

```
for i in range(1, len(s) + 1):
```

```
 # 遍历之前的位置 j, 从 max(0, i - max_word_length) 到 i-1
```

```
 # 这样可以避免检查过长的子字符串
```

```
 start = max(0, i - max_word_length)
```

```
 for j in range(start, i):
```

```
 # 状态转移: 如果前 j 个字符可以被拆分, 且子字符串 s[j:i] 在字典中, 则前 i 个字符可以被拆
```

```
分
```

```
 if dp[j] and s[j:i] in word_set:
```

```
 dp[i] = True
```

```
 break # 只要找到一种拆分方式即可
```

```
返回结果: 整个字符串 s 是否可以被拆分
```

```
return dp[len(s)]
```

```
def word_break_optimized(s: str, word_dict: List[str]) -> bool:
 """
 优化版本：移除 max_word_length 的计算，直接使用普通的两层循环
 """

 # 参数验证
 if not s:
 return False

 # 将 word_dict 转换为 set，提高查找效率
 word_set = set(word_dict)

 # 创建一维 DP 数组
 dp = [False] * (len(s) + 1)
 dp[0] = True

 # 遍历字符串 s 的每个位置 i
 for i in range(1, len(s) + 1):
 # 遍历之前的位置 j
 for j in range(i):
 # 状态转移
 if dp[j] and s[j:i] in word_set:
 dp[i] = True
 break # 只要找到一种拆分方式即可

 return dp[len(s)]
```

```
def word_break_knapsack_style(s: str, word_dict: List[str]) -> bool:
 """
 另一种实现方式：先遍历字典中的单词，再遍历字符串位置
 这更符合完全背包问题的思路
 """

 # 参数验证
 if not s:
 return False

 # 创建一维 DP 数组
 dp = [False] * (len(s) + 1)
 dp[0] = True

 # 先遍历容量（字符串长度）
 for i in range(1, len(s) + 1):
```

```
再遍历物品（字典中的单词）
for word in word_dict:
 word_length = len(word)
 # 如果当前位置 i 大于等于单词长度，并且前 i-word_length 个字符可以被拆分
 # 并且子字符串 s[i-word_length:i] 等于当前单词
 if i >= word_length and dp[i - word_length] and \
 s[i - word_length:i] == word:
 dp[i] = True
 break # 只要找到一种拆分方式即可

return dp[len(s)]
```

```
def word_break_dfs(s: str, word_dict: List[str]) -> bool:
 """
 递归+记忆化搜索实现
 """

 # 参数验证
 if not s:
 return False

 # 将 word_dict 转换为 set，提高查找效率
 word_set = set(word_dict)

 # 使用字典作为缓存，键为子字符串的起始索引，值为该子字符串是否可以被拆分
 memo = {}

 def dfs(start: int) -> bool:
 """
 递归辅助函数
 # 基础情况：如果已经到达字符串末尾，表示成功拆分
 if start == len(s):
 return True

 # 检查缓存
 if start in memo:
 return memo[start]

 # 尝试从 start 位置开始的所有可能的子字符串
 for end in range(start + 1, len(s) + 1):
 # 如果子字符串在字典中，并且剩余部分也可以被拆分
 if s[start:end] in word_set and dfs(end):
 memo[start] = True
 return True

 return False

 return dfs(0)
```



```
 visited[end] = True

队列为空仍未返回 true, 表示无法拆分
return False

class TrieNode:
 """
 Trie 树节点类
 """

 def __init__(self):
 self.children = {}
 self.is_end = False

def build_trie(word_dict: List[str]) -> TrieNode:
 """
 构建 Trie 树
 """

 root = TrieNode()
 for word in word_dict:
 node = root
 for char in word:
 if char not in node.children:
 node.children[char] = TrieNode()
 node = node.children[char]
 node.is_end = True
 return root

def word_break_with_trie(s: str, word_dict: List[str]) -> bool:
 """
 使用 Trie 树优化查找效率
 """

 # 参数验证
 if not s:
 return False

 # 构建 Trie 树
 root = build_trie(word_dict)

 # 创建一维 DP 数组
 dp = [False] * (len(s) + 1)
```

```
dp[0] = True

遍历字符串 s 的每个位置 i
for i in range(len(s)):
 # 如果前 i 个字符无法被拆分，跳过
 if not dp[i]:
 continue

 # 从 i 位置开始，在 Trie 树中查找可能的单词
 node = root
 for j in range(i, len(s)):
 char = s[j]
 if char not in node.children:
 break # 无法继续匹配
 node = node.children[char]
 # 如果找到一个单词，标记 dp[j+1] 为 true
 if node.is_end:
 dp[j + 1] = True

return dp[len(s)]
```

  

```
测试代码
if __name__ == "__main__":
 # 测试用例 1
 s1 = "leetcode"
 word_dict1 = ["leet", "code"]
 print(f"测试用例 1 结果: {word_break(s1, word_dict1)}") # 预期输出: True

 # 测试用例 2
 s2 = "applepenapple"
 word_dict2 = ["apple", "pen"]
 print(f"测试用例 2 结果: {word_break(s2, word_dict2)}") # 预期输出: True

 # 测试用例 3
 s3 = "catsandog"
 word_dict3 = ["cats", "dog", "sand", "and", "cat"]
 print(f"测试用例 3 结果: {word_break(s3, word_dict3)}") # 预期输出: False

 # 测试用例 4
 s4 = ""
 word_dict4 = ["a"]
 print(f"测试用例 4 结果: {word_break(s4, word_dict4)}") # 预期输出: False
```

文件: Code26\_CoinChange2.cpp

```
#include <iostream>
#include <vector>
#include <algorithm>

// LeetCode 518. 零钱兑换 II
// 题目描述: 给你一个整数数组 coins 表示不同面额的硬币，另给一个整数 amount 表示总金额。
// 请你计算并返回可以凑成总金额的硬币组合数。如果任何硬币组合都无法凑出总金额，返回 0。
// 假设每一种面额的硬币有无限个。
// 链接: https://leetcode.cn/problems/coin-change-ii/
//
// 解题思路:
// 这是一个典型的完全背包问题，其中:
// - 背包容量: 总金额 amount
// - 物品: 不同面额的硬币
// - 物品可以无限使用 (完全背包)
// - 目标: 求恰好装满背包的方案数
//
// 状态定义: dp[i] 表示凑成总金额 i 的硬币组合数
// 状态转移方程: dp[i] += dp[i - coin]，其中 coin 是当前考虑的硬币面额，且 i >= coin
// 初始状态: dp[0] = 1 表示凑成总金额 0 有一种方式 (不使用任何硬币)
//
// 注意: 为了计算组合数而不是排列数，需要先遍历物品 (硬币)，再遍历容量 (金额)
// 时间复杂度: O(amount * n)，其中 n 是硬币种类数
// 空间复杂度: O(amount)，使用一维 DP 数组

using namespace std;

/**
 * 计算凑成总金额的硬币组合数
 * @param amount 总金额
 * @param coins 不同面额的硬币数组
 * @return 可以凑成总金额的硬币组合数
 */
int change(int amount, vector<int>& coins) {
 // 参数验证
 if (amount < 0) {
 return 0;
 }
```

```

// 创建一维 DP 数组, dp[i] 表示凑成总金额 i 的硬币组合数
vector<int> dp(amount + 1, 0);

// 初始状态: 凑成总金额 0 有一种方式 (不使用任何硬币)
dp[0] = 1;

// 先遍历物品 (硬币), 再遍历容量 (金额), 这样计算的是组合数
// 如果先遍历容量再遍历物品, 计算的是排列数
for (int coin : coins) {
 // 对于完全背包问题, 容量遍历是正序的, 允许物品被重复使用
 for (int i = coin; i <= amount; i++) {
 dp[i] += dp[i - coin];
 }
}

// 返回结果: 凑成总金额 amount 的硬币组合数
return dp[amount];
}

```

```

/**
 * 优化版本: 添加硬币排序和剪枝
 */
int changeOptimized(int amount, vector<int>& coins) {
 // 参数验证
 if (amount < 0) {
 return 0;
 }
 if (coins.empty()) {
 return amount == 0 ? 1 : 0;
 }

 // 对硬币进行排序, 便于后续剪枝
 sort(coins.begin(), coins.end());
}

// 创建一维 DP 数组
vector<int> dp(amount + 1, 0);
dp[0] = 1;

```

```

// 先遍历物品 (硬币), 再遍历容量 (金额)
for (int coin : coins) {
 // 如果当前硬币面额已经大于 amount, 可以跳过
 if (coin > amount) {

```

```

 break;
 }

 // 正序遍历容量，允许重复使用硬币
 for (int i = coin; i <= amount; i++) {
 dp[i] += dp[i - coin];
 }
}

return dp[amount];
}

/***
 * 二维 DP 数组实现
 * dp[i][j] 表示使用前 i 种硬币凑成总金额 j 的组合数
 */
int change2D(int amount, vector<int>& coins) {
 // 参数验证
 if (amount < 0) {
 return 0;
 }
 if (coins.empty()) {
 return amount == 0 ? 1 : 0;
 }

 int n = coins.size();
 // 创建二维 DP 数组
 vector<vector<int>> dp(n + 1, vector<int>(amount + 1, 0));

 // 初始化：使用 0 种硬币只能凑成总金额 0
 dp[0][0] = 1;

 // 填充 DP 数组
 for (int i = 1; i <= n; i++) {
 int coin = coins[i - 1];
 for (int j = 0; j <= amount; j++) {
 // 不使用当前硬币
 dp[i][j] = dp[i - 1][j];
 // 使用当前硬币（如果可以的话）
 if (j >= coin) {
 dp[i][j] += dp[i][j - coin];
 }
 }
 }
}

```

```

 return dp[n][amount];
}

/***
 * 递归+记忆化搜索实现
*/
int changeDFS(int amount, vector<int>& coins) {
 // 参数验证
 if (amount < 0) {
 return 0;
 }
 if (coins.empty()) {
 return amount == 0 ? 1 : 0;
 }

 // 对硬币进行排序，便于剪枝
 sort(coins.begin(), coins.end());

 // 使用二维数组作为缓存，memo[i][j]表示使用前 i 种硬币凑成总金额 j 的组合数
 vector<vector<int>> memo(coins.size(), vector<int>(amount + 1, -1));

 // 定义 DFS 函数
 function<int(int, int)> dfs = [&](int index, int remaining) {
 // 基础情况：如果剩余金额为 0，找到了一种组合
 if (remaining == 0) {
 return 1;
 }

 // 基础情况：如果已经考虑完所有硬币或者剩余金额小于 0，无法凑成
 if (index == coins.size() || remaining < 0) {
 return 0;
 }

 // 检查缓存
 if (memo[index][remaining] != -1) {
 return memo[index][remaining];
 }

 // 选择不使用当前硬币
 int notUse = dfs(index + 1, remaining);

 // 选择使用当前硬币（如果可以的话）
 int use = 0;
 for (int coin : coins) {
 if (coin >= remaining) {
 break;
 }
 use += dfs(index, remaining - coin);
 }

 memo[index][remaining] = use + notUse;
 return use + notUse;
 };
}

```

```

int use = 0;
if (remaining >= coins[index]) {
 // 注意这里 index 不变, 表示可以重复使用当前硬币
 use = dfs(index, remaining - coins[index]);
}

// 计算总组合数并缓存
memo[index][remaining] = notUse + use;
return memo[index][remaining];
};

// 调用递归函数
return dfs(0, amount);
}

/***
 * 另一种递归+记忆化实现方式, 不考虑硬币的顺序
 * 使用 index 来确保每种硬币只按顺序考虑一次, 避免重复计算
 */
int changeDFS2(int amount, vector<int>& coins) {
 // 参数验证
 if (amount < 0) {
 return 0;
 }
 if (coins.empty()) {
 return amount == 0 ? 1 : 0;
 }

 // 对硬币进行排序, 便于剪枝
 sort(coins.begin(), coins.end());

 // 使用二维数组作为缓存
 vector<vector<int>> memo(coins.size(), vector<int>(amount + 1, -1));

 // 定义 DFS 函数
 function<int(int, int)> dfs2 = [&](int index, int remaining) {
 // 基础情况
 if (remaining == 0) {
 return 1;
 }
 if (index == coins.size() || remaining < coins[index]) {
 return 0;
 }

```

```

// 检查缓存
if (memo[index][remaining] != -1) {
 return memo[index][remaining];
}

int count = 0;
// 尝试使用当前硬币 0 次、1 次、2 次... 直到超过剩余金额
for (int k = 0; k * coins[index] <= remaining; k++) {
 // 使用 k 次当前硬币后，剩余金额为 remaining - k * coins[index]，接下来考虑下一种硬币
 count += dfs2(index + 1, remaining - k * coins[index]);
}

// 缓存结果
memo[index][remaining] = count;
return count;
};

// 调用递归函数
return dfs2(0, amount);
}

/***
 * 计算排列数的实现（如果题目要求不同顺序算不同的组合）
 * 注意：这不是本题的要求，但作为对比提供
 */
int changePermutation(int amount, vector<int>& coins) {
 // 参数验证
 if (amount < 0) {
 return 0;
 }

 // 创建一维 DP 数组
 vector<int> dp(amount + 1, 0);
 dp[0] = 1;

 // 先遍历容量（金额），再遍历物品（硬币），这样计算的是排列数
 for (int i = 1; i <= amount; i++) {
 for (int coin : coins) {
 if (i >= coin) {
 dp[i] += dp[i - coin];
 }
 }
 }
}

```

```

 }

 return dp[amount];
}

int main() {
 // 测试用例 1
 vector<int> coins1 = {1, 2, 5};
 int amount1 = 5;
 cout << "测试用例 1 结果: " << change(amount1, coins1) << endl; // 预期输出: 4

 // 测试用例 2
 vector<int> coins2 = {2};
 int amount2 = 3;
 cout << "测试用例 2 结果: " << change(amount2, coins2) << endl; // 预期输出: 0

 // 测试用例 3
 vector<int> coins3 = {10};
 int amount3 = 10;
 cout << "测试用例 3 结果: " << change(amount3, coins3) << endl; // 预期输出: 1

 // 测试用例 4
 vector<int> coins4 = {1, 2, 5};
 int amount4 = 100;
 cout << "测试用例 4 结果: " << change(amount4, coins4) << endl; // 预期输出: 204

 return 0;
}

```

=====

文件: Code26\_CoinChange2.java

=====

```

package class073;

import java.util.Arrays;

// LeetCode 518. 零钱兑换 II
// 题目描述: 给你一个整数数组 coins 表示不同面额的硬币，另给一个整数 amount 表示总金额。
// 请你计算并返回可以凑成总金额的硬币组合数。如果任何硬币组合都无法凑出总金额，返回 0。
// 假设每一种面额的硬币有无限个。
// 链接: https://leetcode.cn/problems/coin-change-ii/
//
```

```

// 解题思路:
// 这是一个典型的完全背包问题, 其中:
// - 背包容量: 总金额 amount
// - 物品: 不同面额的硬币
// - 物品可以无限使用 (完全背包)
// - 目标: 求恰好装满背包的方案数
//
// 状态定义: dp[i] 表示凑成总金额 i 的硬币组合数
// 状态转移方程: dp[i] += dp[i - coin], 其中 coin 是当前考虑的硬币面额, 且 i >= coin
// 初始状态: dp[0] = 1 表示凑成总金额 0 有一种方式 (不使用任何硬币)
//
// 注意: 为了计算组合数而不是排列数, 需要先遍历物品 (硬币), 再遍历容量 (金额)
// 时间复杂度: O(amount * n), 其中 n 是硬币种类数
// 空间复杂度: O(amount), 使用一维 DP 数组

public class Code26_CoinChange2 {

 // 主方法, 用于测试
 public static void main(String[] args) {
 // 测试用例 1
 int[] coins1 = {1, 2, 5};
 int amount1 = 5;
 System.out.println("测试用例 1 结果: " + change(amount1, coins1)); // 预期输出: 4

 // 测试用例 2
 int[] coins2 = {2};
 int amount2 = 3;
 System.out.println("测试用例 2 结果: " + change(amount2, coins2)); // 预期输出: 0

 // 测试用例 3
 int[] coins3 = {10};
 int amount3 = 10;
 System.out.println("测试用例 3 结果: " + change(amount3, coins3)); // 预期输出: 1

 // 测试用例 4
 int[] coins4 = {1, 2, 5};
 int amount4 = 100;
 System.out.println("测试用例 4 结果: " + change(amount4, coins4)); // 预期输出: 204
 }

 /**
 * 计算凑成总金额的硬币组合数
 * @param amount 总金额

```

```

* @param coins 不同面额的硬币数组
* @return 可以凑成总金额的硬币组合数
*/
public static int change(int amount, int[] coins) {
 // 参数验证
 if (amount < 0) {
 return 0;
 }

 // 创建一维 DP 数组, dp[i] 表示凑成总金额 i 的硬币组合数
 int[] dp = new int[amount + 1];

 // 初始状态: 凑成总金额 0 有一种方式 (不使用任何硬币)
 dp[0] = 1;

 // 先遍历物品 (硬币), 再遍历容量 (金额), 这样计算的是组合数
 // 如果先遍历容量再遍历物品, 计算的是排列数
 for (int coin : coins) {
 // 对于完全背包问题, 容量遍历是正序的, 允许物品被重复使用
 for (int i = coin; i <= amount; i++) {
 dp[i] += dp[i - coin];
 }
 }

 // 返回结果: 凑成总金额 amount 的硬币组合数
 return dp[amount];
}

/***
 * 优化版本: 添加硬币排序和剪枝
 */
public static int changeOptimized(int amount, int[] coins) {
 // 参数验证
 if (amount < 0) {
 return 0;
 }

 if (coins == null || coins.length == 0) {
 return amount == 0 ? 1 : 0;
 }

 // 对硬币进行排序, 便于后续剪枝
 Arrays.sort(coins);
}

```

```

// 创建一维 DP 数组
int[] dp = new int[amount + 1];
dp[0] = 1;

// 先遍历物品（硬币），再遍历容量（金额）
for (int coin : coins) {
 // 如果当前硬币面额已经大于 amount，可以跳过
 if (coin > amount) {
 break;
 }
 // 正序遍历容量，允许重复使用硬币
 for (int i = coin; i <= amount; i++) {
 dp[i] += dp[i - coin];
 }
}

return dp[amount];
}

/***
 * 二维 DP 数组实现
 * dp[i][j] 表示使用前 i 种硬币凑成总金额 j 的组合数
 */
public static int change2D(int amount, int[] coins) {
 // 参数验证
 if (amount < 0) {
 return 0;
 }
 if (coins == null || coins.length == 0) {
 return amount == 0 ? 1 : 0;
 }

 int n = coins.length;
 // 创建二维 DP 数组
 int[][] dp = new int[n + 1][amount + 1];

 // 初始化：使用 0 种硬币只能凑成总金额 0
 dp[0][0] = 1;

 // 填充 DP 数组
 for (int i = 1; i <= n; i++) {
 int coin = coins[i - 1];
 for (int j = 0; j <= amount; j++) {

```

```

 // 不使用当前硬币
 dp[i][j] = dp[i - 1][j];
 // 使用当前硬币（如果可以的话）
 if (j >= coin) {
 dp[i][j] += dp[i][j - coin];
 }
 }

 return dp[n][amount];
}

/***
 * 递归+记忆化搜索实现
 */
public static int changeDFS(int amount, int[] coins) {
 // 参数验证
 if (amount < 0) {
 return 0;
 }
 if (coins == null || coins.length == 0) {
 return amount == 0 ? 1 : 0;
 }

 // 对硬币进行排序，便于剪枝
 Arrays.sort(coins);

 // 使用二维数组作为缓存，memo[i][j]表示使用前 i 种硬币凑成总金额 j 的组合数
 Integer[][] memo = new Integer[coins.length][amount + 1];

 // 调用递归辅助函数
 return dfs(coins, 0, amount, memo);
}

/***
 * 递归辅助函数
 * @param coins 硬币数组
 * @param index 当前考虑的硬币索引
 * @param amount 剩余需要凑的金额
 * @param memo 缓存数组
 * @return 可以凑成剩余金额的组合数
 */
private static int dfs(int[] coins, int index, int amount, Integer[][] memo) {

```

```

// 基础情况：如果剩余金额为 0，找到了一种组合
if (amount == 0) {
 return 1;
}

// 基础情况：如果已经考虑完所有硬币或者剩余金额小于 0，无法凑成
if (index == coins.length || amount < 0) {
 return 0;
}

// 检查缓存
if (memo[index][amount] != null) {
 return memo[index][amount];
}

// 选择不使用当前硬币
int notUse = dfs(coins, index + 1, amount, memo);

// 选择使用当前硬币（如果可以的话）
int use = 0;
if (amount >= coins[index]) {
 // 注意这里 index 不变，表示可以重复使用当前硬币
 use = dfs(coins, index, amount - coins[index], memo);
}

// 计算总组合数并缓存
memo[index][amount] = notUse + use;
return memo[index][amount];
}

/**
 * 另一种递归+记忆化实现方式，不考虑硬币的顺序
 * 使用 index 来确保每种硬币只按顺序考虑一次，避免重复计算
 */
public static int changeDFS2(int amount, int[] coins) {
 // 参数验证
 if (amount < 0) {
 return 0;
 }
 if (coins == null || coins.length == 0) {
 return amount == 0 ? 1 : 0;
 }
}

```

```

// 对硬币进行排序，便于剪枝
Arrays.sort(coins);

// 使用二维数组作为缓存
Integer[][] memo = new Integer[coins.length][amount + 1];

// 调用递归辅助函数
return dfs2(coins, 0, amount, memo);
}

/**
 * 另一种递归辅助函数实现
 */
private static int dfs2(int[] coins, int index, int amount, Integer[][] memo) {
 // 基础情况
 if (amount == 0) {
 return 1;
 }
 if (index == coins.length || amount < coins[index]) {
 return 0;
 }

 // 检查缓存
 if (memo[index][amount] != null) {
 return memo[index][amount];
 }

 int count = 0;
 // 尝试使用当前硬币 0 次、1 次、2 次... 直到超过剩余金额
 for (int k = 0; k * coins[index] <= amount; k++) {
 // 使用 k 次当前硬币后，剩余金额为 amount - k * coins[index]，接下来考虑下一种硬币
 count += dfs2(coins, index + 1, amount - k * coins[index], memo);
 }

 // 缓存结果
 memo[index][amount] = count;
 return count;
}

/**
 * 计算排列数的实现（如果题目要求不同顺序算不同的组合）
 * 注意：这不是本题的要求，但作为对比提供
 */

```

```

public static int changePermutation(int amount, int[] coins) {
 // 参数验证
 if (amount < 0) {
 return 0;
 }

 // 创建一维 DP 数组
 int[] dp = new int[amount + 1];
 dp[0] = 1;

 // 先遍历容量 (金额), 再遍历物品 (硬币), 这样计算的是排列数
 for (int i = 1; i <= amount; i++) {
 for (int coin : coins) {
 if (i >= coin) {
 dp[i] += dp[i - coin];
 }
 }
 }

 return dp[amount];
}
}

```

文件: Code26\_CoinChange2.py

```

LeetCode 518. 零钱兑换 II
题目描述: 给你一个整数数组 coins 表示不同面额的硬币，另给一个整数 amount 表示总金额。
请你计算并返回可以凑成总金额的硬币组合数。如果任何硬币组合都无法凑出总金额，返回 0 。
假设每一种面额的硬币有无限个。
链接: https://leetcode.cn/problems/coin-change-ii/
#
解题思路:
这是一个典型的完全背包问题，其中:
- 背包容量: 总金额 amount
- 物品: 不同面额的硬币
- 物品可以无限使用 (完全背包)
- 目标: 求恰好装满背包的方案数
#
状态定义: dp[i] 表示凑成总金额 i 的硬币组合数
状态转移方程: dp[i] += dp[i - coin], 其中 coin 是当前考虑的硬币面额，且 i >= coin
初始状态: dp[0] = 1 表示凑成总金额 0 有一种方式 (不使用任何硬币)

```

```
注意: 为了计算组合数而不是排列数, 需要先遍历物品(硬币), 再遍历容量(金额)
时间复杂度: O(amount * n), 其中 n 是硬币种类数
空间复杂度: O(amount), 使用一维 DP 数组
```

```
from typing import List, Optional
```

```
def change(amount: int, coins: List[int]) -> int:
```

```
"""
```

```
 计算凑成总金额的硬币组合数
```

```
Args:
```

```
 amount: 总金额
```

```
 coins: 不同面额的硬币数组
```

```
Returns:
```

```
 int: 可以凑成总金额的硬币组合数
```

```
"""
```

```
参数验证
```

```
if amount < 0:
```

```
 return 0
```

```
创建一维 DP 数组, dp[i] 表示凑成总金额 i 的硬币组合数
```

```
dp = [0] * (amount + 1)
```

```
初始状态: 凑成总金额 0 有一种方式(不使用任何硬币)
```

```
dp[0] = 1
```

```
先遍历物品(硬币), 再遍历容量(金额), 这样计算的是组合数
```

```
如果先遍历容量再遍历物品, 计算的是排列数
```

```
for coin in coins:
```

```
 # 对于完全背包问题, 容量遍历是正序的, 允许物品被重复使用
```

```
 for i in range(coin, amount + 1):
```

```
 dp[i] += dp[i - coin]
```

```
返回结果: 凑成总金额 amount 的硬币组合数
```

```
return dp[amount]
```

```
def change_optimized(amount: int, coins: List[int]) -> int:
```

```
"""
```

```
 优化版本: 添加硬币排序和剪枝
```

```

"""
参数验证
if amount < 0:
 return 0
if not coins:
 return 1 if amount == 0 else 0

对硬币进行排序，便于后续剪枝
coins.sort()

创建一维 DP 数组
dp = [0] * (amount + 1)
dp[0] = 1

先遍历物品（硬币），再遍历容量（金额）
for coin in coins:
 # 如果当前硬币面额已经大于 amount，可以跳过
 if coin > amount:
 break
 # 正序遍历容量，允许重复使用硬币
 for i in range(coin, amount + 1):
 dp[i] += dp[i - coin]

return dp[amount]

def change_2d(amount: int, coins: List[int]) -> int:
"""
二维 DP 数组实现
dp[i][j] 表示使用前 i 种硬币凑成总金额 j 的组合数
"""

参数验证
if amount < 0:
 return 0
if not coins:
 return 1 if amount == 0 else 0

n = len(coins)
创建二维 DP 数组
dp = [[0] * (amount + 1) for _ in range(n + 1)]

初始化：使用 0 种硬币只能凑成总金额 0
dp[0][0] = 1

```

```

填充 DP 数组
for i in range(1, n + 1):
 coin = coins[i - 1]
 for j in range(amount + 1):
 # 不使用当前硬币
 dp[i][j] = dp[i - 1][j]
 # 使用当前硬币（如果可以的话）
 if j >= coin:
 dp[i][j] += dp[i][j - coin]

return dp[n][amount]

def change_dfs(amount: int, coins: List[int]) -> int:
 """
 递归+记忆化搜索实现
 """

 # 参数验证
 if amount < 0:
 return 0
 if not coins:
 return 1 if amount == 0 else 0

 # 对硬币进行排序，便于剪枝
 coins.sort()

 # 使用二维列表作为缓存，memo[i][j]表示使用前 i 种硬币凑成总金额 j 的组合数
 memo = [[-1] * (amount + 1) for _ in range(len(coins))]

 def dfs(index: int, remaining: int) -> int:
 """
 递归辅助函数
 # 基础情况：如果剩余金额为 0，找到了一种组合
 if remaining == 0:
 return 1
 # 基础情况：如果已经考虑完所有硬币或者剩余金额小于 0，无法凑成
 if index == len(coins) or remaining < 0:
 return 0
 # 检查缓存
 if memo[index][remaining] != -1:
 return memo[index][remaining]
 # 从当前硬币开始尝试
 for coin in coins[index:]:
 if coin > remaining:
 break
 memo[index][remaining] += dfs(index + 1, remaining - coin)
 return memo[index][remaining]

 return dfs(0, amount)

```

```

选择不使用当前硬币
not_use = dfs(index + 1, remaining)

选择使用当前硬币（如果可以的话）
use = 0
if remaining >= coins[index]:
 # 注意这里 index 不变，表示可以重复使用当前硬币
 use = dfs(index, remaining - coins[index])

计算总组合数并缓存
memo[index][remaining] = not_use + use
return memo[index][remaining]

调用递归函数
return dfs(0, amount)

```

```

def change_dfs2(amount: int, coins: List[int]) -> int:
 """
另一种递归+记忆化实现方式，不考虑硬币的顺序
使用 index 来确保每种硬币只按顺序考虑一次，避免重复计算
 """
 # 参数验证
 if amount < 0:
 return 0
 if not coins:
 return 1 if amount == 0 else 0

 # 对硬币进行排序，便于剪枝
 coins.sort()

 # 使用二维列表作为缓存
 memo = [[-1] * (amount + 1) for _ in range(len(coins))]

 def dfs2(index: int, remaining: int) -> int:
 """
另一种递归辅助函数实现"""
 # 基础情况
 if remaining == 0:
 return 1
 if index == len(coins) or remaining < coins[index]:
 return 0

 # 选择不使用当前硬币
 not_use = dfs2(index + 1, remaining)

 # 选择使用当前硬币（如果可以的话）
 use = 0
 if remaining >= coins[index]:
 # 注意这里 index 不变，表示可以重复使用当前硬币
 use = dfs2(index, remaining - coins[index])

 # 计算总组合数并缓存
 memo[index][remaining] = not_use + use
 return memo[index][remaining]

 return dfs2(0, amount)

```

```

检查缓存
if memo[index][remaining] != -1:
 return memo[index][remaining]

count = 0
尝试使用当前硬币 0 次、1 次、2 次... 直到超过剩余金额
for k in range(0, remaining // coins[index] + 1):
 # 使用 k 次当前硬币后，剩余金额为 remaining - k * coins[index]，接下来考虑下一种硬币
 count += dfs2(index + 1, remaining - k * coins[index])

缓存结果
memo[index][remaining] = count
return count

调用递归函数
return dfs2(0, amount)

```

```
def change_permutation(amount: int, coins: List[int]) -> int:
```

```
"""

```

计算排列数的实现（如果题目要求不同顺序算不同的组合）

注意：这不是本题的要求，但作为对比提供

```
"""

```

```
参数验证
```

```
if amount < 0:
 return 0
```

```
创建一维 DP 数组
```

```
dp = [0] * (amount + 1)
dp[0] = 1
```

```
先遍历容量（金额），再遍历物品（硬币），这样计算的是排列数
```

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

```
 for coin in coins:
```

```
 if i >= coin:
```

```
 dp[i] += dp[i - coin]
```

```
return dp[amount]
```

```
测试代码
```

```
if __name__ == "__main__":
```

```
测试用例 1
```

```

coins1 = [1, 2, 5]
amount1 = 5
print(f"测试用例 1 结果: {change(amount1, coins1)}") # 预期输出: 4

测试用例 2
coins2 = [2]
amount2 = 3
print(f"测试用例 2 结果: {change(amount2, coins2)}") # 预期输出: 0

测试用例 3
coins3 = [10]
amount3 = 10
print(f"测试用例 3 结果: {change(amount3, coins3)}") # 预期输出: 1

测试用例 4
coins4 = [1, 2, 5]
amount4 = 100
print(f"测试用例 4 结果: {change(amount4, coins4)}") # 预期输出: 204

```

---

文件: Code27\_TargetSum.cpp

---

```

#include <iostream>
#include <vector>
#include <algorithm>
#include <unordered_map>
#include <string>

// LeetCode 494. 目标和
// 题目描述: 给你一个整数数组 nums 和一个整数 target。
// 向数组中的每个整数前添加 '+' 或 '-'，然后串联起所有整数，可以构造一个 表达式：
// 例如，nums = [2, 1]，可以在 2 之前添加 '+'，在 1 之前添加 '-'，得到表达式 "+2-1"。
// 返回可以通过上述方法构造的、运算结果等于 target 的不同 表达式 的数目。
// 链接: https://leetcode.cn/problems/target-sum/
//
// 解题思路:
// 这是一个背包问题的变种，我们可以将问题转化为:
// 找到一个子集，使得该子集中的元素和与其余元素和的差等于 target
// 设所有元素的和为 sum，子集和为 subsetSum，则:
// subsetSum - (sum - subsetSum) = target
// 即 2*subsetSum = sum + target
// 因此 subsetSum = (sum + target) / 2

```

```
//
// 所以问题转化为：找到和为 subsetSum 的子集数目
// 这是一个 0-1 背包问题（每个元素只能选或不选）

// 状态定义：dp[i] 表示和为 i 的子集数目
// 状态转移方程：dp[i] += dp[i - num]，其中 num 是当前元素，且 i >= num
// 初始状态：dp[0] = 1 表示和为 0 的子集有一个（空集）

// 时间复杂度：O(n * target)，其中 n 是数组长度
// 空间复杂度：O(target)，使用一维 DP 数组
```

```
using namespace std;
```

```
/**
 * 计算可以通过添加'+'或'-'使得表达式结果等于 target 的不同表达式数目
 * @param nums 整数数组
 * @param target 目标和
 * @return 不同表达式的数目
 */
```

```
int findTargetSumWays(vector<int>& nums, int target) {
 // 参数验证
 if (nums.empty()) {
 return 0;
 }
```

```
 // 计算所有元素的和
 int sum = 0;
 for (int num : nums) {
 sum += num;
 }
```

```
 // 检查是否有解的条件
 // 1. sum + target 必须是非负数
 // 2. sum + target 必须是偶数
 if (sum < abs(target) || (sum + target) % 2 != 0) {
 return 0;
 }
```

```
 // 计算目标子集和
 int subsetSum = (sum + target) / 2;
```

```
 // 创建一维 DP 数组，dp[i] 表示和为 i 的子集数目
 vector<int> dp(subsetSum + 1, 0);
```

```

// 初始状态: 和为 0 的子集有一个 (空集)
dp[0] = 1;

// 对于每个元素, 逆序遍历子集和 (0-1 背包问题)
for (int num : nums) {
 for (int i = subsetSum; i >= num; i--) {
 dp[i] += dp[i - num];
 }
}

// 返回结果: 和为 subsetSum 的子集数目
return dp[subsetSum];
}

/***
 * 优化版本: 处理可能的大数问题 (使用 long 类型)
 */
int findTargetSumWaysOptimized(vector<int>& nums, int target) {
 if (nums.empty()) {
 return 0;
 }

 int sum = 0;
 for (int num : nums) {
 sum += num;
 }

 if (sum < abs(target) || (sum + target) % 2 != 0) {
 return 0;
 }

 int subsetSum = (sum + target) / 2;

 // 使用 long 类型防止整数溢出
 vector<long> dp(subsetSum + 1, 0);
 dp[0] = 1;

 for (int num : nums) {
 for (int i = subsetSum; i >= num; i--) {
 dp[i] += dp[i - num];
 }
 }
}

```

```

// 转换为 int 返回
return static_cast<int>(dp[subsetSum]);
}

/***
 * 二维 DP 数组实现
 * dp[i][j] 表示前 i 个元素中和为 j 的子集数目
 */
int findTargetSumWays2D(vector<int>& nums, int target) {
 if (nums.empty()) {
 return 0;
 }

 int sum = 0;
 for (int num : nums) {
 sum += num;
 }

 if (sum < abs(target) || (sum + target) % 2 != 0) {
 return 0;
 }

 int subsetSum = (sum + target) / 2;
 int n = nums.size();

 // 创建二维 DP 数组
 vector<vector<int>> dp(n + 1, vector<int>(subsetSum + 1, 0));

 // 初始化: 前 0 个元素中和为 0 的子集有一个 (空集)
 dp[0][0] = 1;

 // 填充 DP 数组
 for (int i = 1; i <= n; i++) {
 int num = nums[i - 1];
 for (int j = 0; j <= subsetSum; j++) {
 // 不选当前元素
 dp[i][j] = dp[i - 1][j];
 // 选当前元素 (如果可以的话)
 if (j >= num) {
 dp[i][j] += dp[i - 1][j - num];
 }
 }
 }
}

```

```

}

return dp[n][subsetSum];
}

/***
 * 递归+记忆化搜索实现
 */
int findTargetSumWaysDFS(vector<int>& nums, int target) {
 if (nums.empty()) {
 return 0;
 }

 // 计算所有元素的和
 int sum = 0;
 for (int num : nums) {
 sum += num;
 }

 // 检查是否有解的条件
 if (sum < abs(target) || (sum + target) % 2 != 0) {
 return 0;
 }

 int subsetSum = (sum + target) / 2;
 int n = nums.size();

 // 使用二维数组作为缓存, memo[i][j]表示前 i 个元素中和为 j 的子集数目
 vector<vector<int>> memo(n, vector<int>(subsetSum + 1, -1));

 // 定义 DFS 函数
 function<int(int, int)> dfs = [&](int index, int currentSum) {
 // 基础情况: 如果已经考虑完所有元素
 if (index == n) {
 // 如果当前子集和等于目标子集和, 返回 1, 否则返回 0
 return currentSum == subsetSum ? 1 : 0;
 }

 // 检查缓存
 if (memo[index][currentSum] != -1) {
 return memo[index][currentSum];
 }
 };
}

```

```

// 计算当前状态的解
int result = 0;

// 选择不将当前元素加入子集
result += dfs(index + 1, currentSum);

// 选择将当前元素加入子集（如果不会超过目标和）
if (currentSum + nums[index] <= subsetSum) {
 result += dfs(index + 1, currentSum + nums[index]);
}

// 缓存结果
memo[index][currentSum] = result;
return result;
};

// 调用递归函数
return dfs(0, 0);
}

/***
 * 另一种递归实现方式，直接计算表达式数目
 */
int findTargetSumWaysDFS2(vector<int>& nums, int target) {
 if (nums.empty()) {
 return 0;
 }

 // 使用 unordered_map 作为缓存，键为"index, currentSum"，值为该状态下的表达式数目
 unordered_map<string, int> memo;

 // 定义 DFS 函数
 function<int(int, int)> dfs2 = [&](int index, int currentSum) {
 // 基础情况：如果已经考虑完所有元素
 if (index == nums.size()) {
 // 如果当前和等于目标和，返回 1，否则返回 0
 return currentSum == target ? 1 : 0;
 }

 // 生成缓存键
 string key = to_string(index) + "," + to_string(currentSum);

 // 检查缓存

```

```

 if (memo.find(key) != memo.end()) {
 return memo[key];
 }

 // 选择在当前元素前添加'+'

 int add = dfs2(index + 1, currentSum + nums[index]);

 // 选择在当前元素前添加'-'

 int subtract = dfs2(index + 1, currentSum - nums[index]);

 // 计算总表达式数目

 int total = add + subtract;

 // 缓存结果

 memo[key] = total;
 return total;
};

// 调用递归函数

return dfs2(0, 0);
}

/**

 * 另一种动态规划方法，使用二维数组记录到达每个和的路径数
 */
int findTargetSumWaysAlternative(vector<int>& nums, int target) {
 // 参数验证
 if (nums.empty()) {
 return 0;
 }

 // 计算所有元素的和，用于确定可能的和的范围
 int sum = 0;
 for (int num : nums) {
 sum += num;
 }

 // 检查是否有解的条件
 if (sum < abs(target)) {
 return 0;
 }

 // 创建 DP 数组，dp[i][j] 表示前 i 个元素能组成和为 j 的表达式数目

```

```

// 由于和可能为负数，我们需要进行偏移，将和范围从[-sum, sum]映射到[0, 2*sum]
int offset = sum;
vector<vector<int>> dp(nums.size() + 1, vector<int>(2 * sum + 1, 0));

// 初始状态：前 0 个元素能组成和为 0 的表达式有一个（空表达式）
dp[0][offset] = 1;

// 填充 DP 数组
for (int i = 1; i <= nums.size(); i++) {
 int num = nums[i - 1];
 for (int j = 0; j < 2 * sum + 1; j++) {
 // 如果前 i-1 个元素能组成和为 j 的表达式
 if (dp[i - 1][j] > 0) {
 // 添加'+'：和变为 j + num
 if (j + num < 2 * sum + 1) {
 dp[i][j + num] += dp[i - 1][j];
 }
 // 添加'-'：和变为 j - num
 if (j - num >= 0) {
 dp[i][j - num] += dp[i - 1][j];
 }
 }
 }
}

// 返回结果：前 n 个元素能组成和为 target 的表达式数目
// 注意需要加上偏移量
return dp[nums.size()][target + offset];
}

int main() {
 // 测试用例 1
 vector<int> nums1 = {1, 1, 1, 1, 1};
 int target1 = 3;
 cout << "测试用例 1 结果：" << findTargetSumWays(nums1, target1) << endl; // 预期输出: 5

 // 测试用例 2
 vector<int> nums2 = {1};
 int target2 = 1;
 cout << "测试用例 2 结果：" << findTargetSumWays(nums2, target2) << endl; // 预期输出: 1

 // 测试用例 3
 vector<int> nums3 = {1, 2, 3, 4, 5};

```

```
int target3 = 3;
cout << "测试用例 3 结果：" << findTargetSumWays(nums3, target3) << endl; // 预期输出：3

return 0;
}
```

---

文件: Code27\_TargetSum.java

---

```
package class073;

import java.util.Arrays;
import java.util.HashMap;
import java.util.Map;

// LeetCode 494. 目标和
// 题目描述: 给你一个整数数组 nums 和一个整数 target。
// 向数组中的每个整数前添加 '+' 或 '-'，然后串联起所有整数，可以构造一个 表达式：
// 例如，nums = [2, 1]，可以在 2 之前添加 '+'，在 1 之前添加 '-'，得到表达式 "+2-1"。
// 返回可以通过上述方法构造的、运算结果等于 target 的不同 表达式 的数目。
// 链接: https://leetcode.cn/problems/target-sum/
//
// 解题思路:
// 这是一个背包问题的变种，我们可以将问题转化为:
// 找到一个子集，使得该子集中的元素和与其余元素和的差等于 target
// 设所有元素的和为 sum，子集和为 subsetSum，则:
//
$$subsetSum - (sum - subsetSum) = target$$

// 即 $2 * subsetSum = sum + target$
// 因此 $subsetSum = (sum + target) / 2$
//
// 所以问题转化为: 找到和为 subsetSum 的子集数目
// 这是一个 0-1 背包问题（每个元素只能选或不选）
//
// 状态定义: dp[i] 表示和为 i 的子集数目
// 状态转移方程: $dp[i] += dp[i - num]$ ，其中 num 是当前元素，且 $i \geq num$
// 初始状态: $dp[0] = 1$ 表示和为 0 的子集有一个（空集）
//
// 时间复杂度: $O(n * target)$ ，其中 n 是数组长度
// 空间复杂度: $O(target)$ ，使用一维 DP 数组

public class Code27_TargetSum {
```

```

// 主方法，用于测试
public static void main(String[] args) {
 // 测试用例 1
 int[] nums1 = {1, 1, 1, 1, 1};
 int target1 = 3;
 System.out.println("测试用例 1 结果: " + findTargetSumWays(nums1, target1)); // 预期输出:
5

 // 测试用例 2
 int[] nums2 = {1};
 int target2 = 1;
 System.out.println("测试用例 2 结果: " + findTargetSumWays(nums2, target2)); // 预期输出:
1

 // 测试用例 3
 int[] nums3 = {1, 2, 3, 4, 5};
 int target3 = 3;
 System.out.println("测试用例 3 结果: " + findTargetSumWays(nums3, target3)); // 预期输出:
3
}

/**
 * 计算可以通过添加'+'或'-'使得表达式结果等于 target 的不同表达式数目
 * @param nums 整数数组
 * @param target 目标和
 * @return 不同表达式的数目
 */
public static int findTargetSumWays(int[] nums, int target) {
 // 参数验证
 if (nums == null || nums.length == 0) {
 return 0;
 }

 // 计算所有元素的和
 int sum = 0;
 for (int num : nums) {
 sum += num;
 }

 // 检查是否有解的条件
 // 1. sum + target 必须是非负数
 // 2. sum + target 必须是偶数
 if (sum < Math.abs(target) || (sum + target) % 2 != 0) {

```

```

 return 0;
}

// 计算目标子集和
int subsetSum = (sum + target) / 2;

// 创建一维 DP 数组, dp[i] 表示和为 i 的子集数目
int[] dp = new int[subsetSum + 1];

// 初始状态: 和为 0 的子集有一个 (空集)
dp[0] = 1;

// 对于每个元素, 逆序遍历子集和 (0-1 背包问题)
for (int num : nums) {
 for (int i = subsetSum; i >= num; i--) {
 dp[i] += dp[i - num];
 }
}

// 返回结果: 和为 subsetSum 的子集数目
return dp[subsetSum];
}

/**
 * 优化版本: 处理可能的大数问题 (使用 long 类型)
 */
public static int findTargetSumWaysOptimized(int[] nums, int target) {
 if (nums == null || nums.length == 0) {
 return 0;
 }

 int sum = 0;
 for (int num : nums) {
 sum += num;
 }

 if (sum < Math.abs(target) || (sum + target) % 2 != 0) {
 return 0;
 }

 int subsetSum = (sum + target) / 2;

 // 使用 long 类型防止整数溢出

```

```

long[] dp = new long[subsetSum + 1];
dp[0] = 1;

for (int num : nums) {
 for (int i = subsetSum; i >= num; i--) {
 dp[i] += dp[i - num];
 }
}

// 转换为 int 返回
return (int) dp[subsetSum];
}

/***
 * 二维 DP 数组实现
 * dp[i][j] 表示前 i 个元素中和为 j 的子集数目
 */
public static int findTargetSumWays2D(int[] nums, int target) {
 if (nums == null || nums.length == 0) {
 return 0;
 }

 int sum = 0;
 for (int num : nums) {
 sum += num;
 }

 if (sum < Math.abs(target) || (sum + target) % 2 != 0) {
 return 0;
 }

 int subsetSum = (sum + target) / 2;
 int n = nums.length;

 // 创建二维 DP 数组
 int[][] dp = new int[n + 1][subsetSum + 1];

 // 初始化: 前 0 个元素中和为 0 的子集有一个 (空集)
 dp[0][0] = 1;

 // 填充 DP 数组
 for (int i = 1; i <= n; i++) {
 int num = nums[i - 1];

```

```

 for (int j = 0; j <= subsetSum; j++) {
 // 不选当前元素
 dp[i][j] = dp[i - 1][j];
 // 选当前元素（如果可以的话）
 if (j >= num) {
 dp[i][j] += dp[i - 1][j - num];
 }
 }
 }

 return dp[n][subsetSum];
}

/***
 * 递归+记忆化搜索实现
 */
public static int findTargetSumWaysDFS(int[] nums, int target) {
 if (nums == null || nums.length == 0) {
 return 0;
 }

 // 计算所有元素的和
 int sum = 0;
 for (int num : nums) {
 sum += num;
 }

 // 检查是否有解的条件
 if (sum < Math.abs(target) || (sum + target) % 2 != 0) {
 return 0;
 }

 int subsetSum = (sum + target) / 2;

 // 使用二维数组作为缓存，memo[i][j]表示前 i 个元素中和为 j 的子集数目
 Integer[][] memo = new Integer[nums.length][subsetSum + 1];

 // 调用递归辅助函数
 return dfs(nums, 0, 0, subsetSum, memo);
}

/***
 * 递归辅助函数

```

```

* @param nums 整数数组
* @param index 当前考虑的元素索引
* @param currentSum 当前子集和
* @param targetSum 目标子集和
* @param memo 缓存数组
* @return 可以组成目标子集和的子集数目
*/
private static int dfs(int[] nums, int index, int currentSum, int targetSum, Integer[][] memo) {
 // 基础情况：如果已经考虑完所有元素
 if (index == nums.length) {
 // 如果当前子集和等于目标子集和，返回 1，否则返回 0
 return currentSum == targetSum ? 1 : 0;
 }

 // 检查缓存
 if (memo[index][currentSum] != null) {
 return memo[index][currentSum];
 }

 // 计算当前状态的解
 int result = 0;

 // 选择不将当前元素加入子集
 result += dfs(nums, index + 1, currentSum, targetSum, memo);

 // 选择将当前元素加入子集（如果不会超过目标和）
 if (currentSum + nums[index] <= targetSum) {
 result += dfs(nums, index + 1, currentSum + nums[index], targetSum, memo);
 }

 // 缓存结果
 memo[index][currentSum] = result;
 return result;
}

/**
 * 另一种递归实现方式，直接计算表达式数目
 */
public static int findTargetSumWaysDFS2(int[] nums, int target) {
 if (nums == null || nums.length == 0) {
 return 0;
 }
}

```

```

// 使用 HashMap 作为缓存，键为"index, currentSum"，值为该状态下的表达式数目
Map<String, Integer> memo = new HashMap<>();

// 调用递归辅助函数
return dfs2(nums, 0, 0, target, memo);
}

/***
 * 另一种递归辅助函数实现
 */
private static int dfs2(int[] nums, int index, int currentSum, int target, Map<String, Integer> memo) {
 // 基础情况：如果已经考虑完所有元素
 if (index == nums.length) {
 // 如果当前和等于目标和，返回 1，否则返回 0
 return currentSum == target ? 1 : 0;
 }

 // 生成缓存键
 String key = index + "," + currentSum;

 // 检查缓存
 if (memo.containsKey(key)) {
 return memo.get(key);
 }

 // 选择在当前元素前添加'+'
 int add = dfs2(nums, index + 1, currentSum + nums[index], target, memo);

 // 选择在当前元素前添加 '-'
 int subtract = dfs2(nums, index + 1, currentSum - nums[index], target, memo);

 // 计算总表达式数目
 int total = add + subtract;

 // 缓存结果
 memo.put(key, total);
 return total;
}

/***
 * 另一种动态规划方法，使用二维数组记录到达每个和的路径数
*/

```

```

*/
public static int findTargetSumWaysAlternative(int[] nums, int target) {
 // 参数验证
 if (nums == null || nums.length == 0) {
 return 0;
 }

 // 计算所有元素的和，用于确定可能的和的范围
 int sum = 0;
 for (int num : nums) {
 sum += num;
 }

 // 检查是否有解的条件
 if (sum < Math.abs(target)) {
 return 0;
 }

 // 创建 DP 数组，dp[i][j] 表示前 i 个元素能组成和为 j 的表达式数目
 // 由于和可能为负数，我们需要进行偏移，将和范围从 [-sum, sum] 映射到 [0, 2*sum]
 int offset = sum;
 int[][] dp = new int[nums.length + 1][2 * sum + 1];

 // 初始状态：前 0 个元素能组成和为 0 的表达式有一个（空表达式）
 dp[0][offset] = 1;

 // 填充 DP 数组
 for (int i = 1; i <= nums.length; i++) {
 int num = nums[i - 1];
 for (int j = 0; j < 2 * sum + 1; j++) {
 // 如果前 i-1 个元素能组成和为 j 的表达式
 if (dp[i - 1][j] > 0) {
 // 添加'+'：和变为 j + num
 dp[i][j + num] += dp[i - 1][j];
 // 添加'-'：和变为 j - num
 dp[i][j - num] += dp[i - 1][j];
 }
 }
 }

 // 返回结果：前 n 个元素能组成和为 target 的表达式数目
 // 注意需要加上偏移量
 return dp[nums.length][target + offset];
}

```

```
}
```

```
}
```

```
=====
```

文件: Code27\_TargetSum.py

```
=====
```

```
LeetCode 494. 目标和
题目描述: 给你一个整数数组 nums 和一个整数 target 。
向数组中的每个整数前添加 '+' 或 '-' ，然后串联起所有整数，可以构造一个 表达式：
例如，nums = [2, 1] ，可以在 2 之前添加 '+' ，在 1 之前添加 '-' ，得到表达式 "+2-1" 。
返回可以通过上述方法构造的、运算结果等于 target 的不同 表达式 的数目。
链接: https://leetcode.cn/problems/target-sum/

解题思路:
这是一个背包问题的变种，我们可以将问题转化为:
找到一个子集，使得该子集中的元素和与其余元素和的差等于 target
设所有元素的和为 sum，子集和为 subsetSum，则:
subsetSum - (sum - subsetSum) = target
即 2*subsetSum = sum + target
因此 subsetSum = (sum + target) / 2

所以问题转化为：找到和为 subsetSum 的子集数目
这是一个 0-1 背包问题（每个元素只能选或不选）

状态定义: dp[i] 表示和为 i 的子集数目
状态转移方程: dp[i] += dp[i - num]，其中 num 是当前元素，且 i >= num
初始状态: dp[0] = 1 表示和为 0 的子集有一个（空集）

时间复杂度: O(n * target)，其中 n 是数组长度
空间复杂度: O(target)，使用一维 DP 数组
```

```
from typing import List, Dict, Optional
```

```
def find_target_sum_ways(nums: List[int], target: int) -> int:
 """
```

```
 计算可以通过添加'+'或'-'使得表达式结果等于 target 的不同表达式数目
```

Args:

nums: 整数数组

target: 目标和

Returns:

int: 不同表达式的数目

"""

# 参数验证

```
if not nums:
 return 0
```

# 计算所有元素的和

```
total_sum = sum(nums)
```

# 检查是否有解的条件

# 1. total\_sum + target 必须是非负数

# 2. total\_sum + target 必须是偶数

```
if total_sum < abs(target) or (total_sum + target) % 2 != 0:
 return 0
```

# 计算目标子集和

```
subset_sum = (total_sum + target) // 2
```

# 创建一维 DP 数组, dp[i] 表示和为 i 的子集数目

```
dp = [0] * (subset_sum + 1)
```

# 初始状态: 和为 0 的子集有一个 (空集)

```
dp[0] = 1
```

# 对于每个元素, 逆序遍历子集和 (0-1 背包问题)

```
for num in nums:
```

# 从大到小遍历, 避免重复使用同一个元素

```
for i in range(subset_sum, num - 1, -1):
 dp[i] += dp[i - num]
```

# 返回结果: 和为 subset\_sum 的子集数目

```
return dp[subset_sum]
```

```
def find_target_sum_ways_optimized(nums: List[int], target: int) -> int:
```

"""

优化版本: 处理可能的大数问题 (使用适当的数据类型)

"""

```
if not nums:
 return 0
```

```
total_sum = sum(nums)
```

```

if total_sum < abs(target) or (total_sum + target) % 2 != 0:
 return 0

subset_sum = (total_sum + target) // 2

使用 Python 的 int 类型（可以处理大数）
dp = [0] * (subset_sum + 1)
dp[0] = 1

for num in nums:
 for i in range(subset_sum, num - 1, -1):
 dp[i] += dp[i - num]

return dp[subset_sum]

def find_target_sum_ways_2d(nums: List[int], target: int) -> int:
 """
 二维 DP 数组实现
 dp[i][j] 表示前 i 个元素中和为 j 的子集数目
 """
 if not nums:
 return 0

 total_sum = sum(nums)

 if total_sum < abs(target) or (total_sum + target) % 2 != 0:
 return 0

 subset_sum = (total_sum + target) // 2
 n = len(nums)

 # 创建二维 DP 数组
 dp = [[0] * (subset_sum + 1) for _ in range(n + 1)]

 # 初始化：前 0 个元素中和为 0 的子集有一个（空集）
 dp[0][0] = 1

 # 填充 DP 数组
 for i in range(1, n + 1):
 num = nums[i - 1]
 for j in range(subset_sum + 1):
 dp[i][j] = dp[i - 1][j]
 if j - num >= 0:
 dp[i][j] += dp[i - 1][j - num]

```

```

不选当前元素
dp[i][j] = dp[i - 1][j]
选当前元素（如果可以的话）
if j >= num:
 dp[i][j] += dp[i - 1][j - num]

return dp[n][subset_sum]

def find_target_sum_ways_dfs(nums: List[int], target: int) -> int:
 """
 递归+记忆化搜索实现
 """

 if not nums:
 return 0

 # 计算所有元素的和
 total_sum = sum(nums)

 # 检查是否有解的条件
 if total_sum < abs(target) or (total_sum + target) % 2 != 0:
 return 0

 subset_sum = (total_sum + target) // 2
 n = len(nums)

 # 使用二维列表作为缓存，memo[i][j]表示前 i 个元素中和为 j 的子集数目
 memo = [[-1] * (subset_sum + 1) for _ in range(n)]

 def dfs(index: int, current_sum: int) -> int:
 """
 递归辅助函数
 # 基础情况：如果已经考虑完所有元素
 if index == n:
 # 如果当前子集和等于目标子集和，返回 1，否则返回 0
 return 1 if current_sum == subset_sum else 0

 # 检查缓存
 if memo[index][current_sum] != -1:
 return memo[index][current_sum]

 # 计算当前状态的解
 result = 0
 """

 # 不选当前元素
 dp[i][j] = dp[i - 1][j]
 # 选当前元素（如果可以的话）
 if j >= num:
 dp[i][j] += dp[i - 1][j - num]

 return dp[n][subset_sum]

```

```
选择不将当前元素加入子集
result += dfs(index + 1, current_sum)

选择将当前元素加入子集（如果不会超过目标和）
if current_sum + nums[index] <= subset_sum:
 result += dfs(index + 1, current_sum + nums[index])

缓存结果
memo[index][current_sum] = result
return result

调用递归函数
return dfs(0, 0)
```

```
def find_target_sum_ways_dfs2(nums: List[int], target: int) -> int:
 """
另一种递归实现方式，直接计算表达式数目
 """
 if not nums:
 return 0

 # 使用字典作为缓存，键为"index, current_sum"，值为该状态下的表达式数目
 memo: Dict[str, int] = {}

 def dfs2(index: int, current_sum: int) -> int:
 """
另一种递归辅助函数实现"""
 # 基础情况：如果已经考虑完所有元素
 if index == len(nums):
 # 如果当前和等于目标和，返回 1，否则返回 0
 return 1 if current_sum == target else 0

 # 生成缓存键
 key = f'{index}, {current_sum}'

 # 检查缓存
 if key in memo:
 return memo[key]

 # 选择在当前元素前添加'+' 或 '-'
 add = dfs2(index + 1, current_sum + nums[index])
 minus = dfs2(index + 1, current_sum - nums[index])

 # 将结果缓存起来
 memo[key] = add + minus

 return add + minus

 return dfs2(0, target)
```

```

subtract = dfs2(index + 1, current_sum - nums[index])

计算总表达式数目
total = add + subtract

缓存结果
memo[key] = total
return total

调用递归函数
return dfs2(0, 0)

```

```

def find_target_sum_ways_alternative(nums: List[int], target: int) -> int:
 """
 另一种动态规划方法，使用二维数组记录到达每个和的路径数
 """

 # 参数验证
 if not nums:
 return 0

 # 计算所有元素的和，用于确定可能的和的范围
 total_sum = sum(nums)

 # 检查是否有解的条件
 if total_sum < abs(target):
 return 0

 # 创建 DP 数组，dp[i][j] 表示前 i 个元素能组成和为 j 的表达式数目
 # 由于和可能为负数，我们需要进行偏移，将和范围从 [-total_sum, total_sum] 映射到 [0, 2*total_sum]
 offset = total_sum
 dp = [[0] * (2 * total_sum + 1) for _ in range(len(nums) + 1)]

 # 初始状态：前 0 个元素能组成和为 0 的表达式有一个（空表达式）
 dp[0][offset] = 1

 # 填充 DP 数组
 for i in range(1, len(nums) + 1):
 num = nums[i - 1]
 for j in range(2 * total_sum + 1):
 # 如果前 i-1 个元素能组成和为 j 的表达式
 if dp[i - 1][j] > 0:
 # 添加 '+'：和变为 j + num
 dp[i][j + num] += dp[i - 1][j]

```

```

 if j + num < 2 * total_sum + 1:
 dp[i][j + num] += dp[i - 1][j]
 # 添加'-'：和变为 j - num
 if j - num >= 0:
 dp[i][j - num] += dp[i - 1][j]

 # 返回结果：前 n 个元素能组成和为 target 的表达式数目
 # 注意需要加上偏移量
 return dp[len(nums)][target + offset]

测试代码
if __name__ == "__main__":
 # 测试用例 1
 nums1 = [1, 1, 1, 1, 1]
 target1 = 3
 print(f"测试用例 1 结果: {find_target_sum_ways(nums1, target1)}") # 预期输出: 5

 # 测试用例 2
 nums2 = [1]
 target2 = 1
 print(f"测试用例 2 结果: {find_target_sum_ways(nums2, target2)}") # 预期输出: 1

 # 测试用例 3
 nums3 = [1, 2, 3, 4, 5]
 target3 = 3
 print(f"测试用例 3 结果: {find_target_sum_ways(nums3, target3)}") # 预期输出: 3

```

文件: Code28\_PartitionEqualSubsetSum.cpp

```

#include <iostream>
#include <vector>
#include <algorithm>

// LeetCode 416. 分割等和子集
// 题目描述: 给你一个 只包含正整数 的 非空 数组 nums 。请你判断是否可以将这个数组分割成两个子集，使得两个子集的元素和相等。
// 链接: https://leetcode.cn/problems/partition-equal-subset-sum/
//
// 解题思路:
// 这是一个 0-1 背包问题的应用，问题可以转化为:

```

```
// 1. 计算数组的总和 sum
// 2. 如果 sum 是奇数，那么无法将数组分成两个和相等的子集，直接返回 false
// 3. 如果 sum 是偶数，那么问题转化为：是否存在一个子集，使得其和为 sum/2
//
// 状态定义：dp[i] 表示是否可以从数组中选择一些元素，使得它们的和为 i
// 状态转移方程：dp[i] = dp[i] || dp[i - num]，其中 num 是当前元素，且 i >= num
// 初始状态：dp[0] = true，表示和为 0 的子集存在（空集）
//
// 时间复杂度：O(n * target)，其中 n 是数组长度，target 是数组和的一半
// 空间复杂度：O(target)，使用一维 DP 数组
```

```
using namespace std;
```

```
/***
 * 判断是否可以将数组分割成两个和相等的子集
 * @param nums 非空正整数数组
 * @return 是否可以分割成两个和相等的子集
 */
```

```
bool canPartition(vector<int>& nums) {
```

```
 // 参数验证
 if (nums.size() < 2) {
 return false;
 }
```

```
 // 计算数组总和
```

```
 int sum = 0;
 for (int num : nums) {
 sum += num;
 }
```

```
 // 如果总和是奇数，无法分成两个和相等的子集
```

```
 if (sum % 2 != 0) {
 return false;
 }
```

```
 // 计算目标和：总和的一半
```

```
 int target = sum / 2;
```

```
 // 创建一维 DP 数组，dp[i] 表示是否可以从数组中选择一些元素，使得它们的和为 i
 vector<bool> dp(target + 1, false);
```

```
 // 初始状态：和为 0 的子集存在（空集）
 dp[0] = true;
```

```

// 对于每个元素，逆序遍历目标和（0-1 背包问题）
for (int num : nums) {
 for (int i = target; i >= num; i--) {
 // 状态转移：如果 dp[i - num] 为 true，说明可以组成和为 i - num 的子集，
 // 那么再加上当前元素 num，就可以组成和为 i 的子集
 dp[i] = dp[i] || dp[i - num];
 }
}

// 返回是否可以组成和为 target 的子集
return dp[target];
}

/***
 * 优化版本：提前剪枝
 */
bool canPartitionOptimized(vector<int>& nums) {
 if (nums.size() < 2) {
 return false;
 }

 int sum = 0;
 int maxNum = 0;
 for (int num : nums) {
 sum += num;
 maxNum = max(maxNum, num);
 }

 // 如果总和是奇数，无法分成两个和相等的子集
 if (sum % 2 != 0) {
 return false;
 }

 int target = sum / 2;

 // 如果最大元素大于目标和，无法分成两个和相等的子集
 if (maxNum > target) {
 return false;
 }

 // 排序数组，方便后续剪枝
 sort(nums.begin(), nums.end());
}

```

```

vector<bool> dp(target + 1, false);
dp[0] = true;

for (int num : nums) {
 // 剪枝: 如果当前元素已经大于目标和, 可以跳过
 if (num > target) {
 continue;
 }

 for (int i = target; i >= num; i--) {
 dp[i] = dp[i] || dp[i - num];
 }
}

// 提前结束: 如果已经找到解, 可以直接返回 true
if (dp[target]) {
 return true;
}

return dp[target];
}

/***
 * 二维 DP 数组实现
 * dp[i][j] 表示前 i 个元素中是否可以选择一些元素, 使得它们的和为 j
 */
bool canPartition2D(vector<int>& nums) {
 if (nums.size() < 2) {
 return false;
 }

 int sum = 0;
 for (int num : nums) {
 sum += num;
 }

 if (sum % 2 != 0) {
 return false;
 }

 int target = sum / 2;
 int n = nums.size();

```

```

// 创建二维 DP 数组
vector<vector<bool>> dp(n + 1, vector<bool>(target + 1, false));

// 初始化：前 0 个元素可以组成和为 0 的子集
for (int i = 0; i <= n; i++) {
 dp[i][0] = true;
}

// 填充 DP 数组
for (int i = 1; i <= n; i++) {
 int num = nums[i - 1];
 for (int j = 1; j <= target; j++) {
 // 不选当前元素
 dp[i][j] = dp[i - 1][j];
 // 选当前元素（如果可以的话）
 if (j >= num) {
 dp[i][j] = dp[i][j] || dp[i - 1][j - num];
 }
 }
}

// 提前结束：如果已经找到解，可以直接返回 true
if (dp[i][target]) {
 return true;
}

return dp[n][target];
}

/**
 * 递归+记忆化搜索实现
 */
bool canPartitionDFS(vector<int>& nums) {
 if (nums.size() < 2) {
 return false;
 }

 int sum = 0;
 for (int num : nums) {
 sum += num;
 }
}

```

```

if (sum % 2 != 0) {
 return false;
}

int target = sum / 2;
int n = nums.size();

// 使用二维数组作为缓存, memo[i][j] 表示从第 i 个元素开始, 是否可以组成和为 j 的子集
vector<vector<int>> memo(n, vector<int>(target + 1, -1)); // -1 表示未计算, 0 表示 false, 1 表示 true

// 定义 DFS 函数
function<bool(int, int)> dfs = [&](int index, int remaining) -> bool {
 // 基础情况: 如果剩余和为 0, 说明找到了一个子集
 if (remaining == 0) {
 return true;
 }

 // 基础情况: 如果已经考虑完所有元素或者剩余和小于 0, 返回 false
 if (index == n || remaining < 0) {
 return false;
 }

 // 检查缓存
 if (memo[index][remaining] != -1) {
 return memo[index][remaining] == 1;
 }

 // 尝试两种选择: 选或不选当前元素
 // 1. 选当前元素: 剩余和减去当前元素的值, 继续考虑下一个元素
 bool choose = dfs(index + 1, remaining - nums[index]);

 // 2. 不选当前元素: 剩余和不变, 继续考虑下一个元素
 bool notChoose = dfs(index + 1, remaining);

 // 缓存结果
 memo[index][remaining] = (choose || notChoose) ? 1 : 0;
 return memo[index][remaining] == 1;
};

// 调用递归函数
return dfs(0, target);
}

```

```

/***
 * 位运算优化的 DP 实现
 * 每个二进制位表示是否可以组成对应索引的和
 */
bool canPartitionBit(vector<int>& nums) {
 if (nums.size() < 2) {
 return false;
 }

 int sum = 0;
 for (int num : nums) {
 sum += num;
 }

 if (sum % 2 != 0) {
 return false;
 }

 int target = sum / 2;

 // 使用位集，每个位表示是否可以组成对应的和
 // bits[0] 表示和为 0, bits[i] 表示和为 i
 // 初始时，只有和为 0 的情况是可能的
 long long bits = 1; // 0b000...0001, 表示和为 0 是可以的

 for (int num : nums) {
 // 位运算：将当前 bits 左移 num 位，并与原 bits 进行或操作
 // 这样，新的 bits 中的第 i 位为 1 当且仅当原来的 bits 中的第 i 位为 1（不选当前元素）
 // 或者原来的 bits 中的第 i-num 位为 1（选当前元素）
 bits |= bits << num;

 // 检查目标和是否已经可达
 if ((bits & (1LL << target)) != 0) {
 return true;
 }
 }

 // 检查目标和是否可达
 return (bits & (1LL << target)) != 0;
}

int main() {

```

```

// 测试用例 1
vector<int> nums1 = {1, 5, 11, 5};
cout << "测试用例 1 结果: " << (canPartition(nums1) ? "true" : "false") << endl; // 预期输出:
true

// 测试用例 2
vector<int> nums2 = {1, 2, 3, 5};
cout << "测试用例 2 结果: " << (canPartition(nums2) ? "true" : "false") << endl; // 预期输出:
false

// 测试用例 3
vector<int> nums3 = {1, 2, 5};
cout << "测试用例 3 结果: " << (canPartition(nums3) ? "true" : "false") << endl; // 预期输出:
false

return 0;
}

```

=====

文件: Code28\_PartitionEqualSubsetSum.java

```

=====
package class073;

import java.util.Arrays;

// LeetCode 416. 分割等和子集
// 题目描述: 给你一个 只包含正整数 的 非空 数组 nums 。请你判断是否可以将这个数组分割成两个子集,
// 使得两个子集的元素和相等。
// 链接: https://leetcode.cn/problems/partition-equal-subset-sum/
//
// 解题思路:
// 这是一个 0-1 背包问题的应用, 问题可以转化为:
// 1. 计算数组的总和 sum
// 2. 如果 sum 是奇数, 那么无法将数组分成两个和相等的子集, 直接返回 false
// 3. 如果 sum 是偶数, 那么问题转化为: 是否存在一个子集, 使得其和为 sum/2
//
// 状态定义: dp[i] 表示是否可以从数组中选择一些元素, 使得它们的和为 i
// 状态转移方程: dp[i] = dp[i] || dp[i - num], 其中 num 是当前元素, 且 i >= num
// 初始状态: dp[0] = true, 表示和为 0 的子集存在 (空集)
//
// 时间复杂度: O(n * target), 其中 n 是数组长度, target 是数组和的一半
// 空间复杂度: O(target), 使用一维 DP 数组

```

```
public class Code28_PartitionEqualSubsetSum {

 // 主方法，用于测试
 public static void main(String[] args) {
 // 测试用例 1
 int[] nums1 = {1, 5, 11, 5};
 System.out.println("测试用例 1 结果: " + canPartition(nums1)); // 预期输出: true

 // 测试用例 2
 int[] nums2 = {1, 2, 3, 5};
 System.out.println("测试用例 2 结果: " + canPartition(nums2)); // 预期输出: false

 // 测试用例 3
 int[] nums3 = {1, 2, 5};
 System.out.println("测试用例 3 结果: " + canPartition(nums3)); // 预期输出: false
 }

 /**
 * 判断是否可以将数组分割成两个和相等的子集
 * @param nums 非空正整数数组
 * @return 是否可以分割成两个和相等的子集
 */
 public static boolean canPartition(int[] nums) {
 // 参数验证
 if (nums == null || nums.length < 2) {
 return false;
 }

 // 计算数组总和
 int sum = 0;
 for (int num : nums) {
 sum += num;
 }

 // 如果总和是奇数，无法分成两个和相等的子集
 if (sum % 2 != 0) {
 return false;
 }

 // 计算目标和：总和的一半
 int target = sum / 2;
```

```

// 创建一维 DP 数组，dp[i] 表示是否可以从数组中选择一些元素，使得它们的和为 i
boolean[] dp = new boolean[target + 1];

// 初始状态：和为 0 的子集存在（空集）
dp[0] = true;

// 对于每个元素，逆序遍历目标和（0-1 背包问题）
for (int num : nums) {
 for (int i = target; i >= num; i--) {
 // 状态转移：如果 dp[i - num] 为 true，说明可以组成和为 i - num 的子集，
 // 那么再加上当前元素 num，就可以组成和为 i 的子集
 dp[i] = dp[i] || dp[i - num];
 }
}

// 返回是否可以组成和为 target 的子集
return dp[target];
}

/**
 * 优化版本：提前剪枝
 */
public static boolean canPartitionOptimized(int[] nums) {
 if (nums == null || nums.length < 2) {
 return false;
 }

 int sum = 0;
 int maxNum = 0;
 for (int num : nums) {
 sum += num;
 maxNum = Math.max(maxNum, num);
 }

 // 如果总和是奇数，无法分成两个和相等的子集
 if (sum % 2 != 0) {
 return false;
 }

 int target = sum / 2;

 // 如果最大元素大于目标和，无法分成两个和相等的子集
 if (maxNum > target) {

```

```

 return false;
 }

 // 排序数组，方便后续剪枝
 Arrays.sort(nums);

 boolean[] dp = new boolean[target + 1];
 dp[0] = true;

 for (int num : nums) {
 // 剪枝：如果当前元素已经大于目标和，可以跳过
 if (num > target) {
 continue;
 }

 for (int i = target; i >= num; i--) {
 dp[i] = dp[i] || dp[i - num];
 }
 }

 // 提前结束：如果已经找到解，可以直接返回 true
 if (dp[target]) {
 return true;
 }
}

return dp[target];
}

/***
 * 二维 DP 数组实现
 * dp[i][j] 表示前 i 个元素中是否可以选择一些元素，使得它们的和为 j
 */
public static boolean canPartition2D(int[] nums) {
 if (nums == null || nums.length < 2) {
 return false;
 }

 int sum = 0;
 for (int num : nums) {
 sum += num;
 }

 if (sum % 2 != 0) {

```

```

 return false;
 }

 int target = sum / 2;
 int n = nums.length;

 // 创建二维 DP 数组
 boolean[][] dp = new boolean[n + 1][target + 1];

 // 初始化: 前 0 个元素可以组成和为 0 的子集
 for (int i = 0; i <= n; i++) {
 dp[i][0] = true;
 }

 // 填充 DP 数组
 for (int i = 1; i <= n; i++) {
 int num = nums[i - 1];
 for (int j = 1; j <= target; j++) {
 // 不选当前元素
 dp[i][j] = dp[i - 1][j];
 // 选当前元素 (如果可以的话)
 if (j >= num) {
 dp[i][j] = dp[i][j] || dp[i - 1][j - num];
 }
 }
 }

 // 提前结束: 如果已经找到解, 可以直接返回 true
 if (dp[i][target]) {
 return true;
 }
}

return dp[n][target];
}

/**
 * 递归+记忆化搜索实现
 */
public static boolean canPartitionDFS(int[] nums) {
 if (nums == null || nums.length < 2) {
 return false;
 }
}

```

```

int sum = 0;
for (int num : nums) {
 sum += num;
}

if (sum % 2 != 0) {
 return false;
}

int target = sum / 2;
int n = nums.length;

// 使用二维数组作为缓存, memo[i][j]表示从第 i 个元素开始, 是否可以组成和为 j 的子集
Boolean[][] memo = new Boolean[n][target + 1];

// 调用递归辅助函数
return dfs(nums, 0, target, memo);
}

/**
 * 递归辅助函数
 * @param nums 数组
 * @param index 当前考虑的元素索引
 * @param remaining 剩余需要达到的和
 * @param memo 缓存数组
 * @return 是否可以从当前索引开始, 组成和为 remaining 的子集
 */
private static boolean dfs(int[] nums, int index, int remaining, Boolean[][] memo) {
 // 基础情况: 如果剩余和为 0, 说明找到了一个子集
 if (remaining == 0) {
 return true;
 }

 // 基础情况: 如果已经考虑完所有元素或者剩余和小于 0, 返回 false
 if (index == nums.length || remaining < 0) {
 return false;
 }

 // 检查缓存
 if (memo[index][remaining] != null) {
 return memo[index][remaining];
 }

 for (int i = index; i < nums.length; i++) {
 if (dfs(nums, i + 1, remaining - nums[i], memo)) {
 memo[index][remaining] = true;
 break;
 }
 }
}

```

```

// 尝试两种选择：选或不选当前元素
// 1. 选当前元素：剩余和减去当前元素的值，继续考虑下一个元素
boolean choose = dfs(nums, index + 1, remaining - nums[index], memo);

// 2. 不选当前元素：剩余和不变，继续考虑下一个元素
boolean notChoose = dfs(nums, index + 1, remaining, memo);

// 缓存结果
memo[index][remaining] = choose || notChoose;
return memo[index][remaining];
}

/**
 * 位运算优化的 DP 实现
 * 每个二进制位表示是否可以组成对应索引的和
 */
public static boolean canPartitionBit(int[] nums) {
 if (nums == null || nums.length < 2) {
 return false;
 }

 int sum = 0;
 for (int num : nums) {
 sum += num;
 }

 if (sum % 2 != 0) {
 return false;
 }

 int target = sum / 2;

 // 使用位集，每个位表示是否可以组成对应的和
 // bits[0] 表示和为 0，bits[i] 表示和为 i
 // 初始时，只有和为 0 的情况是可能的
 int bits = 1; // 0b000...0001，表示和为 0 是可以的

 for (int num : nums) {
 // 位运算：将当前 bits 左移 num 位，并与原 bits 进行或操作
 // 这样，新的 bits 中的第 i 位为 1 当且仅当原来的 bits 中的第 i 位为 1（不选当前元素）
 // 或者原来的 bits 中的第 i-num 位为 1（选当前元素）
 bits |= bits << num;
 }
}

```

```

 // 检查目标和是否已经可达
 if ((bits & (1 << target)) != 0) {
 return true;
 }
}

// 检查目标和是否可达
return (bits & (1 << target)) != 0;
}
}

```

=====

文件: Code28\_PartitionEqualSubsetSum.py

=====

```

LeetCode 416. 分割等和子集
题目描述: 给你一个 只包含正整数 的 非空 数组 nums 。请你判断是否可以将这个数组分割成两个子集,
使得两个子集的元素和相等。
链接: https://leetcode.cn/problems/partition-equal-subset-sum/
#
解题思路:
这是一个 0-1 背包问题的应用, 问题可以转化为:
1. 计算数组的总和 sum
2. 如果 sum 是奇数, 那么无法将数组分成两个和相等的子集, 直接返回 false
3. 如果 sum 是偶数, 那么问题转化为: 是否存在一个子集, 使得其和为 sum/2
#
状态定义: dp[i] 表示是否可以从数组中选择一些元素, 使得它们的和为 i
状态转移方程: dp[i] = dp[i] || dp[i - num], 其中 num 是当前元素, 且 i >= num
初始状态: dp[0] = true, 表示和为 0 的子集存在 (空集)
#
时间复杂度: O(n * target), 其中 n 是数组长度, target 是数组和的一半
空间复杂度: O(target), 使用一维 DP 数组

```

from typing import List, Optional

```
def can_partition(nums: List[int]) -> bool:
```

```
"""

```

判断是否可以将数组分割成两个和相等的子集

Args:

nums: 非空正整数数组

Returns:

bool: 是否可以分割成两个和相等的子集

"""

# 参数验证

```
if len(nums) < 2:
 return False
```

# 计算数组总和

```
total_sum = sum(nums)
```

# 如果总和是奇数，无法分成两个和相等的子集

```
if total_sum % 2 != 0:
 return False
```

# 计算目标和：总和的一半

```
target = total_sum // 2
```

# 创建一维 DP 数组，dp[i] 表示是否可以从数组中选择一些元素，使得它们的和为 i

```
dp = [False] * (target + 1)
```

# 初始状态：和为 0 的子集存在（空集）

```
dp[0] = True
```

# 对于每个元素，逆序遍历目标和（0-1 背包问题）

```
for num in nums:
```

# 逆序遍历，避免重复使用同一个元素

```
for i in range(target, num - 1, -1):
```

# 状态转移：如果 dp[i - num] 为 True，说明可以组成和为 i - num 的子集，

# 那么再加上当前元素 num，就可以组成和为 i 的子集

```
dp[i] = dp[i] or dp[i - num]
```

# 返回是否可以组成和为 target 的子集

```
return dp[target]
```

```
def can_partition_optimized(nums: List[int]) -> bool:
```

"""

优化版本：提前剪枝

"""

```
if len(nums) < 2:
 return False
```

```
total_sum = 0
```

```

max_num = 0
for num in nums:
 total_sum += num
 max_num = max(max_num, num)

如果总和是奇数，无法分成两个和相等的子集
if total_sum % 2 != 0:
 return False

target = total_sum // 2

如果最大元素大于目标和，无法分成两个和相等的子集
if max_num > target:
 return False

排序数组，方便后续剪枝
nums.sort()

dp = [False] * (target + 1)
dp[0] = True

for num in nums:
 # 剪枝：如果当前元素已经大于目标和，可以跳过
 if num > target:
 continue

 for i in range(target, num - 1, -1):
 dp[i] = dp[i] or dp[i - num]

提前结束：如果已经找到解，可以直接返回 True
if dp[target]:
 return True

return dp[target]

```

```

def can_partition_2d(nums: List[int]) -> bool:
 """
 二维 DP 数组实现
 dp[i][j] 表示前 i 个元素中是否可以选择一些元素，使得它们的和为 j
 """
 if len(nums) < 2:
 return False

```

```

total_sum = sum(nums)

if total_sum % 2 != 0:
 return False

target = total_sum // 2
n = len(nums)

创建二维 DP 数组
dp = [[False] * (target + 1) for _ in range(n + 1)]

初始化: 前 0 个元素可以组成和为 0 的子集
for i in range(n + 1):
 dp[i][0] = True

填充 DP 数组
for i in range(1, n + 1):
 num = nums[i - 1]
 for j in range(1, target + 1):
 # 不选当前元素
 dp[i][j] = dp[i - 1][j]
 # 选当前元素 (如果可以的话)
 if j >= num:
 dp[i][j] = dp[i][j] or dp[i - 1][j - num]

提前结束: 如果已经找到解, 可以直接返回 True
if dp[n][target]:
 return True

return dp[n][target]

```

```
def can_partition_dfs(nums: List[int]) -> bool:
```

```
"""

```

```
递归+记忆化搜索实现
"""

```

```
"""

```

```
if len(nums) < 2:
 return False
```

```
total_sum = sum(nums)
```

```
if total_sum % 2 != 0:
```

```

 return False

target = total_sum // 2
n = len(nums)

使用二维列表作为缓存, memo[i][j]表示从第 i 个元素开始, 是否可以组成和为 j 的子集
memo = [[-1 for _ in range(target + 1)] for _ in range(n)] # -1 表示未计算, 0 表示 False, 1 表示 True

def dfs(index: int, remaining: int) -> bool:
 """递归辅助函数"""
 # 基础情况: 如果剩余和为 0, 说明找到了一个子集
 if remaining == 0:
 return True

 # 基础情况: 如果已经考虑完所有元素或者剩余和小于 0, 返回 False
 if index == n or remaining < 0:
 return False

 # 检查缓存
 if memo[index][remaining] != -1:
 return memo[index][remaining] == 1

 # 尝试两种选择: 选或不选当前元素
 # 1. 选当前元素: 剩余和减去当前元素的值, 继续考虑下一个元素
 choose = dfs(index + 1, remaining - nums[index])

 # 2. 不选当前元素: 剩余和不变, 继续考虑下一个元素
 not_choose = dfs(index + 1, remaining)

 # 缓存结果
 memo[index][remaining] = 1 if (choose or not_choose) else 0
 return memo[index][remaining] == 1

 # 调用递归函数
return dfs(0, target)

```

```
def can_partition_bit(nums: List[int]) -> bool:
```

```
"""
```

位运算优化的 DP 实现

每个二进制位表示是否可以组成对应索引的和

```
"""
```

```

if len(nums) < 2:
 return False

total_sum = sum(nums)

if total_sum % 2 != 0:
 return False

target = total_sum // 2

使用位集，每个位表示是否可以组成对应的和
bits 的第 i 位为 1 表示可以组成和为 i 的子集
初始时，只有和为 0 的情况是可能的
bits = 1 # 0b000...0001，表示和为 0 是可以的

for num in nums:
 # 位运算：将当前 bits 左移 num 位，并与原 bits 进行或操作
 # 这样，新的 bits 中的第 i 位为 1 当且仅当原来的 bits 中的第 i 位为 1（不选当前元素）
 # 或者原来的 bits 中的第 i-num 位为 1（选当前元素）
 bits |= bits << num

 # 检查目标和是否已经可达
 if (bits & (1 << target)) != 0:
 return True

检查目标和是否可达
return (bits & (1 << target)) != 0

测试代码
if __name__ == "__main__":
 # 测试用例 1
 nums1 = [1, 5, 11, 5]
 print(f"测试用例 1 结果: {can_partition(nums1)}") # 预期输出: True

 # 测试用例 2
 nums2 = [1, 2, 3, 5]
 print(f"测试用例 2 结果: {can_partition(nums2)}") # 预期输出: False

 # 测试用例 3
 nums3 = [1, 2, 5]
 print(f"测试用例 3 结果: {can_partition(nums3)}") # 预期输出: False

```

文件: Code29\_LastStoneWeightII.cpp

```
#include <iostream>
#include <vector>
#include <functional>

// LeetCode 1049. 最后一块石头的重量 II
// 题目描述: 有一堆石头, 每块石头的重量都是正整数。每一回合, 从中选出任意两块石头, 然后将它们一起粉碎。
// 假设石头的重量分别为 x 和 y, 且 x <= y。那么粉碎的可能结果如下:
// - 如果 x == y, 那么两块石头都会被完全粉碎;
// - 如果 x != y, 那么重量为 x 的石头会被完全粉碎, 而重量为 y 的石头新重量为 y - x。
// 最后, 最多只会剩下一块石头。返回此石头的最小可能重量。如果没有石头剩下, 就返回 0。
// 链接: https://leetcode.cn/problems/last-stone-weight-ii/
//
// 解题思路:
// 这是一个 0-1 背包问题的变种, 问题可以转化为:
// 将石头分成两组, 使得两组的重量差最小。那么剩下的石头重量就是两组重量的差的最小值。
// 等价于: 找到一组石头, 使得它们的重量尽可能接近总重量的一半。
//
// 状态定义: dp[i] 表示是否可以组成和为 i 的子集
// 状态转移方程: dp[i] = dp[i] || dp[i - stone], 其中 stone 是当前石头的重量, 且 i >= stone
// 初始状态: dp[0] = true, 表示和为 0 的子集存在(空集)
//
// 时间复杂度: O(n * target), 其中 n 是石头的数量, target 是总重量的一半
// 空间复杂度: O(target), 使用一维 DP 数组
```

```
using namespace std;
```

```
/***
 * 计算最后一块石头的最小可能重量
 * @param stones 石头重量数组
 * @return 最后一块石头的最小可能重量
 */
```

```
int lastStoneWeightII(vector<int>& stones) {
 // 参数验证
 if (stones.empty()) {
 return 0;
 }
```

```
 // 计算石头总重量
```

```

int sum = 0;
for (int stone : stones) {
 sum += stone;
}

// 计算目标和: 总重量的一半 (向下取整)
int target = sum / 2;

// 创建一维 DP 数组, dp[i] 表示是否可以组成和为 i 的子集
vector<bool> dp(target + 1, false);

// 初始状态: 和为 0 的子集存在 (空集)
dp[0] = true;

// 对于每个石头, 逆序遍历目标和 (0-1 背包问题)
for (int stone : stones) {
 for (int i = target; i >= stone; i--) {
 // 状态转移: 如果 dp[i - stone] 为 true, 说明可以组成和为 i - stone 的子集,
 // 那么再加上当前石头 stone, 就可以组成和为 i 的子集
 dp[i] = dp[i] || dp[i - stone];
 }
}

// 找到最大的 i, 使得 dp[i] 为 true, 其中 i <= target
// 剩下的石头重量就是 sum - 2 * i
for (int i = target; i >= 0; i--) {
 if (dp[i]) {
 return sum - 2 * i;
 }
}

return 0; // 理论上不会到达这里
}

/***
 * 二维 DP 数组实现
 * dp[i][j] 表示前 i 个石头中是否可以选择一些石头, 使得它们的和为 j
 */
int lastStoneWeightII2D(vector<int>& stones) {
 if (stones.empty()) {
 return 0;
 }

 int sum = 0;
 for (int stone : stones) {
 sum += stone;
 }

 int target = sum / 2;
 vector<vector<bool>> dp(stones.size() + 1, vector<bool>(target + 1, false));

 dp[0][0] = true;

 for (int i = 1; i <= stones.size(); i++) {
 for (int j = 0; j <= target; j++) {
 if (j - stones[i - 1] <= 0) {
 dp[i][j] = dp[i - 1][j];
 } else {
 dp[i][j] = dp[i - 1][j] || dp[i - 1][j - stones[i - 1]];
 }
 }
 }

 for (int i = stones.size(); i >= 0; i--) {
 if (dp[i][target]) {
 return sum - 2 * i;
 }
 }
}

```

```

int sum = 0;
for (int stone : stones) {
 sum += stone;
}

int target = sum / 2;
int n = stones.size();

// 创建二维 DP 数组
vector<vector<bool>> dp(n + 1, vector<bool>(target + 1, false));

// 初始化: 前 0 个石头可以组成和为 0 的子集
for (int i = 0; i <= n; i++) {
 dp[i][0] = true;
}

// 填充 DP 数组
for (int i = 1; i <= n; i++) {
 int stone = stones[i - 1];
 for (int j = 1; j <= target; j++) {
 // 不选当前石头
 dp[i][j] = dp[i - 1][j];
 // 选当前石头 (如果可以的话)
 if (j >= stone) {
 dp[i][j] = dp[i][j] || dp[i - 1][j - stone];
 }
 }
}

// 找到最大的 j, 使得 dp[n][j] 为 true, 其中 j <= target
for (int j = target; j >= 0; j--) {
 if (dp[n][j]) {
 return sum - 2 * j;
 }
}

return 0;
}

/***
 * 优化版本: 提前剪枝
 */
int lastStoneWeightIIOptimized(vector<int>& stones) {

```

```

if (stones.empty()) {
 return 0;
}

// 计算石头总重量
int sum = 0;
for (int stone : stones) {
 sum += stone;
}

// 计算目标和：总重量的一半（向下取整）
int target = sum / 2;

// 创建一维 DP 数组，dp[i] 表示是否可以组成和为 i 的子集
vector<bool> dp(target + 1, false);
dp[0] = true;

// 使用一个变量记录当前可以达到的最大和
int currentMax = 0;

for (int stone : stones) {
 // 逆序遍历目标和
 for (int i = target; i >= stone; i--) {
 if (dp[i - stone] && !dp[i]) {
 dp[i] = true;
 currentMax = max(currentMax, i);
 // 提前剪枝：如果已经可以达到目标和，直接返回结果
 if (currentMax == target) {
 return sum - 2 * target;
 }
 }
 }
}

// 返回最小可能的最后一块石头重量
return sum - 2 * currentMax;
}

/**
 * 递归+记忆化搜索实现
 */
int lastStoneWeightIIDFS(vector<int>& stones) {
 if (stones.empty()) {

```

```

 return 0;
}

int sum = 0;
for (int stone : stones) {
 sum += stone;
}

int target = sum / 2;
int n = stones.size();

// 使用二维数组作为缓存, memo[i][j]表示从第 i 个石头开始, 是否可以组成和为 j 的子集
vector<vector<int>> memo(n, vector<int>(target + 1, -1)); // -1 表示未计算, 0 表示 false, 1 表
示 true

// 定义 DFS 函数
function<bool(int, int)> dfs = [&](int index, int remaining) -> bool {
 // 基础情况: 如果剩余和为 0, 说明找到了一个子集
 if (remaining == 0) {
 return true;
 }

 // 基础情况: 如果已经考虑完所有石头或者剩余和小于 0, 返回 false
 if (index == n || remaining < 0) {
 return false;
 }

 // 检查缓存
 if (memo[index][remaining] != -1) {
 return memo[index][remaining] == 1;
 }

 // 尝试两种选择: 选或不选当前石头
 // 1. 选当前石头: 剩余和减去当前石头的重量, 继续考虑下一个石头
 bool choose = dfs(index + 1, remaining - stones[index]);

 // 2. 不选当前石头: 剩余和不变, 继续考虑下一个石头
 bool notChoose = dfs(index + 1, remaining);

 // 缓存结果
 memo[index][remaining] = (choose || notChoose) ? 1 : 0;
 return memo[index][remaining] == 1;
};

```

```

// 从最大的可能和开始，找到最大的可达到的和
for (int i = target; i >= 0; i--) {
 if (dfs(0, i)) {
 return sum - 2 * i;
 }
}

return 0;
}

/***
 * 位运算优化的 DP 实现
 * 每个二进制位表示是否可以组成对应索引的和
 */
int lastStoneWeightIIBit(vector<int>& stones) {
 if (stones.empty()) {
 return 0;
 }

 int sum = 0;
 for (int stone : stones) {
 sum += stone;
 }

 int target = sum / 2;

 // 使用位集，每个位表示是否可以组成对应的和
 // bits 的第 i 位为 1 表示可以组成和为 i 的子集
 long long bits = 1; // 0b000...0001，表示和为 0 是可以的

 for (int stone : stones) {
 // 位运算：将当前 bits 左移 stone 位，并与原 bits 进行或操作
 bits |= bits << stone;

 // 限制 bits 的范围，避免不必要的计算
 if (bits > (1LL << (target + 1)) - 1) {
 bits &= (1LL << (target + 1)) - 1;
 }
 }

 // 找到最大的 i，使得 bits 的第 i 位为 1，其中 i <= target
 for (int i = target; i >= 0; i--) {

```

```

 if ((bits & (1LL << i)) != 0) {
 return sum - 2 * i;
 }
}

return 0;
}

/***
 * 另一种方法：直接计算可能的重量差异
 * 使用集合来记录所有可能的重量和
 */
int lastStoneWeightIISet(vector<int>& stones) {
 if (stones.empty()) {
 return 0;
 }

 // 使用布尔数组来记录所有可能的重量和
 int sum = 0;
 for (int stone : stones) {
 sum += stone;
 }

 int target = sum / 2;
 vector<bool> dp(target + 1, false);
 dp[0] = true;

 for (int stone : stones) {
 for (int i = target; i >= stone; i--) {
 dp[i] = dp[i] || dp[i - stone];
 }
 }

 // 找到最大的可能和
 for (int i = target; i >= 0; i--) {
 if (dp[i]) {
 return sum - 2 * i;
 }
 }

 return 0;
}

```

```

int main() {
 // 测试用例 1
 vector<int> stones1 = {2, 7, 4, 1, 8, 1};
 cout << "测试用例 1 结果: " << lastStoneWeightII(stones1) << endl; // 预期输出: 1

 // 测试用例 2
 vector<int> stones2 = {31, 26, 33, 21, 40};
 cout << "测试用例 2 结果: " << lastStoneWeightII(stones2) << endl; // 预期输出: 5

 // 测试用例 3
 vector<int> stones3 = {1, 2};
 cout << "测试用例 3 结果: " << lastStoneWeightII(stones3) << endl; // 预期输出: 1

 return 0;
}

```

---

文件: Code29\_LastStoneWeightII.java

---

```

package class073;

// LeetCode 1049. 最后一块石头的重量 II
// 题目描述: 有一堆石头, 每块石头的重量都是正整数。每一回合, 从中选出任意两块石头, 然后将它们一起粉碎。
// 假设石头的重量分别为 x 和 y, 且 x <= y。那么粉碎的可能结果如下:
// - 如果 x == y, 那么两块石头都会被完全粉碎;
// - 如果 x != y, 那么重量为 x 的石头会被完全粉碎, 而重量为 y 的石头新重量为 y - x。
// 最后, 最多只会剩下一块石头。返回此石头的最小可能重量。如果没有石头剩下, 就返回 0。
// 链接: https://leetcode.cn/problems/last-stone-weight-ii/
//
// 解题思路:
// 这是一个 0-1 背包问题的变种, 问题可以转化为:
// 将石头分成两组, 使得两组的重量差最小。那么剩下的石头重量就是两组重量的差的最小值。
// 等价于: 找到一组石头, 使得它们的重量尽可能接近总重量的一半。
//
// 状态定义: dp[i] 表示是否可以组成和为 i 的子集
// 状态转移方程: dp[i] = dp[i] || dp[i - stone], 其中 stone 是当前石头的重量, 且 i >= stone
// 初始状态: dp[0] = true, 表示和为 0 的子集存在 (空集)
//
// 时间复杂度: O(n * target), 其中 n 是石头的数量, target 是总重量的一半
// 空间复杂度: O(target), 使用一维 DP 数组

```

```
public class Code29_LastStoneWeightII {

 // 主方法，用于测试
 public static void main(String[] args) {
 // 测试用例 1
 int[] stones1 = {2, 7, 4, 1, 8, 1};
 System.out.println("测试用例 1 结果: " + lastStoneWeightII(stones1)); // 预期输出: 1

 // 测试用例 2
 int[] stones2 = {31, 26, 33, 21, 40};
 System.out.println("测试用例 2 结果: " + lastStoneWeightII(stones2)); // 预期输出: 5

 // 测试用例 3
 int[] stones3 = {1, 2};
 System.out.println("测试用例 3 结果: " + lastStoneWeightII(stones3)); // 预期输出: 1
 }

 /**
 * 计算最后一块石头的最小可能重量
 * @param stones 石头重量数组
 * @return 最后一块石头的最小可能重量
 */
 public static int lastStoneWeightII(int[] stones) {
 // 参数验证
 if (stones == null || stones.length == 0) {
 return 0;
 }

 // 计算石头总重量
 int sum = 0;
 for (int stone : stones) {
 sum += stone;
 }

 // 计算目标和：总重量的一半（向下取整）
 int target = sum / 2;

 // 创建一维 DP 数组，dp[i] 表示是否可以组成和为 i 的子集
 boolean[] dp = new boolean[target + 1];

 // 初始状态：和为 0 的子集存在（空集）
 dp[0] = true;
```

```

// 对于每个石头，逆序遍历目标和（0-1 背包问题）
for (int stone : stones) {
 for (int i = target; i >= stone; i--) {
 // 状态转移：如果 dp[i - stone] 为 true，说明可以组成和为 i - stone 的子集，
 // 那么再加上当前石头 stone，就可以组成和为 i 的子集
 dp[i] = dp[i] || dp[i - stone];
 }
}

// 找到最大的 i，使得 dp[i] 为 true，其中 i <= target
// 剩下的石头重量就是 sum - 2 * i
for (int i = target; i >= 0; i--) {
 if (dp[i]) {
 return sum - 2 * i;
 }
}

return 0; // 理论上不会到达这里
}

/***
 * 二维 DP 数组实现
 * dp[i][j] 表示前 i 个石头中是否可以选择一些石头，使得它们的和为 j
 */
public static int lastStoneWeightII2D(int[] stones) {
 if (stones == null || stones.length == 0) {
 return 0;
 }

 int sum = 0;
 for (int stone : stones) {
 sum += stone;
 }

 int target = sum / 2;
 int n = stones.length;

 // 创建二维 DP 数组
 boolean[][] dp = new boolean[n + 1][target + 1];

 // 初始化：前 0 个石头可以组成和为 0 的子集
 for (int i = 0; i <= n; i++) {
 dp[i][0] = true;
 }
}

```

```

}

// 填充 DP 数组
for (int i = 1; i <= n; i++) {
 int stone = stones[i - 1];
 for (int j = 1; j <= target; j++) {
 // 不选当前石头
 dp[i][j] = dp[i - 1][j];
 // 选当前石头（如果可以的话）
 if (j >= stone) {
 dp[i][j] = dp[i][j] || dp[i - 1][j - stone];
 }
 }
}

// 找到最大的 j, 使得 dp[n][j] 为 true, 其中 j <= target
for (int j = target; j >= 0; j--) {
 if (dp[n][j]) {
 return sum - 2 * j;
 }
}

return 0;
}

/***
 * 优化版本：提前剪枝
 */
public static int lastStoneWeightIIOptimized(int[] stones) {
 if (stones == null || stones.length == 0) {
 return 0;
 }

 // 计算石头总重量
 int sum = 0;
 for (int stone : stones) {
 sum += stone;
 }

 // 计算目标和：总重量的一半（向下取整）
 int target = sum / 2;

 // 创建一维 DP 数组, dp[i] 表示是否可以组成和为 i 的子集

```

```

boolean[] dp = new boolean[target + 1];
dp[0] = true;

// 使用一个变量记录当前可以达到的最大和
int currentMax = 0;

for (int stone : stones) {
 // 逆序遍历目标和
 for (int i = target; i >= stone; i--) {
 if (dp[i - stone] && !dp[i]) {
 dp[i] = true;
 currentMax = Math.max(currentMax, i);
 // 提前剪枝：如果已经可以达到目标和，直接返回结果
 if (currentMax == target) {
 return sum - 2 * target;
 }
 }
 }
}

// 返回最小可能的最后一块石头重量
return sum - 2 * currentMax;
}

/**
 * 递归+记忆化搜索实现
 */
public static int lastStoneWeightIIDFS(int[] stones) {
 if (stones == null || stones.length == 0) {
 return 0;
 }

 int sum = 0;
 for (int stone : stones) {
 sum += stone;
 }

 int target = sum / 2;
 int n = stones.length;

 // 使用二维数组作为缓存，memo[i][j]表示从第 i 个石头开始，是否可以组成和为 j 的子集
 Boolean[][] memo = new Boolean[n][target + 1];

```

```

// 从最大的可能和开始，找到最大的可达到的和
for (int i = target; i >= 0; i--) {
 if (dfs(stones, 0, i, memo)) {
 return sum - 2 * i;
 }
}

return 0;
}

/**
 * 递归辅助函数
 * @param stones 石头重量数组
 * @param index 当前考虑的石头索引
 * @param remaining 需要达到的和
 * @param memo 缓存数组
 * @return 是否可以从当前索引开始，组成和为 remaining 的子集
 */
private static boolean dfs(int[] stones, int index, int remaining, Boolean[][] memo) {
 // 基础情况：如果剩余和为 0，说明找到了一个子集
 if (remaining == 0) {
 return true;
 }

 // 基础情况：如果已经考虑完所有石头或者剩余和小于 0，返回 false
 if (index == stones.length || remaining < 0) {
 return false;
 }

 // 检查缓存
 if (memo[index][remaining] != null) {
 return memo[index][remaining];
 }

 // 尝试两种选择：选或不选当前石头
 // 1. 选当前石头：剩余和减去当前石头的重量，继续考虑下一个石头
 boolean choose = dfs(stones, index + 1, remaining - stones[index], memo);

 // 2. 不选当前石头：剩余和不变，继续考虑下一个石头
 boolean notChoose = dfs(stones, index + 1, remaining, memo);

 // 缓存结果
 memo[index][remaining] = choose || notChoose;
}

```

```

 return memo[index][remaining];
 }

/***
 * 位运算优化的 DP 实现
 * 每个二进制位表示是否可以组成对应索引的和
 */
public static int lastStoneWeightIIBit(int[] stones) {
 if (stones == null || stones.length == 0) {
 return 0;
 }

 int sum = 0;
 for (int stone : stones) {
 sum += stone;
 }

 int target = sum / 2;

 // 使用位集，每个位表示是否可以组成对应的和
 // bits 的第 i 位为 1 表示可以组成和为 i 的子集
 int bits = 1; // 0b000...0001, 表示和为 0 是可以的

 for (int stone : stones) {
 // 位运算：将当前 bits 左移 stone 位，并与原 bits 进行或操作
 bits |= bits << stone;

 // 限制 bits 的范围，避免不必要的计算
 if (bits > (1 << (target + 1)) - 1) {
 bits &= (1 << (target + 1)) - 1;
 }
 }

 // 找到最大的 i，使得 bits 的第 i 位为 1，其中 i <= target
 for (int i = target; i >= 0; i--) {
 if ((bits & (1 << i)) != 0) {
 return sum - 2 * i;
 }
 }

 return 0;
}

```

```

/**
 * 另一种方法：直接计算可能的重量差异
 * 使用集合来记录所有可能的重量和
 */
public static int lastStoneWeightIISet(int[] stones) {
 if (stones == null || stones.length == 0) {
 return 0;
 }

 // 使用布尔数组来记录所有可能的重量和
 int sum = 0;
 for (int stone : stones) {
 sum += stone;
 }

 int target = sum / 2;
 boolean[] dp = new boolean[target + 1];
 dp[0] = true;

 for (int stone : stones) {
 for (int i = target; i >= stone; i--) {
 dp[i] = dp[i] || dp[i - stone];
 }
 }
}

// 找到最大的可能和
for (int i = target; i >= 0; i--) {
 if (dp[i]) {
 return sum - 2 * i;
 }
}

return 0;
}
}

```

文件: Code29\_LastStoneWeightII.py

```

LeetCode 1049. 最后一块石头的重量 II
题目描述: 有一堆石头，每块石头的重量都是正整数。每一回合，从中选出任意两块石头，然后将它们一起粉碎。

```

```
假设石头的重量分别为 x 和 y，且 x <= y。那么粉碎的可能结果如下：
- 如果 x == y，那么两块石头都会被完全粉碎；
- 如果 x != y，那么重量为 x 的石头会被完全粉碎，而重量为 y 的石头新重量为 y - x。
最后，最多只会剩下一块石头。返回此石头的最小可能重量。如果没有石头剩下，就返回 0。
链接: https://leetcode.cn/problems/last-stone-weight-ii/

解题思路：
这是一个 0-1 背包问题的变种，问题可以转化为：
将石头分成两组，使得两组的重量差最小。那么剩下的石头重量就是两组重量的差的最小值。
等价于：找到一组石头，使得它们的重量尽可能接近总重量的一半。

状态定义：dp[i] 表示是否可以组成和为 i 的子集
状态转移方程：dp[i] = dp[i] || dp[i - stone]，其中 stone 是当前石头的重量，且 i >= stone
初始状态：dp[0] = True，表示和为 0 的子集存在（空集）

时间复杂度：O(n * target)，其中 n 是石头的数量，target 是总重量的一半
空间复杂度：O(target)，使用一维 DP 数组
```

```
from typing import List
```

```
def last_stone_weight_ii(stones: List[int]) -> int:
```

```
 """
```

```
 计算最后一块石头的最小可能重量
```

```
Args:
```

```
 stones: 石头重量数组
```

```
Returns:
```

```
 int: 最后一块石头的最小可能重量
```

```
 """
```

```
参数验证
```

```
if not stones:
 return 0
```

```
计算石头总重量
```

```
total_sum = sum(stones)
```

```
计算目标和：总重量的一半（向下取整）
```

```
target = total_sum // 2
```

```
创建一维 DP 数组，dp[i] 表示是否可以组成和为 i 的子集
```

```
dp = [False] * (target + 1)
```

```

初始状态: 和为 0 的子集存在 (空集)
dp[0] = True

对于每个石头, 逆序遍历目标和 (0-1 背包问题)
for stone in stones:
 for i in range(target, stone - 1):
 # 状态转移: 如果 dp[i - stone] 为 True, 说明可以组成和为 i - stone 的子集,
 # 那么再加上当前石头 stone, 就可以组成和为 i 的子集
 dp[i] = dp[i] or dp[i - stone]

找到最大的 i, 使得 dp[i] 为 True, 其中 i <= target
剩下的石头重量就是 total_sum - 2 * i
for i in range(target, -1, -1):
 if dp[i]:
 return total_sum - 2 * i

return 0 # 理论上不会到达这里

```

```

def last_stone_weight_ii_2d(stones: List[int]) -> int:
 """
 二维 DP 数组实现
 dp[i][j] 表示前 i 个石头中是否可以选择一些石头, 使得它们的和为 j
 """

 if not stones:
 return 0

 total_sum = sum(stones)
 target = total_sum // 2
 n = len(stones)

 # 创建二维 DP 数组
 dp = [[False] * (target + 1) for _ in range(n + 1)]

 # 初始化: 前 0 个石头可以组成和为 0 的子集
 for i in range(n + 1):
 dp[i][0] = True

 # 填充 DP 数组
 for i in range(1, n + 1):
 stone = stones[i - 1]
 for j in range(1, target + 1):
 dp[i][j] = dp[i - 1][j] or dp[i - 1][j - stone]

```

```

不选当前石头
dp[i][j] = dp[i - 1][j]
选当前石头（如果可以的话）
if j >= stone:
 dp[i][j] = dp[i][j] or dp[i - 1][j - stone]

找到最大的 j, 使得 dp[n][j] 为 True, 其中 j <= target
for j in range(target, -1, -1):
 if dp[n][j]:
 return total_sum - 2 * j

return 0

def last_stone_weight_ii_optimized(stones: List[int]) -> int:
"""
优化版本：提前剪枝
"""

if not stones:
 return 0

计算石头总重量
total_sum = sum(stones)

计算目标和：总重量的一半（向下取整）
target = total_sum // 2

创建一维 DP 数组，dp[i] 表示是否可以组成和为 i 的子集
dp = [False] * (target + 1)
dp[0] = True

使用一个变量记录当前可以达到的最大和
current_max = 0

for stone in stones:
 # 逆序遍历目标和
 for i in range(target, stone - 1, -1):
 if dp[i - stone] and not dp[i]:
 dp[i] = True
 current_max = max(current_max, i)
 # 提前剪枝：如果已经可以达到目标和，直接返回结果
 if current_max == target:
 return total_sum - 2 * target

```

```

返回最小可能的最后一块石头重量
return total_sum - 2 * current_max

def last_stone_weight_ii_dfs(stones: List[int]) -> int:
 """
 递归+记忆化搜索实现
 """

 if not stones:
 return 0

 total_sum = sum(stones)
 target = total_sum // 2
 n = len(stones)

 # 使用二维列表作为缓存, memo[i][j]表示从第 i 个石头开始, 是否可以组成和为 j 的子集
 memo = [[-1 for _ in range(target + 1)] for _ in range(n)] # -1 表示未计算, 0 表示 False, 1 表示 True

 def dfs(index: int, remaining: int) -> bool:
 """
 递归辅助函数
 # 基础情况: 如果剩余和为 0, 说明找到了一个子集
 if remaining == 0:
 return True

 # 基础情况: 如果已经考虑完所有石头或者剩余和小于 0, 返回 False
 if index == n or remaining < 0:
 return False

 # 检查缓存
 if memo[index][remaining] != -1:
 return memo[index][remaining] == 1

 # 尝试两种选择: 选或不选当前石头
 # 1. 选当前石头: 剩余和减去当前石头的重量, 继续考虑下一个石头
 choose = dfs(index + 1, remaining - stones[index])

 # 2. 不选当前石头: 剩余和不变, 继续考虑下一个石头
 not_choose = dfs(index + 1, remaining)

 # 缓存结果
 memo[index][remaining] = 1 if (choose or not_choose) else 0
 return choose or not_choose

 return dfs(0, target)

```

```

 return memo[index][remaining] == 1

从最大的可能和开始，找到最大的可达到的和
for i in range(target, -1, -1):
 if dfs(0, i):
 return total_sum - 2 * i

return 0

def last_stone_weight_ii_bit(stones: List[int]) -> int:
 """
 位运算优化的 DP 实现
 每个二进制位表示是否可以组成对应索引的和
 """
 if not stones:
 return 0

 total_sum = sum(stones)
 target = total_sum // 2

 # 使用位集，每个位表示是否可以组成对应的和
 # bits 的第 i 位为 1 表示可以组成和为 i 的子集
 bits = 1 # 0b000...0001，表示和为 0 是可以的

 for stone in stones:
 # 位运算：将当前 bits 左移 stone 位，并与原 bits 进行或操作
 bits |= bits << stone

 # 限制 bits 的范围，避免不必要的计算
 if bits > (1 << (target + 1)) - 1:
 bits &= (1 << (target + 1)) - 1

 # 找到最大的 i，使得 bits 的第 i 位为 1，其中 i <= target
 for i in range(target, -1, -1):
 if (bits & (1 << i)) != 0:
 return total_sum - 2 * i

 return 0

def last_stone_weight_ii_set(stones: List[int]) -> int:
 """

```

另一种方法：直接计算可能的重量差异  
使用集合来记录所有可能的重量和

```
"""
if not stones:
 return 0

使用布尔数组来记录所有可能的重量和
total_sum = sum(stones)
target = total_sum // 2
dp = [False] * (target + 1)
dp[0] = True

for stone in stones:
 for i in range(target, stone - 1, -1):
 dp[i] = dp[i] or dp[i - stone]

找到最大的可能和
for i in range(target, -1, -1):
 if dp[i]:
 return total_sum - 2 * i

return 0
```

```
测试代码
if __name__ == "__main__":
 # 测试用例 1
 stones1 = [2, 7, 4, 1, 8, 1]
 print(f"测试用例 1 结果: {last_stone_weight_ii(stones1)}") # 预期输出: 1

 # 测试用例 2
 stones2 = [31, 26, 33, 21, 40]
 print(f"测试用例 2 结果: {last_stone_weight_ii(stones2)}") # 预期输出: 5

 # 测试用例 3
 stones3 = [1, 2]
 print(f"测试用例 3 结果: {last_stone_weight_ii(stones3)}") # 预期输出: 1
```

=====

文件: Code30\_OnesAndZeroes.cpp

=====

```
#include <iostream>
```

```

#include <vector>
#include <string>
#include <functional>

// LeetCode 474. 一和零
// 题目描述：给你一个二进制字符串数组 strs 和两个整数 m 和 n 。请你找出并返回 strs 的最大子集的长度，
// 该子集中最多有 m 个 0 和 n 个 1 。如果所有字符串都不满足条件，返回 0 。
// 链接: https://leetcode.cn/problems/ones-and-zeroes/
//
// 解题思路：
// 这是一个二维费用的 0-1 背包问题：
// - 每个字符串可以看作是一个物品
// - 物品的重量有两个维度：0 的个数和 1 的个数
// - 背包的容量也有两个维度：m（最多 m 个 0）和 n（最多 n 个 1）
// - 我们需要找出最多能放多少个物品，使得两个维度的重量都不超过背包容量
//
// 状态定义：dp[i][j] 表示最多使用 i 个 0 和 j 个 1 时，可以组成的大子集的长度
// 状态转移方程：对于每个字符串，计算其包含的 0 的个数 zeros 和 1 的个数 ones
// dp[i][j] = max(dp[i][j], dp[i-zeros][j-ones] + 1) (当 i >= zeros 且 j >= ones 时)
// 初始状态：dp[0][0] = 0，表示不使用任何字符时，子集长度为 0
//
// 时间复杂度：O(l * m * n)，其中 l 是字符串数组的长度，m 和 n 是给定的两个整数
// 空间复杂度：O(m * n)，使用二维 DP 数组

```

```
using namespace std;
```

```
/**
 * 找出 strs 的最大子集长度，该子集中最多有 m 个 0 和 n 个 1
 * @param strs 二进制字符串数组
 * @param m 最多允许的 0 的个数
 * @param n 最多允许的 1 的个数
 * @return 满足条件的最大子集长度
 */
```

```
int findMaxForm(vector<string>& strs, int m, int n) {
 // 参数验证
 if (strs.empty()) {
 return 0;
 }
```

```
// 创建二维 DP 数组，dp[i][j] 表示最多使用 i 个 0 和 j 个 1 时，可以组成的大子集的长度
vector<vector<int>> dp(m + 1, vector<int>(n + 1, 0));
```

```

// 对于每个字符串，计算其包含的 0 的个数和 1 的个数
for (const string& str : strs) {
 int zeros = 0, ones = 0;
 for (char c : str) {
 if (c == '0') {
 zeros++;
 } else {
 ones++;
 }
 }
}

// 二维 0-1 背包问题，需要逆序遍历两个维度
for (int i = m; i >= zeros; i--) {
 for (int j = n; j >= ones; j--) {
 // 状态转移：选择当前字符串或不选择当前字符串，取最大值
 dp[i][j] = max(dp[i][j], dp[i - zeros][j - ones] + 1);
 }
}
}

// 返回最多使用 m 个 0 和 n 个 1 时，可以组成的大子集的长度
return dp[m][n];
}

/***
 * 优化版本：预处理每个字符串的 0 和 1 的个数
 */
int findMaxFormOptimized(vector<string>& strs, int m, int n) {
 if (strs.empty()) {
 return 0;
 }

 int l = strs.size();
 // 预处理每个字符串的 0 和 1 的个数
 vector<vector<int>> counts(l, vector<int>(2, 0)); // counts[i][0] 表示第 i 个字符串中 0 的个数,
 counts[i][1] 表示 1 的个数

 for (int i = 0; i < l; i++) {
 int zeros = 0, ones = 0;
 for (char c : strs[i]) {
 if (c == '0') {
 zeros++;
 } else {

```

```

 ones++;
 }
}

counts[i][0] = zeros;
counts[i][1] = ones;
}

vector<vector<int>> dp(m + 1, vector<int>(n + 1, 0));

// 遍历每个字符串
for (const auto& count : counts) {
 int zeros = count[0];
 int ones = count[1];

 // 逆序遍历两个维度
 for (int i = m; i >= zeros; i--) {
 for (int j = n; j >= ones; j--) {
 dp[i][j] = max(dp[i][j], dp[i - zeros][j - ones] + 1);
 }
 }
}

return dp[m][n];
}

/***
 * 三维 DP 数组实现（更直观但空间复杂度更高）
 * dp[k][i][j] 表示考虑前 k 个字符串，最多使用 i 个 0 和 j 个 1 时，可以组成
 * 的最大子集的长度
 */
int findMaxForm3D(vector<string>& strs, int m, int n) {
 if (strs.empty()) {
 return 0;
 }

 int l = strs.size();
 // 预处理每个字符串的 0 和 1 的个数
 vector<vector<int>> counts(l, vector<int>(2, 0));

 for (int i = 0; i < l; i++) {
 int zeros = 0, ones = 0;
 for (char c : strs[i]) {
 if (c == '0') {
 zeros++;
 }

```

```

 } else {
 ones++;
 }
 }

counts[i][0] = zeros;
counts[i][1] = ones;
}

// 创建三维 DP 数组
vector<vector<vector<int>>> dp(l + 1, vector<vector<int>>(m + 1, vector<int>(n + 1, 0)));

// 填充 DP 数组
for (int k = 1; k <= l; k++) {
 int zeros = counts[k - 1][0];
 int ones = counts[k - 1][1];

 for (int i = 0; i <= m; i++) {
 for (int j = 0; j <= n; j++) {
 // 不选择第 k 个字符串
 dp[k][i][j] = dp[k - 1][i][j];

 // 选择第 k 个字符串 (如果可以的话)
 if (i >= zeros && j >= ones) {
 dp[k][i][j] = max(dp[k][i][j], dp[k - 1][i - zeros][j - ones] + 1);
 }
 }
 }
}

return dp[l][m][n];
}

/***
 * 提前剪枝优化
 */
int findMaxFormPruned(vector<string>& strs, int m, int n) {
 if (strs.empty()) {
 return 0;
 }

 int l = strs.size();
 // 预处理每个字符串的 0 和 1 的个数，并过滤掉不可能被选中的字符串
 vector<vector<int>> counts(l, vector<int>(2, 0));

```

```

int validCount = 0;

for (int i = 0; i < 1; i++) {
 int zeros = 0, ones = 0;
 for (char c : strs[i]) {
 if (c == '0') {
 zeros++;
 } else {
 ones++;
 }
 }
}

// 剪枝：如果字符串的 0 或 1 的个数超过给定的限制，则无法选择该字符串
if (zeros <= m && ones <= n) {
 counts[validCount][0] = zeros;
 counts[validCount][1] = ones;
 validCount++;
}
}

vector<vector<int>> dp(m + 1, vector<int>(n + 1, 0));

// 遍历有效的字符串
for (int i = 0; i < validCount; i++) {
 int zeros = counts[i][0];
 int ones = counts[i][1];

 // 逆序遍历两个维度
 for (int j = m; j >= zeros; j--) {
 for (int k = n; k >= ones; k--) {
 dp[j][k] = max(dp[j][k], dp[j - zeros][k - ones] + 1);
 }
 }
}

return dp[m][n];
}

/***
 * 使用滚动数组优化空间复杂度
 * 注意：在这个问题中，滚动数组的优化已经在基本实现中完成（使用二维数组）
 * 这个方法只是为了展示如何进一步优化（尽管在这个问题中意义不大）
 */

```

```

int findMaxFormScrolling(vector<string>& strs, int m, int n) {
 if (strs.empty()) {
 return 0;
 }

 // 创建二维 DP 数组
 vector<vector<int>> dp(m + 1, vector<int>(n + 1, 0));

 for (const string& str : strs) {
 int zeros = 0, ones = 0;
 for (char c : str) {
 if (c == '0') {
 zeros++;
 } else {
 ones++;
 }
 }
 }

 // 逆序遍历两个维度（这已经是滚动数组的思想）
 for (int i = m; i >= zeros; i--) {
 for (int j = n; j >= ones; j--) {
 dp[i][j] = max(dp[i][j], dp[i - zeros][j - ones] + 1);
 }
 }
}

return dp[m][n];
}

```

```

/***
 * 递归+记忆化搜索实现
 * 注意：由于这个问题的参数范围较大，递归+记忆化搜索可能会超时
 * 这里仅作为一种实现方式展示
 */

```

```

int findMaxFormDFS(vector<string>& strs, int m, int n) {
 if (strs.empty()) {
 return 0;
 }

 int l = strs.size();
 // 预处理每个字符串的 0 和 1 的个数
 vector<vector<int>> counts(l, vector<int>(2, 0));

```

```

for (int i = 0; i < l; i++) {
 int zeros = 0, ones = 0;
 for (char c : strs[i]) {
 if (c == '0') {
 zeros++;
 } else {
 ones++;
 }
 }
 counts[i][0] = zeros;
 counts[i][1] = ones;
}

```

// 创建三维缓存数组，memo[k][i][j]表示考虑前 k 个字符串，最多使用 i 个 0 和 j 个 1 时，可以组成的最大子集的长度

```
vector<vector<vector<int>>> memo(l, vector<vector<int>>(m + 1, vector<int>(n + 1, -1)));
```

// 定义 DFS 函数

```
function<int(int, int, int)> dfs = [&](int index, int m0, int n1) -> int {
 // 基础情况：如果已经考虑完所有字符串，返回 0
 if (index < 0) {
 return 0;
 }
}
```

// 检查缓存

```
if (memo[index][m0][n1] != -1) {
 return memo[index][m0][n1];
}
```

// 不选择当前字符串

```
int notChoose = dfs(index - 1, m0, n1);
```

// 选择当前字符串（如果可以的话）

```
int choose = 0;
int zeros = counts[index][0];
int ones = counts[index][1];

if (zeros <= m0 && ones <= n1) {
 choose = 1 + dfs(index - 1, m0 - zeros, n1 - ones);
}
```

// 缓存结果

```
memo[index][m0][n1] = max(notChoose, choose);
```

```

 return memo[index][m0][n1];
 };

 // 调用递归函数
 return dfs(l - 1, m, n);
}

int main() {
 // 测试用例 1
 vector<string> strs1 = {"10", "0001", "111001", "1", "0"};
 int m1 = 5, n1 = 3;
 cout << "测试用例 1 结果: " << findMaxForm(strs1, m1, n1) << endl; // 预期输出: 4

 // 测试用例 2
 vector<string> strs2 = {"10", "0", "1"};
 int m2 = 1, n2 = 1;
 cout << "测试用例 2 结果: " << findMaxForm(strs2, m2, n2) << endl; // 预期输出: 2

 // 测试用例 3
 vector<string> strs3 = {"10", "0001", "111001", "1", "0"};
 int m3 = 4, n3 = 3;
 cout << "测试用例 3 结果: " << findMaxForm(strs3, m3, n3) << endl; // 预期输出: 3

 return 0;
}

```

=====

文件: Code30\_OnesAndZeroes.java

=====

```

package class073;

// LeetCode 474. 一和零
// 题目描述: 给你一个二进制字符串数组 strs 和两个整数 m 和 n 。请你找出并返回 strs 的最大子集的长度,
// 该子集中最多有 m 个 0 和 n 个 1 。如果所有字符串都不满足条件, 返回 0 。
// 链接: https://leetcode.cn/problems/ones-and-zeroes/
//
// 解题思路:
// 这是一个二维费用的 0-1 背包问题:
// - 每个字符串可以看作是一个物品
// - 物品的重量有两个维度: 0 的个数和 1 的个数
// - 背包的容量也有两个维度: m (最多 m 个 0) 和 n (最多 n 个 1)

```

```

// - 我们需要找出最多能放多少个物品，使得两个维度的重量都不超过背包容量
//
// 状态定义：dp[i][j] 表示最多使用 i 个 0 和 j 个 1 时，可以组成的大子集的长度
// 状态转移方程：对于每个字符串，计算其包含的 0 的个数 zeros 和 1 的个数 ones
// dp[i][j] = max(dp[i][j], dp[i-zeros][j-ones] + 1) (当 i >= zeros 且 j >= ones 时)
// 初始状态：dp[0][0] = 0，表示不使用任何字符时，子集长度为 0
//
// 时间复杂度：O(1 * m * n)，其中 1 是字符串数组的长度，m 和 n 是给定的两个整数
// 空间复杂度：O(m * n)，使用二维 DP 数组

public class Code30_OnesAndZeroes {

 // 主方法，用于测试
 public static void main(String[] args) {
 // 测试用例 1
 String[] strs1 = {"10", "0001", "111001", "1", "0"};
 int m1 = 5, n1 = 3;
 System.out.println("测试用例 1 结果：" + findMaxForm(strs1, m1, n1)); // 预期输出：4

 // 测试用例 2
 String[] strs2 = {"10", "0", "1"};
 int m2 = 1, n2 = 1;
 System.out.println("测试用例 2 结果：" + findMaxForm(strs2, m2, n2)); // 预期输出：2

 // 测试用例 3
 String[] strs3 = {"10", "0001", "111001", "1", "0"};
 int m3 = 4, n3 = 3;
 System.out.println("测试用例 3 结果：" + findMaxForm(strs3, m3, n3)); // 预期输出：3
 }

 /**
 * 找出 strs 的最大子集长度，该子集中最多有 m 个 0 和 n 个 1
 * @param strs 二进制字符串数组
 * @param m 最多允许的 0 的个数
 * @param n 最多允许的 1 的个数
 * @return 满足条件的最大子集长度
 */
 public static int findMaxForm(String[] strs, int m, int n) {
 // 参数验证
 if (strs == null || strs.length == 0) {
 return 0;
 }

```

```

// 创建二维 DP 数组，dp[i][j]表示最多使用 i 个 0 和 j 个 1 时，可以组成的大子集的长度
int[][] dp = new int[m + 1][n + 1];

// 对于每个字符串，计算其包含的 0 的个数和 1 的个数
for (String str : strs) {
 int zeros = 0, ones = 0;
 for (char c : str.toCharArray()) {
 if (c == '0') {
 zeros++;
 } else {
 ones++;
 }
 }
}

// 二维 0-1 背包问题，需要逆序遍历两个维度
for (int i = m; i >= zeros; i--) {
 for (int j = n; j >= ones; j--) {
 // 状态转移：选择当前字符串或不选择当前字符串，取最大值
 dp[i][j] = Math.max(dp[i][j], dp[i - zeros][j - ones] + 1);
 }
}
}

// 返回最多使用 m 个 0 和 n 个 1 时，可以组成的大子集的长度
return dp[m][n];
}

/***
 * 优化版本：预处理每个字符串的 0 和 1 的个数
 */
public static int findMaxFormOptimized(String[] strs, int m, int n) {
 if (strs == null || strs.length == 0) {
 return 0;
 }

 int l = strs.length;
 // 预处理每个字符串的 0 和 1 的个数
 int[][] counts = new int[l][2]; // counts[i][0] 表示第 i 个字符串中 0 的个数，counts[i][1] 表示 1 的个数

 for (int i = 0; i < l; i++) {
 int zeros = 0, ones = 0;
 for (char c : strs[i].toCharArray()) {

```

```

 if (c == '0') {
 zeros++;
 } else {
 ones++;
 }
 }

 counts[i][0] = zeros;
 counts[i][1] = ones;
}

int[][] dp = new int[m + 1][n + 1];

// 遍历每个字符串
for (int[] count : counts) {
 int zeros = count[0];
 int ones = count[1];

 // 逆序遍历两个维度
 for (int i = m; i >= zeros; i--) {
 for (int j = n; j >= ones; j--) {
 dp[i][j] = Math.max(dp[i][j], dp[i - zeros][j - ones] + 1);
 }
 }
}

return dp[m][n];
}

/***
 * 三维 DP 数组实现（更直观但空间复杂度更高）
 * dp[k][i][j] 表示考虑前 k 个字符串，最多使用 i 个 0 和 j 个 1 时，可以组成
 * 的最大子集的长度
 */
public static int findMaxForm3D(String[] strs, int m, int n) {
 if (strs == null || strs.length == 0) {
 return 0;
 }

 int l = strs.length;
 // 预处理每个字符串的 0 和 1 的个数
 int[][] counts = new int[l][2];

 for (int i = 0; i < l; i++) {
 int zeros = 0, ones = 0;

```

```

 for (char c : strs[i].toCharArray()) {
 if (c == '0') {
 zeros++;
 } else {
 ones++;
 }
 }
 counts[i][0] = zeros;
 counts[i][1] = ones;
 }

 // 创建三维 DP 数组
 int[][][] dp = new int[1 + 1][m + 1][n + 1];

 // 填充 DP 数组
 for (int k = 1; k <= 1; k++) {
 int zeros = counts[k - 1][0];
 int ones = counts[k - 1][1];

 for (int i = 0; i <= m; i++) {
 for (int j = 0; j <= n; j++) {
 // 不选择第 k 个字符串
 dp[k][i][j] = dp[k - 1][i][j];

 // 选择第 k 个字符串 (如果可以的话)
 if (i >= zeros && j >= ones) {
 dp[k][i][j] = Math.max(dp[k][i][j], dp[k - 1][i - zeros][j - ones] + 1);
 }
 }
 }
 }

 return dp[1][m][n];
}

/**
 * 提前剪枝优化
 */
public static int findMaxFormPruned(String[] strs, int m, int n) {
 if (strs == null || strs.length == 0) {
 return 0;
 }
}

```

```

int l = strs.length;
// 预处理每个字符串的 0 和 1 的个数，并过滤掉不可能被选中的字符串
int[][] counts = new int[l][2];
int validCount = 0;

for (int i = 0; i < l; i++) {
 int zeros = 0, ones = 0;
 for (char c : strs[i].toCharArray()) {
 if (c == '0') {
 zeros++;
 } else {
 ones++;
 }
 }
}

// 剪枝：如果字符串的 0 或 1 的个数超过给定的限制，则无法选择该字符串
if (zeros <= m && ones <= n) {
 counts[validCount][0] = zeros;
 counts[validCount][1] = ones;
 validCount++;
}
}

int[][] dp = new int[m + 1][n + 1];

// 遍历有效的字符串
for (int i = 0; i < validCount; i++) {
 int zeros = counts[i][0];
 int ones = counts[i][1];

 // 逆序遍历两个维度
 for (int j = m; j >= zeros; j--) {
 for (int k = n; k >= ones; k--) {
 dp[j][k] = Math.max(dp[j][k], dp[j - zeros][k - ones] + 1);
 }
 }
}

return dp[m][n];
}

/**
 * 使用滚动数组优化空间复杂度

```

```

* 注意: 在这个问题中, 滚动数组的优化已经在基本实现中完成 (使用二维数组)
* 这个方法只是为了展示如何进一步优化 (尽管在这个问题中意义不大)
*/
public static int findMaxFormScrolling(String[] strs, int m, int n) {
 if (strs == null || strs.length == 0) {
 return 0;
 }

 // 创建二维 DP 数组
 int[][] dp = new int[m + 1][n + 1];

 for (String str : strs) {
 int zeros = 0, ones = 0;
 for (char c : str.toCharArray()) {
 if (c == '0') {
 zeros++;
 } else {
 ones++;
 }
 }
 }

 // 逆序遍历两个维度 (这已经是滚动数组的思想)
 for (int i = m; i >= zeros; i--) {
 for (int j = n; j >= ones; j--) {
 dp[i][j] = Math.max(dp[i][j], dp[i - zeros][j - ones] + 1);
 }
 }

 return dp[m][n];
}

/***
 * 递归+记忆化搜索实现
 * 注意: 由于这个问题的参数范围较大, 递归+记忆化搜索可能会超时
 * 这里仅作为一种实现方式展示
*/
public static int findMaxFormDFS(String[] strs, int m, int n) {
 if (strs == null || strs.length == 0) {
 return 0;
 }

 int l = strs.length;

```

```

// 预处理每个字符串的 0 和 1 的个数
int[][] counts = new int[1][2];

for (int i = 0; i < 1; i++) {
 int zeros = 0, ones = 0;
 for (char c : strs[i].toCharArray()) {
 if (c == '0') {
 zeros++;
 } else {
 ones++;
 }
 }
 counts[i][0] = zeros;
 counts[i][1] = ones;
}

```

// 创建三维缓存数组，memo[k][i][j]表示考虑前 k 个字符串，最多使用 i 个 0 和 j 个 1 时，可以组成的最大子集的长度

```
Integer[][][] memo = new Integer[1][m + 1][n + 1];
```

```

// 调用递归辅助函数
return dfs(counts, 1 - 1, m, n, memo);
}
```

```

/**
 * 递归辅助函数
 * @param counts 每个字符串的 0 和 1 的个数
 * @param index 当前考虑的字符串索引
 * @param m0 剩余可用的 0 的个数
 * @param n1 剩余可用的 1 的个数
 * @param memo 缓存数组
 * @return 满足条件的最大子集长度
 */

```

```

private static int dfs(int[][] counts, int index, int m0, int n1, Integer[][][] memo) {
 // 基础情况：如果已经考虑完所有字符串，返回 0
 if (index < 0) {
 return 0;
 }
}
```

```

// 检查缓存
if (memo[index][m0][n1] != null) {
 return memo[index][m0][n1];
}
```

```

// 不选择当前字符串
int notChoose = dfs(counts, index - 1, m0, n1, memo);

// 选择当前字符串（如果可以的话）
int choose = 0;
int zeros = counts[index][0];
int ones = counts[index][1];

if (zeros <= m0 && ones <= n1) {
 choose = 1 + dfs(counts, index - 1, m0 - zeros, n1 - ones, memo);
}

// 缓存结果
memo[index][m0][n1] = Math.max(notChoose, choose);
return memo[index][m0][n1];
}
}

```

=====

文件: Code30\_OnesAndZeroes.py

=====

```

LeetCode 474. 一和零
题目描述: 给你一个二进制字符串数组 strs 和两个整数 m 和 n 。请你找出并返回 strs 的最大子集的长度,
该子集中最多有 m 个 0 和 n 个 1 。如果所有字符串都不满足条件, 返回 0 。
链接: https://leetcode.cn/problems/ones-and-zeroes/
#
解题思路:
这是一个二维费用的 0-1 背包问题:
- 每个字符串可以看作是一个物品
- 物品的重量有两个维度: 0 的个数和 1 的个数
- 背包的容量也有两个维度: m (最多 m 个 0) 和 n (最多 n 个 1)
- 我们需要找出最多能放多少个物品, 使得两个维度的重量都不超过背包容量
#
状态定义: dp[i][j] 表示最多使用 i 个 0 和 j 个 1 时, 可以组成的大子集的长度
状态转移方程: 对于每个字符串, 计算其包含的 0 的个数 zeros 和 1 的个数 ones
dp[i][j] = max(dp[i][j], dp[i-zeros][j-ones] + 1) (当 i >= zeros 且 j >= ones 时)
初始状态: dp[0][0] = 0, 表示不使用任何字符时, 子集长度为 0
#
时间复杂度: O(l * m * n), 其中 l 是字符串数组的长度, m 和 n 是给定的两个整数
空间复杂度: O(m * n), 使用二维 DP 数组

```

```

from typing import List
from functools import lru_cache

def findMaxForm(strs: List[str], m: int, n: int) -> int:
 """
 找出 strs 的最大子集长度，该子集中最多有 m 个 0 和 n 个 1

 Args:
 strs: 二进制字符串数组
 m: 最多允许的 0 的个数
 n: 最多允许的 1 的个数

 Returns:
 满足条件的最大子集长度
 """
 # 参数验证
 if not strs:
 return 0

 # 创建二维 DP 数组，dp[i][j] 表示最多使用 i 个 0 和 j 个 1 时，可以组成
 # 的最大子集的长度
 dp = [[0] * (n + 1) for _ in range(m + 1)]

 # 对于每个字符串，计算其包含的 0 的个数和 1 的个数
 for s in strs:
 zeros = s.count('0')
 ones = len(s) - zeros

 # 二维 0-1 背包问题，需要逆序遍历两个维度
 for i in range(m, zeros - 1, -1):
 for j in range(n, ones - 1, -1):
 # 状态转移：选择当前字符串或不选择当前字符串，取最大值
 dp[i][j] = max(dp[i][j], dp[i - zeros][j - ones] + 1)

 # 返回最多使用 m 个 0 和 n 个 1 时，可以组成
 # 的最大子集的长度
 return dp[m][n]

def findMaxFormOptimized(strs: List[str], m: int, n: int) -> int:
 """
 优化版本：预处理每个字符串的 0 和 1 的个数
 """

```

```

if not strs:
 return 0

l = len(strs)
预处理每个字符串的 0 和 1 的个数
counts = [] # counts[i][0] 表示第 i 个字符串中 0 的个数, counts[i][1] 表示 1 的个数

for s in strs:
 zeros = s.count('0')
 ones = len(s) - zeros
 counts.append([zeros, ones])

dp = [[0] * (n + 1) for _ in range(m + 1)]

遍历每个字符串
for zeros, ones in counts:
 # 逆序遍历两个维度
 for i in range(m, zeros - 1, -1):
 for j in range(n, ones - 1, -1):
 dp[i][j] = max(dp[i][j], dp[i - zeros][j - ones] + 1)

return dp[m][n]

```

```

def findMaxForm3D(strs: List[str], m: int, n: int) -> int:
 """
 三维 DP 数组实现 (更直观但空间复杂度更高)
 dp[k][i][j] 表示考虑前 k 个字符串, 最多使用 i 个 0 和 j 个 1 时, 可以组成
 的最大子集的长度
 """

 if not strs:
 return 0

 l = len(strs)
 # 预处理每个字符串的 0 和 1 的个数
 counts = []

 for s in strs:
 zeros = s.count('0')
 ones = len(s) - zeros
 counts.append([zeros, ones])

 # 创建三维 DP 数组
 dp = [[[0] * (n + 1) for _ in range(m + 1)] for __ in range(l + 1)]

```

```

填充 DP 数组
for k in range(1, l + 1):
 zeros, ones = counts[k - 1]

 for i in range(m + 1):
 for j in range(n + 1):
 # 不选择第 k 个字符串
 dp[k][i][j] = dp[k - 1][i][j]

 # 选择第 k 个字符串（如果可以的话）
 if i >= zeros and j >= ones:
 dp[k][i][j] = max(dp[k][i][j], dp[k - 1][i - zeros][j - ones] + 1)

return dp[l][m][n]

def findMaxFormPruned(strs: List[str], m: int, n: int) -> int:
 """
 提前剪枝优化
 """

 if not strs:
 return 0

 # 预处理每个字符串的 0 和 1 的个数，并过滤掉不可能被选中的字符串
 valid_counts = []

 for s in strs:
 zeros = s.count('0')
 ones = len(s) - zeros

 # 剪枝：如果字符串的 0 或 1 的个数超过给定的限制，则无法选择该字符串
 if zeros <= m and ones <= n:
 valid_counts.append([zeros, ones])

 dp = [[0] * (n + 1) for _ in range(m + 1)]

 # 遍历有效的字符串
 for zeros, ones in valid_counts:
 # 逆序遍历两个维度
 for i in range(m, zeros - 1, -1):
 for j in range(n, ones - 1, -1):
 dp[i][j] = max(dp[i][j], dp[i - zeros][j - ones] + 1)

```

```

return dp[m][n]

def findMaxFormScrolling(strs: List[str], m: int, n: int) -> int:
 """
 使用滚动数组优化空间复杂度
 注意：在这个问题中，滚动数组的优化已经在基本实现中完成（使用二维数组）
 这个方法只是为了展示如何进一步优化（尽管在这个问题中意义不大）
 """

 if not strs:
 return 0

 # 创建二维 DP 数组
 dp = [[0] * (n + 1) for _ in range(m + 1)]

 for s in strs:
 zeros = s.count('0')
 ones = len(s) - zeros

 # 逆序遍历两个维度（这已经是滚动数组的思想）
 for i in range(m, zeros - 1, -1):
 for j in range(n, ones - 1, -1):
 dp[i][j] = max(dp[i][j], dp[i - zeros][j - ones] + 1)

 return dp[m][n]

```

```

def findMaxFormDFS(strs: List[str], m: int, n: int) -> int:
 """
 递归+记忆化搜索实现
 注意：由于这个问题的参数范围较大，递归+记忆化搜索可能会超时
 这里仅作为一种实现方式展示
 """

 if not strs:
 return 0

 # 预处理每个字符串的 0 和 1 的个数
 counts = []
 for s in strs:
 zeros = s.count('0')
 ones = len(s) - zeros
 counts.append([zeros, ones])

```

```
使用 lru_cache 进行记忆化
@lru_cache(maxsize=None)
def dfs(index, m0, n1):
 """
 递归辅助函数

 Args:
 index: 当前考虑的字符串索引
 m0: 剩余可用的 0 的个数
 n1: 剩余可用的 1 的个数

 Returns:
 满足条件的最大子集长度
 """
 # 基础情况: 如果已经考虑完所有字符串, 返回 0
 if index < 0:
 return 0

 # 不选择当前字符串
 not_choose = dfs(index - 1, m0, n1)

 # 选择当前字符串 (如果可以的话)
 choose = 0
 zeros, ones = counts[index]

 if zeros <= m0 and ones <= n1:
 choose = 1 + dfs(index - 1, m0 - zeros, n1 - ones)

 # 返回最大值
 return max(not_choose, choose)

 # 调用递归函数
 return dfs(len(strs) - 1, m, n)

测试函数
def test_findMaxForm():
 # 测试用例 1
 strs1 = ["10", "0001", "111001", "1", "0"]
 m1, n1 = 5, 3
 print(f"测试用例 1 结果: {findMaxForm(strs1, m1, n1)}") # 预期输出: 4
```

```
测试用例 2
strs2 = ["10", "0", "1"]
m2, n2 = 1, 1
print(f"测试用例 2 结果: {findMaxForm(strs2, m2, n2)}") # 预期输出: 2
```

```
测试用例 3
strs3 = ["10", "0001", "111001", "1", "0"]
m3, n3 = 4, 3
print(f"测试用例 3 结果: {findMaxForm(strs3, m3, n3)}") # 预期输出: 3
```

```
执行测试
if __name__ == "__main__":
 test_findMaxForm()
```

```
=====
文件: Code31_CoinChange.cpp
=====
```

```
#include <iostream>
#include <vector>
#include <climits>
#include <algorithm>
#include <queue>

// LeetCode 322. 零钱兑换
// 题目描述: 给你一个整数数组 coins，表示不同面额的硬币；以及一个整数 amount，表示总金额。
// 计算并返回可以凑成总金额所需的最少的硬币个数。如果没有任何一种硬币组合能组成总金额，返回 -1。
// 你可以认为每种硬币的数量是无限的。
// 链接: https://leetcode.cn/problems/coin-change/
//
// 解题思路:
// 这是一个典型的完全背包问题:
// - 硬币可以多次使用（完全背包的特点）
// - 目标是凑成总金额，并且硬币个数最少（最优解问题）
//
// 状态定义: dp[i] 表示凑成金额 i 所需的最少硬币个数
// 状态转移方程: dp[i] = min(dp[i], dp[i - coin] + 1)，其中 coin 是每种硬币的面额，且 i >= coin
// 初始状态: dp[0] = 0 (凑成金额 0 不需要任何硬币)，其他位置初始化为一个较大的值（表示无法凑成）
//
// 时间复杂度: O(amount * n)，其中 n 是硬币的种类数
// 空间复杂度: O(amount)，使用一维 DP 数组
```

```

using namespace std;

/**
 * 计算凑成总金额所需的最少硬币个数
 * @param coins 不同面额的硬币数组
 * @param amount 总金额
 * @return 最少硬币个数，如果无法凑成则返回-1
 */
int coinChange(vector<int>& coins, int amount) {
 // 参数验证
 if (amount < 0) {
 return -1;
 }
 if (amount == 0) {
 return 0;
 }
 if (coins.empty()) {
 return -1;
 }

 // 创建 DP 数组，dp[i] 表示凑成金额 i 所需的最少硬币个数
 // 初始化为 amount + 1，因为最多使用 amount 个面值为 1 的硬币，所以 amount + 1 是一个不可能达到的
 // 值
 vector<int> dp(amount + 1, amount + 1);
 dp[0] = 0; // 凑成金额 0 不需要任何硬币

 // 遍历每种硬币
 for (int coin : coins) {
 // 完全背包问题，正序遍历金额（允许重复使用硬币）
 for (int i = coin; i <= amount; ++i) {
 // 状态转移：取不使用当前硬币和使用当前硬币的最小值
 dp[i] = min(dp[i], dp[i - coin] + 1);
 }
 }

 // 如果 dp[amount] 仍然是初始值，说明无法凑成总金额
 return dp[amount] > amount ? -1 : dp[amount];
}

/**
 * 优化版本：提前排序硬币，允许提前终止某些循环
 */
int coinChangeOptimized(vector<int>& coins, int amount) {

```

```

if (amount < 0) {
 return -1;
}
if (amount == 0) {
 return 0;
}
if (coins.empty()) {
 return -1;
}

// 排序硬币，从小到大
sort(coins.begin(), coins.end());

vector<int> dp(amount + 1, amount + 1);
dp[0] = 0;

for (int coin : coins) {
 // 如果当前硬币面额大于 amount，可以直接跳过
 if (coin > amount) {
 break;
 }

 for (int i = coin; i <= amount; ++i) {
 dp[i] = min(dp[i], dp[i - coin] + 1);
 }
}

return dp[amount] > amount ? -1 : dp[amount];
}

```

```

/**
 * 二维 DP 数组实现（更直观但空间复杂度更高）
 * dp[i][j] 表示使用前 i 种硬币，凑成金额 j 所需的最少硬币个数
 */

```

```

int coinChange2D(vector<int>& coins, int amount) {
 if (amount < 0) {
 return -1;
 }
 if (amount == 0) {
 return 0;
 }
 if (coins.empty()) {
 return -1;
 }

```

```

}

int n = coins.size();
// 创建二维 DP 数组
vector<vector<int>> dp(n + 1, vector<int>(amount + 1, amount + 1));
dp[0][0] = 0; // 溉成金额 0 不需要任何硬币

// 填充 DP 数组
for (int i = 1; i <= n; ++i) {
 int coin = coins[i - 1];

 for (int j = 0; j <= amount; ++j) {
 // 不使用第 i 种硬币
 dp[i][j] = dp[i - 1][j];

 // 使用第 i 种硬币 (如果可以的话)
 if (j >= coin) {
 // 完全背包问题：可以重复使用同一种硬币，所以是 dp[i][j-coin] 而不是 dp[i-1][j-coin]
 dp[i][j] = min(dp[i][j], dp[i][j - coin] + 1);
 }
 }
}

return dp[n][amount] > amount ? -1 : dp[n][amount];
}

/***
 * 递归+记忆化搜索实现
 */
int coinChangeDFS(vector<int>& coins, int amount) {
 if (amount < 0) {
 return -1;
 }
 if (amount == 0) {
 return 0;
 }
 if (coins.empty()) {
 return -1;
 }

 // 创建记忆化数组，memo[i] 表示湉成金额 i 所需的最少硬币个数
 vector<int> memo(amount + 1, -2); // -2 表示未计算过

```

```

memo[0] = 0;

function<int(int)> dfs = [&](int currentAmount) -> int {
 // 如果金额小于 0，无法凑成
 if (currentAmount < 0) {
 return -1;
 }

 // 如果已经计算过，直接返回
 if (memo[currentAmount] != -2) {
 return memo[currentAmount];
 }

 int minCoins = INT_MAX;

 // 尝试使用每种硬币
 for (int coin : coins) {
 int subResult = dfs(currentAmount - coin);
 // 如果子问题有解，更新最小硬币数
 if (subResult != -1) {
 minCoins = min(minCoins, subResult + 1);
 }
 }

 // 记录结果
 memo[currentAmount] = (minCoins == INT_MAX) ? -1 : minCoins;
 return memo[currentAmount];
};

return dfs(amount);
}

/***
 * BFS 实现：找到最少硬币个数，相当于找到从 0 到 amount 的最短路径
 */
int coinChangeBFS(vector<int>& coins, int amount) {
 if (amount < 0) {
 return -1;
 }
 if (amount == 0) {
 return 0;
 }
 if (coins.empty()) {

```

```
return -1;
}

// 使用 BFS，队列中存储当前金额
// 为了避免重复访问，使用 visited 数组记录已经访问过的金额
vector<bool> visited(amount + 1, false);
queue<int> q;
int step = 0;

// 初始状态：金额为 0
q.push(0);
visited[0] = true;

while (!q.empty()) {
 int size = q.size();
 step++;

 // 遍历当前层的所有节点
 for (int i = 0; i < size; ++i) {
 int current = q.front();
 q.pop();

 // 尝试使用每种硬币
 for (int coin : coins) {
 int next = current + coin;

 // 如果达到目标金额，返回当前步数
 if (next == amount) {
 return step;
 }

 // 如果 next 在有效范围内且未被访问过，加入队列
 if (next < amount && !visited[next]) {
 visited[next] = true;
 q.push(next);
 }
 }
 }
}

// 无法凑成总金额
return -1;
}
```

```

/***
 * 贪心算法 + DFS 剪枝（在某些情况下可能比动态规划更快）
 * 注意：贪心算法并不总是能得到最优解，因为可能存在较大面额的硬币虽然看起来更优，但会导致后续无法
凑成总金额
*/
int coinChangeGreedy(vector<int>& coins, int amount) {
 if (amount < 0) {
 return -1;
 }
 if (amount == 0) {
 return 0;
 }
 if (coins.empty()) {
 return -1;
 }

 // 按面值降序排序
 sort(coins.begin(), coins.end(), greater<int>());

 int minCoins = INT_MAX;

 function<void(int, int, int)> greedyDFS = [&](int index, int currentAmount, int count) {
 // 如果剩余金额为 0，更新最小硬币个数
 if (currentAmount == 0) {
 minCoins = min(minCoins, count);
 return;
 }

 // 如果已经考虑完所有硬币，或者当前硬币个数已经超过已知的最小硬币个数，返回
 if (index == coins.size() || count >= minCoins) {
 return;
 }

 int coin = coins[index];
 // 贪心策略：尽可能多地使用当前面额的硬币
 int maxCount = currentAmount / coin;

 // 从最多可以使用的个数开始尝试，逐步减少
 for (int i = maxCount; i >= 0; --i) {
 // 剪枝：如果当前硬币个数加上剩余金额用面值为 1 的硬币（最坏情况）都超过已知的最小硬币
个数，直接跳过
 if (count + i >= minCoins) {

```

```

 break;
 }

 // 递归尝试使用剩余的硬币凑成剩余的金额
 greedyDFS(index + 1, currentAmount - i * coin, count + i);
}

};

// 调用贪心 DFS
greedyDFS(0, amount, 0);

return minCoins == INT_MAX ? -1 : minCoins;
}

int main() {
 // 测试用例 1
 vector<int> coins1 = {1, 2, 5};
 int amount1 = 11;
 cout << "测试用例 1 结果: " << coinChange(coins1, amount1) << endl; // 预期输出: 3 (5+5+1)

 // 测试用例 2
 vector<int> coins2 = {2};
 int amount2 = 3;
 cout << "测试用例 2 结果: " << coinChange(coins2, amount2) << endl; // 预期输出: -1

 // 测试用例 3
 vector<int> coins3 = {1};
 int amount3 = 0;
 cout << "测试用例 3 结果: " << coinChange(coins3, amount3) << endl; // 预期输出: 0

 return 0;
}

```

=====

文件: Code31\_CoinChange.java

=====

```

package class073;

import java.util.Arrays;

// LeetCode 322. 零钱兑换
// 题目描述: 给你一个整数数组 coins，表示不同面额的硬币；以及一个整数 amount，表示总金额。

```

```
// 计算并返回可以凑成总金额所需的最少的硬币个数。如果没有任何一种硬币组合能组成总金额，返回 -1。
// 你可以认为每种硬币的数量是无限的。
// 链接: https://leetcode.cn/problems/coin-change/

// 解题思路:
// 这是一个典型的完全背包问题:
// - 硬币可以多次使用（完全背包的特点）
// - 目标是凑成总金额，并且硬币个数最少（最优解问题）

// 状态定义: dp[i] 表示凑成金额 i 所需的最少硬币个数
// 状态转移方程: dp[i] = min(dp[i], dp[i - coin] + 1)，其中 coin 是每种硬币的面额，且 i >= coin
// 初始状态: dp[0] = 0 (凑成金额 0 不需要任何硬币)，其他位置初始化为一个较大的值（表示无法凑成）

// 时间复杂度: O(amount * n)，其中 n 是硬币的种类数
// 空间复杂度: O(amount)，使用一维 DP 数组
```

```
public class Code31_CoinChange {

 // 主方法，用于测试
 public static void main(String[] args) {
 // 测试用例 1
 int[] coins1 = {1, 2, 5};
 int amount1 = 11;
 System.out.println("测试用例 1 结果: " + coinChange(coins1, amount1)); // 预期输出: 3
 (5+5+1)

 // 测试用例 2
 int[] coins2 = {2};
 int amount2 = 3;
 System.out.println("测试用例 2 结果: " + coinChange(coins2, amount2)); // 预期输出: -1

 // 测试用例 3
 int[] coins3 = {1};
 int amount3 = 0;
 System.out.println("测试用例 3 结果: " + coinChange(coins3, amount3)); // 预期输出: 0
 }

 /**
 * 计算凑成总金额所需的最少硬币个数
 * @param coins 不同面额的硬币数组
 * @param amount 总金额
 * @return 最少硬币个数，如果无法凑成则返回-1
 */
```

```

public static int coinChange(int[] coins, int amount) {
 // 参数验证
 if (amount < 0) {
 return -1;
 }
 if (amount == 0) {
 return 0;
 }
 if (coins == null || coins.length == 0) {
 return -1;
 }

 // 创建 DP 数组, dp[i] 表示凑成金额 i 所需的最少硬币个数
 // 初始化为 amount + 1, 因为最多使用 amount 个面值为 1 的硬币, 所以 amount + 1 是一个不可能达到的值
 int[] dp = new int[amount + 1];
 Arrays.fill(dp, amount + 1);
 dp[0] = 0; // 凑成金额 0 不需要任何硬币

 // 遍历每种硬币
 for (int coin : coins) {
 // 完全背包问题, 正序遍历金额 (允许重复使用硬币)
 for (int i = coin; i <= amount; i++) {
 // 状态转移: 取不使用当前硬币和使用当前硬币的最小值
 dp[i] = Math.min(dp[i], dp[i - coin] + 1);
 }
 }

 // 如果 dp[amount] 仍然是初始值, 说明无法凑成总金额
 return dp[amount] > amount ? -1 : dp[amount];
}

/**
 * 优化版本: 提前排序硬币, 允许提前终止某些循环
 */
public static int coinChangeOptimized(int[] coins, int amount) {
 if (amount < 0) {
 return -1;
 }
 if (amount == 0) {
 return 0;
 }
 if (coins == null || coins.length == 0) {

```

```

 return -1;
 }

 // 排序硬币，从小到大
 Arrays.sort(coins);

 int[] dp = new int[amount + 1];
 Arrays.fill(dp, amount + 1);
 dp[0] = 0;

 for (int coin : coins) {
 // 如果当前硬币面额大于 amount，可以直接跳过
 if (coin > amount) {
 break;
 }

 for (int i = coin; i <= amount; i++) {
 dp[i] = Math.min(dp[i], dp[i - coin] + 1);
 }
 }

 return dp[amount] > amount ? -1 : dp[amount];
}

/***
 * 二维 DP 数组实现（更直观但空间复杂度更高）
 * dp[i][j] 表示使用前 i 种硬币，凑成金额 j 所需的最少硬币个数
 */
public static int coinChange2D(int[] coins, int amount) {
 if (amount < 0) {
 return -1;
 }
 if (amount == 0) {
 return 0;
 }
 if (coins == null || coins.length == 0) {
 return -1;
 }

 int n = coins.length;
 // 创建二维 DP 数组
 int[][] dp = new int[n + 1][amount + 1];

```

```

// 初始化第一行（不使用任何硬币）
Arrays.fill(dp[0], amount + 1);
dp[0][0] = 0; // 溉成金额 0 不需要任何硬币

// 填充 DP 数组
for (int i = 1; i <= n; i++) {
 int coin = coins[i - 1];

 for (int j = 0; j <= amount; j++) {
 // 不使用第 i 种硬币
 dp[i][j] = dp[i - 1][j];

 // 使用第 i 种硬币（如果可以的话）
 if (j >= coin) {
 // 完全背包问题：可以重复使用同一种硬币，所以是 dp[i][j-coin] 而不是 dp[i-1][j-coin]
 dp[i][j] = Math.min(dp[i][j], dp[i][j - coin] + 1);
 }
 }
}

return dp[n][amount] > amount ? -1 : dp[n][amount];
}

/**
 * 递归+记忆化搜索实现
 */
public static int coinChangeDFS(int[] coins, int amount) {
 if (amount < 0) {
 return -1;
 }
 if (amount == 0) {
 return 0;
 }
 if (coins == null || coins.length == 0) {
 return -1;
 }

 // 创建记忆化数组，memo[i] 表示湉成金额 i 所需的最少硬币个数
 int[] memo = new int[amount + 1];
 Arrays.fill(memo, -2); // -2 表示未计算过
 memo[0] = 0;

```

```

 return dfs(coins, amount, memo);
 }

/***
 * 递归辅助函数
 * @param coins 硬币数组
 * @param amount 当前需要凑成的金额
 * @param memo 记忆化数组
 * @return 凑成当前金额所需的最少硬币个数，如果无法凑成则返回-1
 */
private static int dfs(int[] coins, int amount, int[] memo) {
 // 如果金额小于0，无法凑成
 if (amount < 0) {
 return -1;
 }

 // 如果已经计算过，直接返回
 if (memo[amount] != -2) {
 return memo[amount];
 }

 int minCoins = Integer.MAX_VALUE;

 // 尝试使用每种硬币
 for (int coin : coins) {
 int subResult = dfs(coins, amount - coin, memo);
 // 如果子问题有解，更新最小硬币数
 if (subResult != -1) {
 minCoins = Math.min(minCoins, subResult + 1);
 }
 }

 // 记录结果
 memo[amount] = (minCoins == Integer.MAX_VALUE) ? -1 : minCoins;
 return memo[amount];
}

/***
 * BFS 实现：找到最少硬币个数，相当于找到从 0 到 amount 的最短路径
 */
public static int coinChangeBFS(int[] coins, int amount) {
 if (amount < 0) {
 return -1;
 }
}

```

```
}

if (amount == 0) {
 return 0;
}

if (coins == null || coins.length == 0) {
 return -1;
}

// 使用 BFS，队列中存储当前金额和已使用的硬币个数
// 为了避免重复访问，使用 visited 数组记录已经访问过的金额
boolean[] visited = new boolean[amount + 1];
int[] queue = new int[amount + 1];
int front = 0, rear = 0;
int step = 0;

// 初始状态：金额为 0，使用 0 个硬币
queue[rear++] = 0;
visited[0] = true;

while (front < rear) {
 int size = rear - front;
 step++;

 // 遍历当前层的所有节点
 for (int i = 0; i < size; i++) {
 int current = queue[front++];

 // 尝试使用每种硬币
 for (int coin : coins) {
 int next = current + coin;

 // 如果达到目标金额，返回当前步数
 if (next == amount) {
 return step;
 }

 // 如果 next 在有效范围内且未被访问过，加入队列
 if (next < amount && !visited[next]) {
 visited[next] = true;
 queue[rear++] = next;
 }
 }
 }
}
```

```

 }

 // 无法凑成总金额
 return -1;
}

/***
 * 贪心算法 + DFS 剪枝（在某些情况下可能比动态规划更快）
 * 注意：贪心算法并不总是能得到最优解，因为可能存在较大面额的硬币虽然看起来更优，但会导致后续
无法凑成总金额
*/
public static int coinChangeGreedy(int[] coins, int amount) {
 if (amount < 0) {
 return -1;
 }
 if (amount == 0) {
 return 0;
 }
 if (coins == null || coins.length == 0) {
 return -1;
 }

 // 按面值降序排序
 Arrays.sort(coins);
 reverse(coins);

 int minCoins = Integer.MAX_VALUE;

 // 贪心DFS，尝试尽可能多地使用较大面额的硬币
 minCoins = greedyDFS(coins, 0, amount, 0, minCoins);

 return minCoins == Integer.MAX_VALUE ? -1 : minCoins;
}

/***
 * 贪心DFS辅助函数
 * @param coins 按面值降序排序的硬币数组
 * @param index 当前考虑的硬币索引
 * @param amount 剩余需要凑成的金额
 * @param count 已经使用的硬币个数
 * @param minCoins 当前找到的最小硬币个数
 * @return 最小硬币个数
*/

```

```

private static int greedyDFS(int[] coins, int index, int amount, int count, int minCoins) {
 // 如果剩余金额为 0，返回当前硬币个数
 if (amount == 0) {
 return Math.min(count, minCoins);
 }

 // 如果已经考虑完所有硬币，或者当前硬币个数已经超过已知的最小硬币个数，返回 minCoins
 if (index == coins.length || count >= minCoins) {
 return minCoins;
 }

 int coin = coins[index];
 // 贪心策略：尽可能多地使用当前面额的硬币
 int maxCount = amount / coin;

 // 从最多可以使用的个数开始尝试，逐步减少
 for (int i = maxCount; i >= 0; i--) {
 // 剪枝：如果当前硬币个数加上剩余金额用面值为 1 的硬币（最坏情况）都超过已知的最小硬币
 // 个数，直接跳过
 if (count + i >= minCoins) {
 break;
 }

 // 递归尝试使用剩余的硬币凑成剩余的金额
 minCoins = greedyDFS(coins, index + 1, amount - i * coin, count + i, minCoins);
 }
}

return minCoins;
}

/***
 * 反转数组
 */
private static void reverse(int[] arr) {
 int left = 0, right = arr.length - 1;
 while (left < right) {
 int temp = arr[left];
 arr[left] = arr[right];
 arr[right] = temp;
 left++;
 right--;
 }
}

```

```
}
```

```
=====
```

文件: Code31\_CoinChange.py

```
LeetCode 322. 零钱兑换
题目描述: 给你一个整数数组 coins , 表示不同面额的硬币; 以及一个整数 amount , 表示总金额。
计算并返回可以凑成总金额所需的最少的硬币个数。如果没有任何一种硬币组合能组成总金额, 返回 -1 。
你可以认为每种硬币的数量是无限的。
链接: https://leetcode.cn/problems/coin-change/
#
解题思路:
这是一个典型的完全背包问题:
- 硬币可以多次使用 (完全背包的特点)
- 目标是凑成总金额, 并且硬币个数最少 (最优解问题)
#
状态定义: dp[i] 表示凑成金额 i 所需的最少硬币个数
状态转移方程: dp[i] = min(dp[i], dp[i - coin] + 1), 其中 coin 是每种硬币的面额, 且 i >= coin
初始状态: dp[0] = 0 (凑成金额 0 不需要任何硬币), 其他位置初始化为一个较大的值 (表示无法凑成)
#
时间复杂度: O(amount * n), 其中 n 是硬币的种类数
空间复杂度: O(amount), 使用一维 DP 数组
```

```
from typing import List
from functools import lru_cache
import sys
```

```
def coinChange(coins: List[int], amount: int) -> int:
```

```
 """

```

计算凑成总金额所需的最少硬币个数

Args:

coins: 不同面额的硬币数组

amount: 总金额

Returns:

最少硬币个数, 如果无法凑成则返回-1

```
 """

```

# 参数验证

```
if amount < 0:
 return -1
```

```

if amount == 0:
 return 0
if not coins:
 return -1

创建 DP 数组, dp[i] 表示凑成金额 i 所需的最少硬币个数
初始化为 amount + 1, 因为最多使用 amount 个面值为 1 的硬币, 所以 amount + 1 是一个不可能达到的值
dp = [amount + 1] * (amount + 1)
dp[0] = 0 # 凑成金额 0 不需要任何硬币

遍历每种硬币
for coin in coins:
 # 完全背包问题, 正序遍历金额 (允许重复使用硬币)
 for i in range(coin, amount + 1):
 # 状态转移: 取不使用当前硬币和使用当前硬币的最小值
 dp[i] = min(dp[i], dp[i - coin] + 1)

如果 dp[amount] 仍然是初始值, 说明无法凑成总金额
return dp[amount] if dp[amount] <= amount else -1

```

```
def coinChangeOptimized(coins: List[int], amount: int) -> int:
 """

```

优化版本: 提前排序硬币, 允许提前终止某些循环

```
"""

```

```
if amount < 0:
 return -1
if amount == 0:
 return 0
if not coins:
 return -1
```

# 排序硬币, 从小到大

```
coins.sort()
```

```
dp = [amount + 1] * (amount + 1)
dp[0] = 0
```

```
for coin in coins:
```

# 如果当前硬币面额大于 amount, 可以直接跳过

```
 if coin > amount:
 break
```

```

for i in range(coin, amount + 1):
 dp[i] = min(dp[i], dp[i - coin] + 1)

return dp[amount] if dp[amount] <= amount else -1

def coinChange2D(coins: List[int], amount: int) -> int:
 """
 二维 DP 数组实现（更直观但空间复杂度更高）
 dp[i][j] 表示使用前 i 种硬币，凑成金额 j 所需的最少硬币个数
 """

 if amount < 0:
 return -1
 if amount == 0:
 return 0
 if not coins:
 return -1

 n = len(coins)
 # 创建二维 DP 数组
 dp = [[amount + 1] * (amount + 1) for _ in range(n + 1)]
 dp[0][0] = 0 # 凑成金额 0 不需要任何硬币

 # 填充 DP 数组
 for i in range(1, n + 1):
 coin = coins[i - 1]

 for j in range(amount + 1):
 # 不使用第 i 种硬币
 dp[i][j] = dp[i - 1][j]

 # 使用第 i 种硬币（如果可以的话）
 if j >= coin:
 # 完全背包问题：可以重复使用同一种硬币，所以是 dp[i][j-coin] 而不是 dp[i-1][j-coin]
 dp[i][j] = min(dp[i][j], dp[i][j - coin] + 1)

 return dp[n][amount] if dp[n][amount] <= amount else -1

def coinChangeDFS(coins: List[int], amount: int) -> int:
 """
 递归+记忆化搜索实现

```

```
"""
if amount < 0:
 return -1
if amount == 0:
 return 0
if not coins:
 return -1

使用 lru_cache 进行记忆化
@lru_cache(maxsize=None)
def dfs(current_amount: int) -> int:
 """
 递归辅助函数

 Args:
 current_amount: 当前需要凑成的金额

 Returns:
 凑成当前金额所需的最少硬币个数，如果无法凑成则返回-1
 """
 # 如果金额小于 0，无法凑成
 if current_amount < 0:
 return -1

 # 如果金额为 0，不需要硬币
 if current_amount == 0:
 return 0

 min_coins = float('inf')

 # 尝试使用每种硬币
 for coin in coins:
 sub_result = dfs(current_amount - coin)
 # 如果子问题有解，更新最小硬币数
 if sub_result != -1:
 min_coins = min(min_coins, sub_result + 1)

 # 返回结果，如果无法凑成返回-1
 return min_coins if min_coins != float('inf') else -1

return dfs(amount)
```

```
def coinChangeBFS(coins: List[int], amount: int) -> int:
 """
 BFS 实现：找到最少硬币个数，相当于找到从 0 到 amount 的最短路径
 """

 if amount < 0:
 return -1
 if amount == 0:
 return 0
 if not coins:
 return -1

 # 使用 BFS，队列中存储当前金额
 # 为了避免重复访问，使用 visited 数组记录已经访问过的金额
 visited = [False] * (amount + 1)
 queue = [0]
 visited[0] = True
 step = 0

 while queue:
 size = len(queue)
 step += 1

 # 遍历当前层的所有节点
 for _ in range(size):
 current = queue.pop(0)

 # 尝试使用每种硬币
 for coin in coins:
 next_amount = current + coin

 # 如果达到目标金额，返回当前步数
 if next_amount == amount:
 return step

 # 如果 next_amount 在有效范围内且未被访问过，加入队列
 if next_amount < amount and not visited[next_amount]:
 visited[next_amount] = True
 queue.append(next_amount)

 # 无法凑成总金额
 return -1
```

```

def coinChangeGreedy(coins: List[int], amount: int) -> int:
 """
 贪心算法 + DFS 剪枝（在某些情况下可能比动态规划更快）
 注意：贪心算法并不总是能得到最优解，因为可能存在较大面额的硬币虽然看起来更优，但会导致后续无法凑成总金额
 """

 if amount < 0:
 return -1
 if amount == 0:
 return 0
 if not coins:
 return -1

 # 按面值降序排序
 coins.sort(reverse=True)

 min_coins = float('inf')

 def greedy_dfs(index: int, current_amount: int, count: int) -> None:
 """
 贪心 DFS 辅助函数
 Args:
 index: 当前考虑的硬币索引
 current_amount: 剩余需要凑成的金额
 count: 已经使用的硬币个数
 """
 nonlocal min_coins

 # 如果剩余金额为 0，更新最小硬币个数
 if current_amount == 0:
 min_coins = min(min_coins, count)
 return

 # 如果已经考虑完所有硬币，或者当前硬币个数已经超过已知的最小硬币个数，返回
 if index == len(coins) or count >= min_coins:
 return

 coin = coins[index]
 # 贪心策略：尽可能多地使用当前面额的硬币
 max_count = current_amount // coin

 # 从最多可以使用的个数开始尝试，逐步减少

```

```

for i in range(max_count, -1, -1):
 # 剪枝：如果当前硬币个数加上剩余金额用面值为 1 的硬币（最坏情况）都超过已知的最小硬币
 # 个数，直接跳过
 if count + i >= min_coins:
 break

 # 递归尝试使用剩余的硬币凑成剩余的金额
 greedy_dfs(index + 1, current_amount - i * coin, count + i)

调用贪心 DFS
greedy_dfs(0, amount, 0)

return min_coins if min_coins != float('inf') else -1

测试函数
def test_coin_change():
 # 测试用例 1
 coins1 = [1, 2, 5]
 amount1 = 11
 print(f"测试用例 1 结果: {coinChange(coins1, amount1)}") # 预期输出: 3 (5+5+1)

 # 测试用例 2
 coins2 = [2]
 amount2 = 3
 print(f"测试用例 2 结果: {coinChange(coins2, amount2)}") # 预期输出: -1

 # 测试用例 3
 coins3 = [1]
 amount3 = 0
 print(f"测试用例 3 结果: {coinChange(coins3, amount3)}") # 预期输出: 0

执行测试
if __name__ == "__main__":
 test_coin_change()

```

---

文件: Code32\_CoinChangeII.cpp

---

```
#include <iostream>
#include <vector>
```

```

#include <algorithm>
#include <functional>

// LeetCode 518. 零钱兑换 II

// 题目描述：给你一个整数数组 coins 表示不同面额的硬币，另给一个整数 amount 表示总金额。
// 请你计算并返回可以凑成总金额的硬币组合数。如果任何硬币组合都无法凑出总金额，返回 0。
// 假设每一种面额的硬币有无限个。
// 链接: https://leetcode.cn/problems/coin-change-ii/
//

// 解题思路：
// 这是一个典型的完全背包问题，但目标是求组合数而不是最少硬币个数：
// - 硬币可以多次使用（完全背包的特点）
// - 目标是计算凑成总金额的不同组合数
//

// 状态定义：dp[i] 表示凑成金额 i 的不同组合数
// 状态转移方程：dp[i] += dp[i - coin]，其中 coin 是每种硬币的面额，且 i >= coin
// 初始状态：dp[0] = 1（凑成金额 0 只有一种方式：不使用任何硬币）
//

// 时间复杂度：O(amount * n)，其中 n 是硬币的种类数
// 空间复杂度：O(amount)，使用一维 DP 数组

using namespace std;

/**
 * 计算凑成总金额的不同硬币组合数
 * @param coins 不同面额的硬币数组
 * @param amount 总金额
 * @return 凑成总金额的不同组合数
 */
int change(vector<int>& coins, int amount) {
 // 参数验证
 if (amount < 0) {
 return 0;
 }
 if (coins.empty()) {
 return (amount == 0) ? 1 : 0;
 }

 // 创建 DP 数组，dp[i] 表示凑成金额 i 的不同组合数
 vector<long long> dp(amount + 1, 0); // 使用 long long 避免整数溢出
 dp[0] = 1; // 凑成金额 0 只有一种方式：不使用任何硬币

 // 遍历每种硬币

```

```

for (int coin : coins) {
 // 完全背包问题，正序遍历金额（允许重复使用硬币）
 // 注意：这里遍历硬币放在外层，金额放在内层，这样可以避免重复计算不同顺序的组合
 // 例如，对于 coins=[1, 2] 和 amount=3，如果先遍历金额再遍历硬币，会计算出 [1, 2] 和 [2, 1] 作为两种不同的组合
 for (int i = coin; i <= amount; ++i) {
 // 状态转移：当前金额可以由 (i - coin) 的金额加上一个 coin 得到
 dp[i] += dp[i - coin];
 }
}

return static_cast<int>(dp[amount]);
}

/***
 * 二维 DP 数组实现（更直观但空间复杂度更高）
 * dp[i][j] 表示使用前 i 种硬币，凑成金额 j 的不同组合数
 */
int change2D(vector<int>& coins, int amount) {
 if (amount < 0) {
 return 0;
 }
 if (coins.empty()) {
 return (amount == 0) ? 1 : 0;
 }

 int n = coins.size();
 // 创建二维 DP 数组
 vector<vector<long long>> dp(n + 1, vector<long long>(amount + 1, 0));

 // 初始化：使用 0 种硬币，只能凑成金额 0，有一种方式
 dp[0][0] = 1;

 // 填充 DP 数组
 for (int i = 1; i <= n; ++i) {
 int coin = coins[i - 1];

 for (int j = 0; j <= amount; ++j) {
 // 不使用第 i 种硬币
 dp[i][j] = dp[i - 1][j];

 // 使用第 i 种硬币（如果可以的话）
 // 完全背包问题：可以重复使用同一种硬币，所以是 dp[i][j - coin] 而不是 dp[i - 1][j - coin]
 if (j - coin >= 0) {
 dp[i][j] += dp[i][j - coin];
 }
 }
 }
}

```

```
 if (j >= coin) {
 dp[i][j] += dp[i][j - coin];
 }
 }

 return static_cast<int>(dp[n][amount]);
}

/***
 * 优化版本：提前处理特殊情况
 */
int changeOptimized(vector<int>& coins, int amount) {
 // 快速处理特殊情况
 if (amount < 0) {
 return 0;
 }
 if (amount == 0) {
 return 1;
 }
 if (coins.empty()) {
 return 0;
 }

 // 创建 DP 数组
 vector<long long> dp(amount + 1, 0);
 dp[0] = 1;

 for (int coin : coins) {
 // 如果当前硬币面额大于 amount，可以跳过
 if (coin > amount) {
 continue;
 }

 for (int i = coin; i <= amount; ++i) {
 dp[i] += dp[i - coin];
 }
 }

 return static_cast<int>(dp[amount]);
}
**/
```

\* 递归+记忆化搜索实现

\* 注意：由于这个问题的参数范围较大，递归+记忆化搜索可能会超时

\* 这里仅作为一种实现方式展示

\*/

```
int changeDFS(vector<int>& coins, int amount) {
```

```
 if (amount < 0) {
```

```
 return 0;
```

```
}
```

```
 if (amount == 0) {
```

```
 return 1;
```

```
}
```

```
 if (coins.empty()) {
```

```
 return 0;
```

```
}
```

// 为了避免重复计算不同顺序的组合，我们先对硬币进行排序，然后确保每次选择的硬币不小于上一次选择的硬币

```
sort(coins.begin(), coins.end());
```

```
int n = coins.size();
```

// 创建记忆化数组，memo[index][remain]表示从第 index 种硬币开始，凑成剩余金额 remain 的不同组合数

```
vector<vector<long long>> memo(n, vector<long long>(amount + 1, -1));
```

```
function<long long(int, int)> dfs = [&](int index, int remain) -> long long {
```

// 基础情况：如果剩余金额为 0，说明找到了一种组合

```
 if (remain == 0) {
```

```
 return 1;
```

```
}
```

// 基础情况：如果已经考虑完所有硬币，或者当前硬币面额大于剩余金额，无法凑成

```
 if (index >= n || coins[index] > remain) {
```

```
 return 0;
```

```
}
```

// 检查缓存

```
 if (memo[index][remain] != -1) {
```

```
 return memo[index][remain];
```

```
}
```

```
long long ways = 0;
```

// 计算使用当前硬币的不同次数的情况

```

// k 表示使用当前硬币的个数，从 0 开始
for (int k = 0; k * coins[index] <= remain; ++k) {
 // 递归计算不使用当前硬币 (k=0) 或使用 k 次当前硬币后的组合数
 ways += dfs(index + 1, remain - k * coins[index]);
}

// 缓存结果
memo[index][remain] = ways;
return ways;
};

return static_cast<int>(dfs(0, amount));
}

/***
 * 另一种递归实现方式，更加简洁
 */
int changeDFS2(vector<int>& coins, int amount) {
 if (amount < 0) {
 return 0;
 }
 if (amount == 0) {
 return 1;
 }
 if (coins.empty()) {
 return 0;
 }

// 排序硬币，避免重复计算
sort(coins.begin(), coins.end());

int n = coins.size();
vector<vector<long long>> memo(n, vector<long long>(amount + 1, -1));

function<long long(int, int)> dfs = [&](int index, int remain) -> long long {
 if (remain == 0) {
 return 1;
 }

 if (index >= n || coins[index] > remain) {
 return 0;
 }
}

```

```

 if (memo[index][remain] != -1) {
 return memo[index][remain];
 }

 // 不使用当前硬币的情况
 long long skip = dfs(index + 1, remain);

 // 使用当前硬币的情况（可以继续使用当前硬币）
 long long use = dfs(index, remain - coins[index]);

 // 缓存结果
 memo[index][remain] = skip + use;
 return memo[index][remain];
}

return static_cast<int>(dfs(0, amount));
}

int main() {
 // 测试用例 1
 vector<int> coins1 = {1, 2, 5};
 int amount1 = 5;
 cout << "测试用例 1 结果: " << change(coins1, amount1) << endl; // 预期输出: 4 ([1,1,1,1,1], [1,1,1,2], [1,2,2], [5])

 // 测试用例 2
 vector<int> coins2 = {2};
 int amount2 = 3;
 cout << "测试用例 2 结果: " << change(coins2, amount2) << endl; // 预期输出: 0

 // 测试用例 3
 vector<int> coins3 = {10};
 int amount3 = 10;
 cout << "测试用例 3 结果: " << change(coins3, amount3) << endl; // 预期输出: 1

 return 0;
}
=====
```

文件: Code32\_CoinChangeII.java

```
=====
package class073;
```

```

// LeetCode 518. 零钱兑换 II
// 题目描述：给你一个整数数组 coins 表示不同面额的硬币，另给一个整数 amount 表示总金额。
// 请你计算并返回可以凑成总金额的硬币组合数。如果任何硬币组合都无法凑出总金额，返回 0 。
// 假设每一种面额的硬币有无限个。
// 链接: https://leetcode.cn/problems/coin-change-ii/
//
// 解题思路：
// 这是一个典型的完全背包问题，但目标是求组合数而不是最少硬币个数：
// - 硬币可以多次使用（完全背包的特点）
// - 目标是计算凑成总金额的不同组合数
//
// 状态定义：dp[i] 表示凑成金额 i 的不同组合数
// 状态转移方程：dp[i] += dp[i - coin]，其中 coin 是每种硬币的面额，且 i >= coin
// 初始状态：dp[0] = 1（凑成金额 0 只有一种方式：不使用任何硬币）
//
// 时间复杂度：O(amount * n)，其中 n 是硬币的种类数
// 空间复杂度：O(amount)，使用一维 DP 数组

public class Code32_CoinChangeII {

 // 主方法，用于测试
 public static void main(String[] args) {
 // 测试用例 1
 int[] coins1 = {1, 2, 5};
 int amount1 = 5;
 System.out.println("测试用例 1 结果：" + change(coins1, amount1)); // 预期输出: 4
 ([1, 1, 1, 1, 1], [1, 1, 1, 2], [1, 2, 2], [5])

 // 测试用例 2
 int[] coins2 = {2};
 int amount2 = 3;
 System.out.println("测试用例 2 结果：" + change(coins2, amount2)); // 预期输出: 0

 // 测试用例 3
 int[] coins3 = {10};
 int amount3 = 10;
 System.out.println("测试用例 3 结果：" + change(coins3, amount3)); // 预期输出: 1
 }

 /**
 * 计算凑成总金额的不同硬币组合数
 * @param coins 不同面额的硬币数组

```

```

* @param amount 总金额
* @return 求成总金额的不同组合数
*/
public static int change(int[] coins, int amount) {
 // 参数验证
 if (amount < 0) {
 return 0;
 }
 if (coins == null) {
 return 0;
 }

 // 创建 DP 数组, dp[i] 表示求成金额 i 的不同组合数
 int[] dp = new int[amount + 1];
 dp[0] = 1; // 求成金额 0 只有一种方式: 不使用任何硬币

 // 遍历每种硬币
 for (int coin : coins) {
 // 完全背包问题, 正序遍历金额 (允许重复使用硬币)
 // 注意: 这里遍历硬币放在外层, 金额放在内层, 这样可以避免重复计算不同顺序的组合
 // 例如, 对于 coins=[1, 2] 和 amount=3, 如果先遍历金额再遍历硬币, 会计算出 [1, 2] 和 [2, 1] 作为两种不同的组合
 for (int i = coin; i <= amount; i++) {
 // 状态转移: 当前金额可以由 (i-coin) 的金额加上一个 coin 得到
 dp[i] += dp[i - coin];
 }
 }

 return dp[amount];
}

/**
 * 二维 DP 数组实现 (更直观但空间复杂度更高)
 * dp[i][j] 表示使用前 i 种硬币, 求成金额 j 的不同组合数
 */
public static int change2D(int[] coins, int amount) {
 if (amount < 0) {
 return 0;
 }
 if (coins == null) {
 return 0;
 }
}

```

```

int n = coins.length;
// 创建二维 DP 数组
int[][] dp = new int[n + 1][amount + 1];

// 初始化: 使用 0 种硬币, 只能凑成金额 0, 有一种方式
for (int j = 0; j <= amount; j++) {
 dp[0][j] = 0;
}
dp[0][0] = 1;

// 填充 DP 数组
for (int i = 1; i <= n; i++) {
 int coin = coins[i - 1];

 for (int j = 0; j <= amount; j++) {
 // 不使用第 i 种硬币
 dp[i][j] = dp[i - 1][j];

 // 使用第 i 种硬币 (如果可以的话)
 // 完全背包问题: 可以重复使用同一种硬币, 所以是 dp[i][j-coin] 而不是 dp[i-1][j-coin]
 if (j >= coin) {
 dp[i][j] += dp[i][j - coin];
 }
 }
}

return dp[n][amount];
}

/**
 * 优化版本: 提前处理特殊情况
 */
public static int changeOptimized(int[] coins, int amount) {
 // 快速处理特殊情况
 if (amount < 0) {
 return 0;
 }
 if (amount == 0) {
 return 1;
 }
 if (coins == null || coins.length == 0) {
 return 0;
 }
}

```

```

}

// 创建 DP 数组
int[] dp = new int[amount + 1];
dp[0] = 1;

for (int coin : coins) {
 // 如果当前硬币面额大于 amount，可以跳过
 if (coin > amount) {
 continue;
 }

 for (int i = coin; i <= amount; i++) {
 dp[i] += dp[i - coin];
 }
}

return dp[amount];
}

/***
 * 递归+记忆化搜索实现
 * 注意：由于这个问题的参数范围较大，递归+记忆化搜索可能会超时
 * 这里仅作为一种实现方式展示
 */
public static int changeDFS(int[] coins, int amount) {
 if (amount < 0) {
 return 0;
 }
 if (amount == 0) {
 return 1;
 }
 if (coins == null || coins.length == 0) {
 return 0;
 }

 // 为了避免重复计算不同顺序的组合，我们先对硬币进行排序，然后确保每次选择的硬币不小于上一次选择的硬币
 java.util.Arrays.sort(coins);

 // 创建记忆化数组，memo[index][remain]表示从第 index 种硬币开始，凑成剩余金额 remain 的不同组合数
 Integer[][] memo = new Integer[coins.length][amount + 1];

```

```

 return dfs(coins, 0, amount, memo);
}

/***
 * 递归辅助函数
 * @param coins 硬币数组
 * @param index 当前考虑的硬币索引
 * @param remain 剩余需要凑成的金额
 * @param memo 记忆化数组
 * @return 从当前索引开始，凑成剩余金额的不同组合数
*/
private static int dfs(int[] coins, int index, int remain, Integer[][] memo) {
 // 基础情况：如果剩余金额为 0，说明找到了一种组合
 if (remain == 0) {
 return 1;
 }

 // 基础情况：如果已经考虑完所有硬币，或者当前硬币面额大于剩余金额，无法凑成
 if (index >= coins.length || coins[index] > remain) {
 return 0;
 }

 // 检查缓存
 if (memo[index][remain] != null) {
 return memo[index][remain];
 }

 int ways = 0;

 // 计算使用当前硬币的不同次数的情况
 // k 表示使用当前硬币的个数，从 0 开始
 for (int k = 0; k * coins[index] <= remain; k++) {
 // 递归计算不使用当前硬币 (k=0) 或使用 k 次当前硬币后的组合数
 ways += dfs(coins, index + 1, remain - k * coins[index], memo);
 }

 // 缓存结果
 memo[index][remain] = ways;
 return ways;
}

/***

```

\* 另一种递归实现方式，更加简洁

\*/

```
public static int changeDFS2(int[] coins, int amount) {
 if (amount < 0) {
 return 0;
 }
 if (amount == 0) {
 return 1;
 }
 if (coins == null || coins.length == 0) {
 return 0;
 }
}
```

// 排序硬币，避免重复计算

```
java.util.Arrays.sort(coins);
```

```
Integer[][] memo = new Integer[coins.length][amount + 1];
```

```
return dfs2(coins, 0, amount, memo);
```

```
}
```

/\*\*

\* 递归辅助函数 - 更简洁的实现

\* @param coins 硬币数组

\* @param index 当前考虑的硬币索引

\* @param remain 剩余需要凑成的金额

\* @param memo 记忆化数组

\* @return 从当前索引开始，凑成剩余金额的不同组合数

\*/

```
private static int dfs2(int[] coins, int index, int remain, Integer[][] memo) {
```

```
 if (remain == 0) {
```

```
 return 1;
 }
```

```
}
```

```
 if (index >= coins.length || coins[index] > remain) {
```

```
 return 0;
 }
```

```
 if (memo[index][remain] != null) {
```

```
 return memo[index][remain];
 }
```

```
}
```

// 不使用当前硬币的情况

```

int skip = dfs2(coins, index + 1, remain, memo);

// 使用当前硬币的情况（可以继续使用当前硬币）
int use = dfs2(coins, index, remain - coins[index], memo);

// 缓存结果
memo[index][remain] = skip + use;
return memo[index][remain];
}

}
=====
```

文件: Code32\_CoinChangeII.py

```
=====
```

```

LeetCode 518. 零钱兑换 II
题目描述: 给你一个整数数组 coins 表示不同面额的硬币，另给一个整数 amount 表示总金额。
请你计算并返回可以凑成总金额的硬币组合数。如果任何硬币组合都无法凑出总金额，返回 0 。
假设每一种面额的硬币有无限个。
链接: https://leetcode.cn/problems/coin-change-ii/
#
解题思路:
这是一个典型的完全背包问题，但目标是求组合数而不是最少硬币个数:
- 硬币可以多次使用（完全背包的特点）
- 目标是计算凑成总金额的不同组合数
#
状态定义: dp[i] 表示凑成金额 i 的不同组合数
状态转移方程: dp[i] += dp[i - coin]，其中 coin 是每种硬币的面额，且 i >= coin
初始状态: dp[0] = 1 (凑成金额 0 只有一种方式: 不使用任何硬币)
#
时间复杂度: O(amount * n)，其中 n 是硬币的种类数
空间复杂度: O(amount)，使用一维 DP 数组
```

```

from typing import List
from functools import lru_cache
```

```

def change(coins: List[int], amount: int) -> int:
 """

```

计算凑成总金额的不同硬币组合数

Args:

coins: 不同面额的硬币数组

```
amount: 总金额
```

```
Returns:
```

```
 捆成总金额的不同组合数
```

```
"""
```

```
参数验证
```

```
if amount < 0:
```

```
 return 0
```

```
if amount == 0:
```

```
 return 1
```

```
if not coins:
```

```
 return 0
```

```
创建 DP 数组, dp[i] 表示捆成金额 i 的不同组合数
```

```
dp = [0] * (amount + 1)
```

```
dp[0] = 1 # 捆成金额 0 只有一种方式: 不使用任何硬币
```

```
遍历每种硬币
```

```
for coin in coins:
```

```
 # 完全背包问题, 正序遍历金额 (允许重复使用硬币)
```

```
 # 注意: 这里遍历硬币放在外层, 金额放在内层, 这样可以避免重复计算不同顺序的组合
```

```
 # 例如, 对于 coins=[1, 2] 和 amount=3, 如果先遍历金额再遍历硬币, 会计算出 [1, 2] 和 [2, 1] 作为两种不同的组合
```

```
 for i in range(coin, amount + 1):
```

```
 # 状态转移: 当前金额可以由 (i-coin) 的金额加上一个 coin 得到
```

```
 dp[i] += dp[i - coin]
```

```
return dp[amount]
```

```
def change2D(coins: List[int], amount: int) -> int:
```

```
"""
```

```
二维 DP 数组实现 (更直观但空间复杂度更高)
```

```
dp[i][j] 表示使用前 i 种硬币, 捆成金额 j 的不同组合数
```

```
"""
```

```
if amount < 0:
```

```
 return 0
```

```
if amount == 0:
```

```
 return 1
```

```
if not coins:
```

```
 return 0
```

```
n = len(coins)
```

```

创建二维 DP 数组
dp = [[0] * (amount + 1) for _ in range(n + 1)]

初始化：使用 0 种硬币，只能凑成金额 0，有一种方式
dp[0][0] = 1

填充 DP 数组
for i in range(1, n + 1):
 coin = coins[i - 1]

 for j in range(amount + 1):
 # 不使用第 i 种硬币
 dp[i][j] = dp[i - 1][j]

 # 使用第 i 种硬币（如果可以的话）
 # 完全背包问题：可以重复使用同一种硬币，所以是 dp[i][j-coin] 而不是 dp[i-1][j-coin]
 if j >= coin:
 dp[i][j] += dp[i][j - coin]

return dp[n][amount]

```

```
def changeOptimized(coins: List[int], amount: int) -> int:
```

```
"""

```

```
优化版本：提前处理特殊情况
"""

```

```
快速处理特殊情况

```

```
if amount < 0:
 return 0

```

```
if amount == 0:
 return 1

```

```
if not coins:
 return 0

```

```
创建 DP 数组

```

```
dp = [0] * (amount + 1)

```

```
dp[0] = 1

```

```
for coin in coins:

```

```
 # 如果当前硬币面额大于 amount，可以跳过

```

```
 if coin > amount:

```

```
 continue

```

```
for i in range(coin, amount + 1):
 dp[i] += dp[i - coin]
```

```
return dp[amount]
```

```
def changeDFS(coins: List[int], amount: int) -> int:
```

```
"""
```

递归+记忆化搜索实现

注意：由于这个问题的参数范围较大，递归+记忆化搜索可能会超时

这里仅作为一种实现方式展示

```
"""
```

```
if amount < 0:
 return 0
```

```
if amount == 0:
 return 1
```

```
if not coins:
 return 0
```

# 为了避免重复计算不同顺序的组合，我们先对硬币进行排序，然后确保每次选择的硬币不小于上一次选择的硬币

```
coins.sort()
```

```
n = len(coins)
```

```
@lru_cache(maxsize=None)
```

```
def dfs(index: int, remain: int) -> int:
 """
```

递归辅助函数

Args:

index: 当前考虑的硬币索引

remain: 剩余需要凑成的金额

Returns:

从当前索引开始，凑成剩余金额的不同组合数

```
"""
```

# 基础情况：如果剩余金额为 0，说明找到了一种组合

```
if remain == 0:
 return 1
```

# 基础情况：如果已经考虑完所有硬币，或者当前硬币面额大于剩余金额，无法凑成

```
if index >= n or coins[index] > remain:
```

```

 return 0

ways = 0

计算使用当前硬币的不同次数的情况
k 表示使用当前硬币的个数，从 0 开始
k = 0
while k * coins[index] <= remain:
 # 递归计算不使用当前硬币 (k=0) 或使用 k 次当前硬币后的组合数
 ways += dfs(index + 1, remain - k * coins[index])
 k += 1

return ways

return dfs(0, amount)

```

```
def changeDFS2(coins: List[int], amount: int) -> int:
```

```
"""

```

```
另一种递归实现方式，更加简洁
"""

```

```
if amount < 0:
 return 0
```

```
if amount == 0:
 return 1
```

```
if not coins:
 return 0
```

```
排序硬币，避免重复计算
```

```
coins.sort()
```

```
n = len(coins)
```

```
@lru_cache(maxsize=None)
```

```
def dfs(index: int, remain: int) -> int:
 """

```

```
递归辅助函数 - 更简洁的实现
```

Args:

index: 当前考虑的硬币索引

remain: 剩余需要凑成的金额

Returns:

```
从当前索引开始，凑成剩余金额的不同组合数
"""

if remain == 0:
 return 1

if index >= n or coins[index] > remain:
 return 0

不使用当前硬币的情况
skip = dfs(index + 1, remain)

使用当前硬币的情况（可以继续使用当前硬币）
use = dfs(index, remain - coins[index])

return skip + use

return dfs(0, amount)
```

```
测试函数
def test_change():
 # 测试用例 1
 coins1 = [1, 2, 5]
 amount1 = 5
 print(f"测试用例 1 结果: {change(coins1, amount1)}") # 预期输出: 4 ([1, 1, 1, 1, 1], [1, 1, 1, 2], [1, 2, 2], [5])

 # 测试用例 2
 coins2 = [2]
 amount2 = 3
 print(f"测试用例 2 结果: {change(coins2, amount2)}") # 预期输出: 0

 # 测试用例 3
 coins3 = [10]
 amount3 = 10
 print(f"测试用例 3 结果: {change(coins3, amount3)}") # 预期输出: 1

 # 测试二维 DP 实现
 print(f"测试用例 1 (二维 DP): {change2D(coins1, amount1)}")

 # 测试优化版本
 print(f"测试用例 1 (优化版本): {changeOptimized(coins1, amount1)})")
```

```
测试 DFS 实现
print(f"测试用例 1 (DFS): {changeDFS(coins1, amount1)}")

测试 DFS2 实现
print(f"测试用例 1 (DFS2): {changeDFS2(coins1, amount1)}")

执行测试
if __name__ == "__main__":
 test_change()

=====
```

文件: Code33\_CombinationSumIV.cpp

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <unordered_map>
#include <functional>

// LeetCode 377. 组合总和 IV
// 题目描述: 给你一个由 不同 整数组成的数组 nums，和一个目标整数 target。
// 请你从 nums 中找出并返回总和为 target 的元素组合的个数。
// 注意: 顺序不同的序列被视作不同的组合。
// 链接: https://leetcode.cn/problems/combination-sum-iv/
//
// 解题思路:
// 这是一个与完全背包相关但关注排列而非组合的问题:
// - 数字可以多次使用 (完全背包的特点)
// - 顺序不同的序列视为不同的组合 (与组合数问题的关键区别)
//
// 状态定义: dp[i] 表示凑成目标值 i 的不同排列数
// 状态转移方程: dp[i] += dp[i - num]，其中 num 是 nums 中的每个元素，且 i >= num
// 初始状态: dp[0] = 1 (凑成目标值 0 只有一种方式: 不选择任何数字)
//
// 时间复杂度: O(target * n)，其中 n 是数组 nums 的长度
// 空间复杂度: O(target)，使用一维 DP 数组
```

```
using namespace std;
```

```
/**
 * 计算凑成目标值的不同排列数
```

```

* @param nums 不同整数组成的数组
* @param target 目标整数
* @return 总和为 target 的元素组合的个数
*/
int combinationSum4(vector<int>& nums, int target) {
 // 参数验证
 if (target < 0) {
 return 0;
 }
 if (target == 0) {
 return 1;
 }
 if (nums.empty()) {
 return 0;
 }

 // 创建 DP 数组, dp[i] 表示凑成目标值 i 的不同排列数
 // 注意: 由于可能的数值较大, 我们需要处理溢出问题
 vector<long long> dp(target + 1, 0);
 dp[0] = 1; // 凑成目标值 0 只有一种方式: 不选择任何数字

 // 遍历目标值, 从 1 到 target
 // 注意: 与零钱兑换 II 不同, 这里我们先遍历目标值, 再遍历数组元素, 这样可以考虑不同顺序的排列
 for (int i = 1; i <= target; ++i) {
 // 遍历数组中的每个元素
 for (int num : nums) {
 // 如果当前元素小于等于剩余需要凑成的目标值, 更新 dp[i]
 if (num <= i) {
 dp[i] += dp[i - num];
 // 防止溢出(题目保证结果在 32 位有符号整数范围内)
 if (dp[i] > INT_MAX) {
 dp[i] = INT_MAX;
 }
 }
 }
 }

 return static_cast<int>(dp[target]);
}

/**
 * 递归+记忆化搜索实现
 * @param nums 不同整数组成的数组

```

```

* @param target 目标整数
* @return 总和为 target 的元素组合的个数
*/
int combinationSum4DFS(vector<int>& nums, int target) {
 // 参数验证
 if (target < 0) {
 return 0;
 }
 if (target == 0) {
 return 1;
 }
 if (nums.empty()) {
 return 0;
 }

 // 使用 unordered_map 作为记忆化缓存
 unordered_map<int, int> memo;
 memo[0] = 1; // 溉成目标值 0 只有一种方式

 function<int(int)> dfs = [&](int remain) -> int {
 // 检查缓存
 if (memo.find(remain) != memo.end()) {
 return memo[remain];
 }

 int ways = 0;

 // 尝试使用每个元素
 for (int num : nums) {
 if (num <= remain) {
 // 递归计算剩余值的排列数
 ways += dfs(remain - num);
 // 防止溢出
 if (ways > INT_MAX) {
 ways = INT_MAX;
 }
 }
 }

 // 缓存结果
 memo[remain] = ways;
 return ways;
 };
}

```

```

 return dfs(target);
}

/***
 * 优化版本：提前排序和剪枝
 * @param nums 不同整数组成的数组
 * @param target 目标整数
 * @return 总和为 target 的元素组合的个数
*/
int combinationSum4Optimized(vector<int>& nums, int target) {
 // 参数验证
 if (target < 0) {
 return 0;
 }
 if (target == 0) {
 return 1;
 }
 if (nums.empty()) {
 return 0;
 }

 // 对数组进行排序，以便在后续处理中进行剪枝
 sort(nums.begin(), nums.end());

 // 创建 DP 数组
 vector<long long> dp(target + 1, 0);
 dp[0] = 1;

 // 遍历目标值
 for (int i = 1; i <= target; ++i) {
 // 遍历数组中的元素
 for (int num : nums) {
 // 如果当前元素大于剩余需要凑成的目标值，由于数组已排序，后面的元素更大，可以提前退出
 // 循环
 if (num > i) {
 break;
 }
 dp[i] += dp[i - num];
 // 防止溢出
 if (dp[i] > INT_MAX) {
 dp[i] = INT_MAX;
 }
 }
 }
}

```

```

 }

}

return static_cast<int>(dp[target]);
}

/***
 * 递归+记忆化搜索实现的另一种方式，使用数组作为缓存
 * @param nums 不同整数组成的数组
 * @param target 目标整数
 * @return 总和为 target 的元素组合的个数
*/
int combinationSum4DFSArray(vector<int>& nums, int target) {
 // 参数验证
 if (target < 0) {
 return 0;
 }
 if (target == 0) {
 return 1;
 }
 if (nums.empty()) {
 return 0;
 }

 // 使用数组作为缓存，初始值为-1 表示未计算
 vector<int> memo(target + 1, -1);
 memo[0] = 1; // 溉成目标值 0 只有一种方式

 function<int(int)> dfs = [&](int remain) -> int {
 // 检查缓存
 if (memo[remain] != -1) {
 return memo[remain];
 }

 int ways = 0;

 // 尝试使用每个元素
 for (int num : nums) {
 if (num <= remain) {
 ways += dfs(remain - num);
 // 防止溢出
 if (ways > INT_MAX) {
 ways = INT_MAX;
 }
 }
 }
 memo[remain] = ways;
 return ways;
 };
}

int combinationSum4(vector<int>& nums, int target) {
 return combinationSum4DFSArray(nums, target);
}

```

```

 }
 }

 // 缓存结果
 memo[remain] = ways;
 return ways;
};

return dfs(target);
}

int main() {
 // 测试用例 1
 vector<int> nums1 = {1, 2, 3};
 int target1 = 4;
 cout << "测试用例 1 结果：" << combinationSum4(nums1, target1) << endl; // 预期输出: 7
([1, 1, 1, 1], [1, 1, 2], [1, 2, 1], [1, 3], [2, 1, 1], [2, 2], [3, 1])

 // 测试用例 2
 vector<int> nums2 = {9};
 int target2 = 3;
 cout << "测试用例 2 结果：" << combinationSum4(nums2, target2) << endl; // 预期输出: 0

 // 测试 DFS 实现
 cout << "测试用例 1 (DFS)：" << combinationSum4DFS(nums1, target1) << endl;

 // 测试优化版本
 cout << "测试用例 1 (优化版本)：" << combinationSum4Optimized(nums1, target1) << endl;

 // 测试 DFS+数组缓存实现
 cout << "测试用例 1 (DFS+数组缓存)：" << combinationSum4DFSArry(nums1, target1) << endl;

 return 0;
}
=====

文件: Code33_CombinationSumIV.java
=====

package class073;

import java.util.Arrays;

```

文件: Code33\_CombinationSumIV.java

```
=====
package class073;
```

```
import java.util.Arrays;
```

```

import java.util.HashMap;
import java.util.Map;

// LeetCode 377. 组合总和 IV
// 题目描述: 给你一个由 不同 整数组成的数组 nums，和一个目标整数 target。
// 请你从 nums 中找出并返回总和为 target 的元素组合的个数。
// 注意: 顺序不同的序列被视作不同的组合。
// 链接: https://leetcode.cn/problems/combination-sum-iv/
//
// 解题思路:
// 这是一个与完全背包相关但关注排列而非组合的问题:
// - 数字可以多次使用 (完全背包的特点)
// - 顺序不同的序列视为不同的组合 (与组合数问题的关键区别)
//
// 状态定义: dp[i] 表示凑成目标值 i 的不同排列数
// 状态转移方程: dp[i] += dp[i - num]，其中 num 是 nums 中的每个元素，且 i >= num
// 初始状态: dp[0] = 1 (凑成目标值 0 只有一种方式: 不选择任何数字)
//
// 时间复杂度: O(target * n)，其中 n 是数组 nums 的长度
// 空间复杂度: O(target)，使用一维 DP 数组

public class Code33_CombinationSumIV {

 // 主方法, 用于测试
 public static void main(String[] args) {
 // 测试用例 1
 int[] nums1 = {1, 2, 3};
 int target1 = 4;
 System.out.println("测试用例 1 结果: " + combinationSum4(nums1, target1)); // 预期输出: 7
 ([1, 1, 1, 1], [1, 1, 2], [1, 2, 1], [1, 3], [2, 1, 1], [2, 2], [3, 1])

 // 测试用例 2
 int[] nums2 = {9};
 int target2 = 3;
 System.out.println("测试用例 2 结果: " + combinationSum4(nums2, target2)); // 预期输出: 0
 }

 /**
 * 计算凑成目标值的不同排列数
 * @param nums 不同整数组成的数组
 * @param target 目标整数
 * @return 总和为 target 的元素组合的个数
 */
}

```

```

public static int combinationSum4(int[] nums, int target) {
 // 参数验证
 if (target < 0) {
 return 0;
 }
 if (target == 0) {
 return 1;
 }
 if (nums == null || nums.length == 0) {
 return 0;
 }

 // 创建 DP 数组, dp[i] 表示凑成目标值 i 的不同排列数
 // 注意: 由于可能的数值较大, 我们需要处理溢出问题
 // 这里使用 long 类型, 最后再转换为 int
 long[] dp = new long[target + 1];
 dp[0] = 1; // 凑成目标值 0 只有一种方式: 不选择任何数字

 // 遍历目标值, 从 1 到 target
 // 注意: 与零钱兑换 II 不同, 这里我们先遍历目标值, 再遍历数组元素, 这样可以考虑不同顺序的
 // 排列
 for (int i = 1; i <= target; i++) {
 // 遍历数组中的每个元素
 for (int num : nums) {
 // 如果当前元素小于等于剩余需要凑成的目标值, 更新 dp[i]
 if (num <= i) {
 dp[i] += dp[i - num];
 // 防止溢出 (题目保证结果在 32 位有符号整数范围内)
 if (dp[i] > Integer.MAX_VALUE) {
 dp[i] = Integer.MAX_VALUE;
 }
 }
 }
 }

 return (int) dp[target];
}

/**
 * 递归+记忆化搜索实现
 * @param nums 不同整数组成的数组
 * @param target 目标整数
 * @return 总和为 target 的元素组合的个数

```

```

*/
public static int combinationSum4DFS(int[] nums, int target) {
 // 参数验证
 if (target < 0) {
 return 0;
 }
 if (target == 0) {
 return 1;
 }
 if (nums == null || nums.length == 0) {
 return 0;
 }

 // 使用 HashMap 作为记忆化缓存
 Map<Integer, Integer> memo = new HashMap<>();

 return dfs(nums, target, memo);
}

/**
 * 递归辅助函数
 * @param nums 不同整数组成的数组
 * @param remain 剩余需要凑成的目标值
 * @param memo 记忆化缓存，键是剩余目标值，值是对应的排列数
 * @return 凑成剩余目标值的不同排列数
 */
private static int dfs(int[] nums, int remain, Map<Integer, Integer> memo) {
 // 基础情况：如果剩余目标值为 0，说明找到了一种排列
 if (remain == 0) {
 return 1;
 }

 // 检查缓存
 if (memo.containsKey(remain)) {
 return memo.get(remain);
 }

 int ways = 0;

 // 尝试使用每个元素
 for (int num : nums) {
 if (num <= remain) {
 // 递归计算剩余值的排列数

```

```

 ways += dfs(nums, remain - num, memo);
 }
}

// 缓存结果
memo.put(remain, ways);
return ways;
}

/**
 * 优化版本：提前排序和剪枝
 * @param nums 不同整数组成的数组
 * @param target 目标整数
 * @return 总和为 target 的元素组合的个数
 */
public static int combinationSum4Optimized(int[] nums, int target) {
 // 参数验证
 if (target < 0) {
 return 0;
 }
 if (target == 0) {
 return 1;
 }
 if (nums == null || nums.length == 0) {
 return 0;
 }

 // 对数组进行排序，以便在后续处理中进行剪枝
 Arrays.sort(nums);

 // 创建 DP 数组
 long[] dp = new long[target + 1];
 dp[0] = 1;

 // 遍历目标值
 for (int i = 1; i <= target; i++) {
 // 遍历数组中的元素
 for (int num : nums) {
 // 如果当前元素大于剩余需要凑成的目标值，由于数组已排序，后面的元素更大，可以提前
 // 退出循环
 if (num > i) {
 break;
 }
 dp[i] += dp[i - num];
 }
 }
 return (int) dp[target];
}

```

```

 dp[i] += dp[i - num];
 // 防止溢出
 if (dp[i] > Integer.MAX_VALUE) {
 dp[i] = Integer.MAX_VALUE;
 }
 }

 return (int) dp[target];
}

/***
 * 递归+记忆化搜索实现的另一种方式，使用数组作为缓存
 * @param nums 不同整数组成的数组
 * @param target 目标整数
 * @return 总和为 target 的元素组合的个数
 */
public static int combinationSum4DFSArray(int[] nums, int target) {
 // 参数验证
 if (target < 0) {
 return 0;
 }
 if (target == 0) {
 return 1;
 }
 if (nums == null || nums.length == 0) {
 return 0;
 }

 // 使用数组作为缓存，初始值为-1 表示未计算
 int[] memo = new int[target + 1];
 Arrays.fill(memo, -1);
 memo[0] = 1; // 溉成目标值 0 只有一种方式

 return dfsArray(nums, target, memo);
}

/***
 * 递归辅助函数 - 使用数组作为缓存
 * @param nums 不同整数组成的数组
 * @param remain 剩余需要溉成的目标值
 * @param memo 记忆化缓存数组
 * @return 溉成剩余目标值的不同排列数
 */

```

```

*/
private static int dfsArray(int[] nums, int remain, int[] memo) {
 // 基础情况：如果剩余目标值为 0，说明找到了一种排列
 if (remain == 0) {
 return 1;
 }

 // 检查缓存
 if (memo[remain] != -1) {
 return memo[remain];
 }

 int ways = 0;

 // 尝试使用每个元素
 for (int num : nums) {
 if (num <= remain) {
 ways += dfsArray(nums, remain - num, memo);
 }
 }

 // 缓存结果
 memo[remain] = ways;
 return ways;
}

```

=====

文件: Code33\_CombinationSumIV.py

=====

```

LeetCode 377. 组合总和 IV
题目描述：给你一个由 不同 整数组成的数组 nums ，和一个目标整数 target 。
请你从 nums 中找出并返回总和为 target 的元素组合的个数。
注意：顺序不同的序列被视作不同的组合。
链接: https://leetcode.cn/problems/combination-sum-iv/
#
解题思路：
// 这是一个与完全背包相关但关注排列而非组合的问题：
// - 数字可以多次使用（完全背包的特点）
// - 顺序不同的序列视为不同的组合（与组合数问题的关键区别）
//
// 状态定义：dp[i] 表示凑成目标值 i 的不同排列数

```

```
// 状态转移方程: dp[i] += dp[i - num], 其中 num 是 nums 中的每个元素, 且 i >= num
// 初始状态: dp[0] = 1 (凑成目标值 0 只有一种方式: 不选择任何数字)
//
// 时间复杂度: O(target * n), 其中 n 是数组 nums 的长度
// 空间复杂度: O(target), 使用一维 DP 数组

from typing import List
from functools import lru_cache

def combinationSum4(nums: List[int], target: int) -> int:
 """
 计算凑成目标值的不同排列数

 Args:
 nums: 不同整数组成的数组
 target: 目标整数

 Returns:
 总和为 target 的元素组合的个数
 """

 # 参数验证
 if target < 0:
 return 0
 if target == 0:
 return 1
 if not nums:
 return 0

 # 创建 DP 数组, dp[i] 表示凑成目标值 i 的不同排列数
 dp = [0] * (target + 1)
 dp[0] = 1 # 凑成目标值 0 只有一种方式: 不选择任何数字

 # 遍历目标值, 从 1 到 target
 # 注意: 与零钱兑换 II 不同, 这里我们先遍历目标值, 再遍历数组元素, 这样可以考虑不同顺序的排列
 for i in range(1, target + 1):
 # 遍历数组中的每个元素
 for num in nums:
 # 如果当前元素小于等于剩余需要凑成的目标值, 更新 dp[i]
 if num <= i:
 dp[i] += dp[i - num]

 return dp[target]
```

```
def combinationSum4DFS(nums: List[int], target: int) -> int:
 """
 递归+记忆化搜索实现
 """

 # 参数验证
 if target < 0:
 return 0
 if target == 0:
 return 1
 if not nums:
 return 0

 @lru_cache(maxsize=None)
 def dfs(remain: int) -> int:
 """
 递归辅助函数

 Args:
 remain: 剩余需要凑成的目标值

 Returns:
 凑成剩余目标值的不同排列数
 """

 # 基础情况: 如果剩余目标值为 0, 说明找到了一种排列
 if remain == 0:
 return 1

 ways = 0

 # 尝试使用每个元素
 for num in nums:
 if num <= remain:
 # 递归计算剩余值的排列数
 ways += dfs(remain - num)

 return ways

 return dfs(target)
```

```
def combinationSum4optimized(nums: List[int], target: int) -> int:
```

```

"""
优化版本：提前排序和剪枝
"""

参数验证
if target < 0:
 return 0
if target == 0:
 return 1
if not nums:
 return 0

对数组进行排序，以便在后续处理中进行剪枝
nums.sort()

创建 DP 数组
dp = [0] * (target + 1)
dp[0] = 1

遍历目标值
for i in range(1, target + 1):
 # 遍历数组中的元素
 for num in nums:
 # 如果当前元素大于剩余需要凑成的目标值，由于数组已排序，后面的元素更大，可以提前退出
循环
 if num > i:
 break
 dp[i] += dp[i - num]

return dp[target]

def combinationSum4MemoArray(nums: List[int], target: int) -> int:
"""
递归+数组缓存实现
"""

参数验证
if target < 0:
 return 0
if target == 0:
 return 1
if not nums:
 return 0

```

```

使用数组作为缓存，初始值为-1 表示未计算
memo = [-1] * (target + 1)
memo[0] = 1 # 溜成目标值 0 只有一种方式

def dfs(remain: int) -> int:
 """
 递归辅助函数

 Args:
 remain: 剩余需要溜成的目标值

 Returns:
 溜成剩余目标值的不同排列数
 """
 # 基础情况：如果剩余目标值为 0，说明找到了一种排列
 if remain == 0:
 return 1

 # 检查缓存
 if memo[remain] != -1:
 return memo[remain]

 ways = 0

 # 尝试使用每个元素
 for num in nums:
 if num <= remain:
 ways += dfs(remain - num)

 # 缓存结果
 memo[remain] = ways
 return ways

return dfs(target)

测试函数
def test_combination_sum4():
 # 测试用例 1
 nums1 = [1, 2, 3]
 target1 = 4
 print(f"测试用例 1 结果: {combinationSum4(nums1, target1)}") # 预期输出: 7 ([1, 1, 1, 1], [1, 1, 2], [1, 2, 1], [1, 3], [2, 1, 1], [2, 2], [3, 1])

```

```

测试用例 2
nums2 = [9]
target2 = 3
print(f"测试用例 2 结果: {combinationSum4(nums2, target2)}") # 预期输出: 0

测试 DFS 实现
print(f"测试用例 1 (DFS): {combinationSum4DFS(nums1, target1)}")

测试优化版本
print(f"测试用例 1 (优化版本): {combinationSum4Optimized(nums1, target1)}")

测试数组缓存实现
print(f"测试用例 1 (数组缓存): {combinationSum4MemoArray(nums1, target1)}")

执行测试
if __name__ == "__main__":
 test_combination_sum4()

```

=====

文件: Code34\_NumberOfDiceRollsWithTargetSum.cpp

=====

```

#include <iostream>
#include <vector>
#include <functional>

// LeetCode 1155. 掷骰子的 N 种方法
// 题目描述: 这里有 n 个一样的骰子，每个骰子上都有 k 个面，分别标有 1 到 k 的数字。
// 给定三个整数 n, k 和 target，请你计算并返回投掷骰子的所有可能得到的结果等于 target 的方案数。
// 答案可能很大，所以需要返回模 10^9 + 7 的结果。
// 链接: https://leetcode.cn/problems/number-of-dice-rolls-with-target-sum/
//
// 解题思路:
// 这是一个典型的分组背包问题，每个骰子可以看作一组，每组有 k 个选项（1 到 k 的点数）
// 我们需要从每组中选择一个选项，使得它们的总和等于 target，求总共有多少种选法。
//
// 状态定义: dp[i][j] 表示使用 i 个骰子能得到点数和为 j 的方案数
// 状态转移方程: dp[i][j] = sum(dp[i-1][j-m]), 其中 m 从 1 到 k 且 j-m >= i-1 (因为每个骰子至少为 1,
i-1 个骰子至少为 i-1)
// 初始状态: dp[0][0] = 1 (使用 0 个骰子得到点数和为 0 只有一种方式)
//

```

```

// 时间复杂度: O(n * k * target)
// 空间复杂度: O(n * target), 可以优化为 O(target)

using namespace std;

// 模数
const int MOD = 1000000007;

/***
 * 计算投掷骰子得到目标点数和的方案数
 * @param n 骰子数量
 * @param k 每个骰子的面数 (1 到 k)
 * @param target 目标点数和
 * @return 方案数模 10^9+7 的结果
*/
int numRollsToTarget(int n, int k, int target) {
 // 参数验证
 if (n < 1 || k < 1 || target < n || target > n * k) {
 return 0; // 不可能的情况: target 小于骰子数或大于骰子数*最大面数
 }

 // 创建二维 DP 数组, dp[i][j] 表示使用 i 个骰子能得到点数和为 j 的方案数
 vector<vector<int>> dp(n + 1, vector<int>(target + 1, 0));

 // 初始状态: 使用 0 个骰子得到点数和为 0 只有一种方式
 dp[0][0] = 1;

 // 填充 DP 数组
 for (int i = 1; i <= n; i++) { // 遍历骰子数量
 for (int j = i; j <= min(target, i * k); j++) { // 遍历可能的点数和 (至少 i, 最多 i*k)
 for (int m = 1; m <= k && m <= j; m++) { // 遍历当前骰子的可能点数
 dp[i][j] = (dp[i][j] + dp[i-1][j-m]) % MOD;
 }
 }
 }

 return dp[n][target];
}

/***
 * 优化空间复杂度的版本, 使用一维 DP 数组
 * @param n 骰子数量
 * @param k 每个骰子的面数 (1 到 k)
*/

```

```

* @param target 目标点数和
* @return 方案数模 10^9+7 的结果
*/
int numRollsToTargetOptimized(int n, int k, int target) {
 // 参数验证
 if (n < 1 || k < 1 || target < n || target > n * k) {
 return 0;
 }

 // 创建一维 DP 数组, dp[j] 表示使用当前骰子数能得到点数和为 j 的方案数
 vector<int> dp(target + 1, 0);

 // 初始状态: 使用 0 个骰子得到点数和为 0 只有一种方式
 dp[0] = 1;

 // 遍历骰子数量
 for (int i = 1; i <= n; i++) {
 // 创建一个新数组来保存当前轮次的结果
 vector<int> newDp(target + 1, 0);

 // 遍历可能的点数和
 for (int j = i; j <= min(target, i * k); j++) {
 // 遍历当前骰子的可能点数
 for (int m = 1; m <= k && m <= j; m++) {
 newDp[j] = (newDp[j] + dp[j - m]) % MOD;
 }
 }
 }

 // 更新 dp 数组为当前轮次的结果
 dp = move(newDp);
}

return dp[target];
}

/***
* 另一种空间优化的方式, 只使用一个一维数组, 并倒序更新
* 这种方式不适用, 因为我们需要严格区分不同骰子数的状态
* 所以这里只是作为对比展示, 不推荐使用
*/
int numRollsToTargetAlternative(int n, int k, int target) {
 // 参数验证
 if (n < 1 || k < 1 || target < n || target > n * k) {

```

```

 return 0;
 }

// 创建 DP 数组
vector<int> dp(target + 1, 0);
dp[0] = 1;

// 遍历骰子数量
for (int i = 1; i <= n; i++) {
 // 创建临时数组来保存前一轮的结果
 vector<int> prevDp = dp;

 // 重置当前轮次的结果数组（除了 dp[0]）
 for (int j = 1; j <= target; j++) {
 dp[j] = 0;
 }

 // 更新当前轮次的结果
 for (int j = 1; j <= target; j++) {
 for (int m = 1; m <= k && m <= j; m++) {
 dp[j] = (dp[j] + prevDp[j - m]) % MOD;
 }
 }
}

return dp[target];
}

/***
 * 递归+记忆化搜索实现
 * @param n 骰子数量
 * @param k 每个骰子的面数 (1 到 k)
 * @param target 目标点数和
 * @return 方案数模 10^9+7 的结果
 */
int numRollsToTargetDFS(int n, int k, int target) {
 // 参数验证
 if (n < 1 || k < 1 || target < n || target > n * k) {
 return 0;
 }

 // 创建记忆化缓存, dp[i][j] 表示使用 i 个骰子能得到点数和为 j 的方案数
 vector<vector<int>> memo(n + 1, vector<int>(target + 1, -1));

```

```

function<int(int, int)> dfs = [&](int remainingDice, int remainingTarget) -> int {
 // 基础情况：如果没有骰子了，检查是否达成目标
 if (remainingDice == 0) {
 return remainingTarget == 0 ? 1 : 0;
 }

 // 检查缓存
 if (memo[remainingDice][remainingTarget] != -1) {
 return memo[remainingDice][remainingTarget];
 }

 int ways = 0;

 // 尝试当前骰子的所有可能点数
 for (int i = 1; i <= k; i++) {
 // 只有当前点数不超过剩余目标，并且剩余的骰子可以凑成剩余的点数时，才继续递归
 if (i <= remainingTarget && (remainingDice - 1) <= (remainingTarget - i) &&
(remainingTarget - i) <= (remainingDice - 1) * k) {
 ways = (ways + dfs(remainingDice - 1, remainingTarget - i)) % MOD;
 }
 }

 // 缓存结果
 return memo[remainingDice][remainingTarget] = ways;
};

return dfs(n, target);
}

int main() {
 // 测试用例 1
 int n1 = 1, k1 = 6, target1 = 3;
 cout << "测试用例 1 结果：" << numRollsToTarget(n1, k1, target1) << endl; // 预期输出: 1

 // 测试用例 2
 int n2 = 2, k2 = 6, target2 = 7;
 cout << "测试用例 2 结果：" << numRollsToTarget(n2, k2, target2) << endl; // 预期输出: 6

 // 测试用例 3
 int n3 = 30, k3 = 30, target3 = 500;
 cout << "测试用例 3 结果：" << numRollsToTarget(n3, k3, target3) << endl; // 预期输出:
222616187
}

```

```

// 测试优化版本
cout << "测试用例 2 (优化版本): " << numRollsToTargetOptimized(n2, k2, target2) << endl;

// 测试 DFS 实现
cout << "测试用例 2 (DFS): " << numRollsToTargetDFS(n2, k2, target2) << endl;

return 0;
}
=====
```

文件: Code34\_NumberOfDiceRollsWithTargetSum. java

```

package class073;

// LeetCode 1155. 掷骰子的 N 种方法
// 题目描述: 这里有 n 个一样的骰子，每个骰子上都有 k 个面，分别标有 1 到 k 的数字。
// 给定三个整数 n, k 和 target，请你计算并返回投掷骰子的所有可能得到的结果等于 target 的方案数。
// 答案可能很大，所以需要返回模 $10^9 + 7$ 的结果。
// 链接: https://leetcode.cn/problems/number-of-dice-rolls-with-target-sum/
//
// 解题思路:
// 这是一个典型的分组背包问题，每个骰子可以看作一组，每组有 k 个选项（1 到 k 的点数）
// 我们需要从每组中选择一个选项，使得它们的总和等于 target，求总共有多少种选法。
//
// 状态定义: dp[i][j] 表示使用 i 个骰子能得到点数和为 j 的方案数
// 状态转移方程: dp[i][j] = sum(dp[i-1][j-m])，其中 m 从 1 到 k 且 $j-m \geq i-1$ （因为每个骰子至少为 1，i-1 个骰子至少为 i-1）
// 初始状态: dp[0][0] = 1 (使用 0 个骰子得到点数和为 0 只有一种方式)
//
// 时间复杂度: O(n * k * target)
// 空间复杂度: O(n * target)，可以优化为 O(target)
```

```
public class Code34_NumberOfDiceRollsWithTargetSum {
```

```

// 主方法，用于测试
public static void main(String[] args) {
 // 测试用例 1
 int n1 = 1, k1 = 6, target1 = 3;
 System.out.println("测试用例 1 结果: " + numRollsToTarget(n1, k1, target1)); // 预期输出:
```

```

// 测试用例 2
int n2 = 2, k2 = 6, target2 = 7;
System.out.println("测试用例 2 结果: " + numRollsToTarget(n2, k2, target2)); // 预期输出:
6

// 测试用例 3
int n3 = 30, k3 = 30, target3 = 500;
System.out.println("测试用例 3 结果: " + numRollsToTarget(n3, k3, target3)); // 预期输出:
222616187
}

// 模数
private static final int MOD = 1000000007;

/**
 * 计算投掷骰子得到目标点数和的方案数
 * @param n 骰子数量
 * @param k 每个骰子的面数 (1 到 k)
 * @param target 目标点数和
 * @return 方案数模 10^9+7 的结果
 */
public static int numRollsToTarget(int n, int k, int target) {
 // 参数验证
 if (n < 1 || k < 1 || target < n || target > n * k) {
 return 0; // 不可能的情况: target 小于骰子数或大于骰子数*最大面数
 }

 // 创建二维 DP 数组, dp[i][j] 表示使用 i 个骰子能得到点数和为 j 的方案数
 int[][] dp = new int[n + 1][target + 1];

 // 初始状态: 使用 0 个骰子得到点数和为 0 只有一种方式
 dp[0][0] = 1;

 // 填充 DP 数组
 for (int i = 1; i <= n; i++) { // 遍历骰子数量
 for (int j = i; j <= Math.min(target, i * k); j++) { // 遍历可能的点数和 (至少 i, 最多 i*k)
 for (int m = 1; m <= k && m <= j; m++) { // 遍历当前骰子的可能点数
 dp[i][j] = (dp[i][j] + dp[i-1][j-m]) % MOD;
 }
 }
 }
}

```

```

 return dp[n][target];
 }

/***
 * 优化空间复杂度的版本，使用一维 DP 数组
 * @param n 骰子数量
 * @param k 每个骰子的面数（1 到 k）
 * @param target 目标点数和
 * @return 方案数模 10^9+7 的结果
 */
public static int numRollsToTargetOptimized(int n, int k, int target) {
 // 参数验证
 if (n < 1 || k < 1 || target < n || target > n * k) {
 return 0;
 }

 // 创建一维 DP 数组，dp[j] 表示使用当前骰子数能得到点数和为 j 的方案数
 int[] dp = new int[target + 1];

 // 初始状态：使用 0 个骰子得到点数和为 0 只有一种方式
 dp[0] = 1;

 // 遍历骰子数量
 for (int i = 1; i <= n; i++) {
 // 创建一个新数组来保存当前轮次的结果
 int[] newDp = new int[target + 1];

 // 遍历可能的点数和
 for (int j = i; j <= Math.min(target, i * k); j++) {
 // 遍历当前骰子的可能点数
 for (int m = 1; m <= k && m <= j; m++) {
 newDp[j] = (newDp[j] + dp[j - m]) % MOD;
 }
 }

 // 更新 dp 数组为当前轮次的结果
 dp = newDp;
 }

 return dp[target];
}
*/

```

```

* 另一种空间优化的方式，只使用一个一维数组，并倒序更新
* 这种方式不适用，因为我们需要严格区分不同骰子数的状态
* 所以这里只是作为对比展示，不推荐使用
*/
public static int numRollsToTargetAlternative(int n, int k, int target) {
 // 参数验证
 if (n < 1 || k < 1 || target < n || target > n * k) {
 return 0;
 }

 // 创建 DP 数组
 int[] dp = new int[target + 1];
 dp[0] = 1;

 // 遍历骰子数量
 for (int i = 1; i <= n; i++) {
 // 创建临时数组来保存前一轮的结果
 int[] prevDp = new int[target + 1];
 for (int j = 0; j <= target; j++) {
 prevDp[j] = dp[j];
 }

 // 重置当前轮次的结果数组（除了 dp[0]）
 for (int j = 1; j <= target; j++) {
 dp[j] = 0;
 }

 // 更新当前轮次的结果
 for (int j = 1; j <= target; j++) {
 for (int m = 1; m <= k && m <= j; m++) {
 dp[j] = (dp[j] + prevDp[j - m]) % MOD;
 }
 }
 }

 return dp[target];
}

/**
 * 递归+记忆化搜索实现
 * @param n 骰子数量
 * @param k 每个骰子的面数 (1 到 k)
 * @param target 目标点数和

```

```

* @return 方案数模 10^9+7 的结果
*/
public static int numRollsToTargetDFS(int n, int k, int target) {
 // 参数验证
 if (n < 1 || k < 1 || target < n || target > n * k) {
 return 0;
 }

 // 创建记忆化缓存, dp[i][j]表示使用 i 个骰子能得到点数和为 j 的方案数
 // 使用二维数组作为缓存
 Integer[][] memo = new Integer[n + 1][target + 1];

 return dfs(n, k, target, memo);
}

/***
 * 递归辅助函数
 * @param remainingDice 剩余骰子数量
 * @param k 每个骰子的面数 (1 到 k)
 * @param remainingTarget 剩余目标点数
 * @param memo 记忆化缓存
 * @return 方案数模 10^9+7 的结果
*/
private static int dfs(int remainingDice, int k, int remainingTarget, Integer[][] memo) {
 // 基础情况: 如果没有骰子了, 检查是否达成目标
 if (remainingDice == 0) {
 return remainingTarget == 0 ? 1 : 0;
 }

 // 检查缓存
 if (memo[remainingDice][remainingTarget] != null) {
 return memo[remainingDice][remainingTarget];
 }

 int ways = 0;

 // 尝试当前骰子的所有可能点数
 for (int i = 1; i <= k; i++) {
 // 只有当前点数不超过剩余目标, 并且剩余的骰子可以凑成剩余的点数时, 才继续递归
 if (i <= remainingTarget && (remainingDice - 1) <= (remainingTarget - i) &&
(remainingTarget - i) <= (remainingDice - 1) * k) {
 ways = (ways + dfs(remainingDice - 1, k, remainingTarget - i, memo)) % MOD;
 }
 }
}

```

```

 }

 // 缓存结果
 memo[remainingDice][remainingTarget] = ways;
 return ways;
}

}
=====
```

文件: Code34\_NumberOfDiceRollsWithTargetSum.py

```

LeetCode 1155. 掷骰子的 N 种方法
题目描述: 这里有 n 个一样的骰子，每个骰子上都有 k 个面，分别标有 1 到 k 的数字。
给定三个整数 n, k 和 target，请你计算并返回投掷骰子的所有可能得到的结果等于 target 的方案数。
答案可能很大，所以需要返回模 10^9 + 7 的结果。
链接: https://leetcode.cn/problems/number-of-dice-rolls-with-target-sum/
#
解题思路:
这是一个典型的分组背包问题，每个骰子可以看作一组，每组有 k 个选项（1 到 k 的点数）
我们需要从每组中选择一个选项，使得它们的总和等于 target，求总共有多少种选法。
#
状态定义: dp[i][j] 表示使用 i 个骰子能得到点数和为 j 的方案数
状态转移方程: dp[i][j] = sum(dp[i-1][j-m]), 其中 m 从 1 到 k 且 j-m >= i-1 (因为每个骰子至少为 1,
i-1 个骰子至少为 i-1)
初始状态: dp[0][0] = 1 (使用 0 个骰子得到点数和为 0 只有一种方式)
#
时间复杂度: O(n * k * target)
空间复杂度: O(n * target)，可以优化为 O(target)
```

MOD = 10\*\*9 + 7

```
def num_rolls_to_target(n: int, k: int, target: int) -> int:
 """
```

计算投掷骰子得到目标点数和的方案数

参数:

- n: 骰子数量
- k: 每个骰子的面数（1 到 k）
- target: 目标点数和

返回:

方案数模 10^9+7 的结果

```

"""
参数验证
if n < 1 or k < 1 or target < n or target > n * k:
 return 0 # 不可能的情况: target 小于骰子数或大于骰子数*最大面数

创建二维 DP 数组, dp[i][j] 表示使用 i 个骰子能得到点数和为 j 的方案数
dp = [[0] * (target + 1) for _ in range(n + 1)]

初始状态: 使用 0 个骰子得到点数和为 0 只有一种方式
dp[0][0] = 1

填充 DP 数组
for i in range(1, n + 1): # 遍历骰子数量
 for j in range(i, min(target, i * k) + 1): # 遍历可能的点数和 (至少 i, 最多 i*k)
 for m in range(1, k + 1): # 遍历当前骰子的可能点数
 if m > j: # 当前点数超过目标, 停止循环
 break
 dp[i][j] = (dp[i][j] + dp[i-1][j-m]) % MOD

return dp[n][target]

```

```

def num_rolls_to_target_optimized(n: int, k: int, target: int) -> int:
"""

```

优化空间复杂度的版本, 使用一维 DP 数组

参数:

- n: 骰子数量
- k: 每个骰子的面数 (1 到 k)
- target: 目标点数和

返回:

方案数模  $10^{9+7}$  的结果

```

"""

```

# 参数验证

```

if n < 1 or k < 1 or target < n or target > n * k:
 return 0

```

```

创建一维 DP 数组, dp[j] 表示使用当前骰子数能得到点数和为 j 的方案数
dp = [0] * (target + 1)

```

```

初始状态: 使用 0 个骰子得到点数和为 0 只有一种方式
dp[0] = 1

```

```

遍历骰子数量
for i in range(1, n + 1):
 # 创建一个新数组来保存当前轮次的结果
 new_dp = [0] * (target + 1)

 # 遍历可能的点数和
 for j in range(i, min(target, i * k) + 1):
 # 遍历当前骰子的可能点数
 for m in range(1, k + 1):
 if m > j: # 当前点数超过目标，停止循环
 break
 new_dp[j] = (new_dp[j] + dp[j - m]) % MOD

 # 更新 dp 数组为当前轮次的结果
 dp = new_dp

return dp[target]

```

```

def num_rolls_to_target_alternative(n: int, k: int, target: int) -> int:
 """

```

另一种空间优化的方式，只使用一个一维数组，并倒序更新  
 这种方式不适用，因为我们需要严格区分不同骰子数的状态  
 所以这里只是作为对比展示，不推荐使用

参数：

n: 骰子数量  
 k: 每个骰子的面数（1 到 k）  
 target: 目标点数和

返回：

方案数模  $10^{9+7}$  的结果

```

 """
参数验证
if n < 1 or k < 1 or target < n or target > n * k:
 return 0

```

# 创建 DP 数组

```

dp = [0] * (target + 1)
dp[0] = 1

```

# 遍历骰子数量

```

for i in range(1, n + 1):
 # 创建临时数组来保存前一轮的结果

```

```

prev_dp = dp.copy()

重置当前轮次的结果数组（除了 dp[0]）
for j in range(1, target + 1):
 dp[j] = 0

更新当前轮次的结果
for j in range(1, target + 1):
 for m in range(1, k + 1):
 if m > j:
 break
 dp[j] = (dp[j] + prev_dp[j - m]) % MOD

return dp[target]

```

def num\_rolls\_to\_target\_dfs(n: int, k: int, target: int) -> int:

"""

递归+记忆化搜索实现

参数:

n: 骰子数量

k: 每个骰子的面数（1 到 k）

target: 目标点数和

返回:

方案数模  $10^{9+7}$  的结果

"""

# 参数验证

```

if n < 1 or k < 1 or target < n or target > n * k:
 return 0

```

# 创建记忆化缓存, dp[i][j] 表示使用 i 个骰子能得到点数和为 j 的方案数

# 使用二维列表作为缓存

```

memo = [[-1] * (target + 1) for _ in range(n + 1)]

```

def dfs(remaining\_dice: int, remaining\_target: int) -> int:

"""

递归辅助函数

参数:

remaining\_dice: 剩余骰子数量

remaining\_target: 剩余目标点数

返回：

方案数模  $10^{9+7}$  的结果

"""

# 基础情况：如果没有骰子了，检查是否达成目标

if remaining\_dice == 0:

    return 1 if remaining\_target == 0 else 0

# 检查缓存

if memo[remaining\_dice][remaining\_target] != -1:

    return memo[remaining\_dice][remaining\_target]

ways = 0

# 尝试当前骰子的所有可能点数

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

    # 只有当前点数不超过剩余目标，并且剩余的骰子可以凑成剩余的点数时，才继续递归

    if (i <= remaining\_target and

        (remaining\_dice - 1) <= (remaining\_target - i) and

        (remaining\_target - i) <= (remaining\_dice - 1) \* k):

        ways = (ways + dfs(remaining\_dice - 1, remaining\_target - i)) % MOD

# 缓存结果

memo[remaining\_dice][remaining\_target] = ways

return ways

return dfs(n, target)

# 测试代码

if \_\_name\_\_ == "\_\_main\_\_":

# 测试用例 1

n1, k1, target1 = 1, 6, 3

print(f"测试用例 1 结果: {num\_rolls\_to\_target(n1, k1, target1)}") # 预期输出: 1

# 测试用例 2

n2, k2, target2 = 2, 6, 7

print(f"测试用例 2 结果: {num\_rolls\_to\_target(n2, k2, target2)}") # 预期输出: 6

# 测试用例 3

n3, k3, target3 = 30, 30, 500

print(f"测试用例 3 结果: {num\_rolls\_to\_target(n3, k3, target3)}") # 预期输出: 222616187

# 测试优化版本

print(f"测试用例 2 (优化版本): {num\_rolls\_to\_target\_optimized(n2, k2, target2)}")

```
测试 DFS 实现
print(f"测试用例 2 (DFS): {num_rolls_to_target_dfs(n2, k2, target2)}")
```

```
=====
```

文件: Code35\_LastStoneWeightII.cpp

```
=====
```

```
#include <iostream>
#include <vector>
#include <algorithm>

// LeetCode 1049. 最后一块石头的重量 II
// 题目描述: 有一堆石头, 每块石头的重量都是正整数。
// 每一回合, 从中选出任意两块石头, 然后将它们一起粉碎。假设石头的重量分别为 x 和 y, 且 x <= y。那么粉碎的可能结果如下:
// 如果 x == y, 那么两块石头都会被完全粉碎;
// 如果 x != y, 那么重量为 x 的石头将会完全粉碎, 而重量为 y 的石头新重量为 y-x。
// 最后, 最多只会剩下一块石头。返回此石头的最小可能重量。如果没有石头剩下, 就返回 0。
// 链接: https://leetcode.cn/problems/last-stone-weight-ii/
//

// 解题思路:
// 这是一个经典的 0-1 背包问题的变种。我们的目标是将石头分成两组, 使得两组的重量差最小。
// 设石头总重量为 sum, 我们希望找到一个子集, 其总重量尽可能接近 sum/2。
// 这样, 两组的重量差就是 sum - 2 * subsetSum, 我们需要最小化这个值。
//

// 状态定义: dp[j] 表示是否能组成重量为 j 的子集
// 状态转移方程: dp[j] = dp[j] || dp[j - stones[i]]
// 初始状态: dp[0] = true (空子集的重量为 0)
//

// 时间复杂度: O(n * target), 其中 n 是石头数量, target 是 sum/2
// 空间复杂度: O(target), 使用一维 DP 数组
```

```
using namespace std;
```

```
/***
 * 计算最后一块石头的最小可能重量
 * @param stones 石头的重量数组
 * @return 最后一块石头的最小可能重量
 */
```

```
int lastStoneWeightII(vector<int>& stones) {
 if (stones.empty()) {
 return 0;
```

```

}

// 计算石头总重量
int sum = 0;
for (int stone : stones) {
 sum += stone;
}

// 目标是找到尽可能接近 sum/2 的子集和
int target = sum / 2;

// 创建 DP 数组, dp[j] 表示是否能组成重量为 j 的子集
vector<bool> dp(target + 1, false);

// 初始状态: 空子集的重量为 0 是可以组成的
dp[0] = true;

// 遍历每一块石头
for (int stone : stones) {
 // 逆序遍历重量, 避免重复使用同一块石头
 for (int j = target; j >= stone; j--) {
 // 如果 j-stone 可以组成, 那么 j 也可以组成
 dp[j] = dp[j] || dp[j - stone];
 }
}

// 找到最大的 j, 使得 dp[j] 为 true, 且 j <= target
int maxSubsetSum = 0;
for (int j = target; j >= 0; j--) {
 if (dp[j]) {
 maxSubsetSum = j;
 break;
 }
}

// 两组的重量差就是 sum - 2 * maxSubsetSum
return sum - 2 * maxSubsetSum;
}

/***
 * 使用二维 DP 数组的版本
 * @param stones 石头的重量数组
 * @return 最后一块石头的最小可能重量
*/

```

```

*/
int lastStoneWeightII2D(vector<int>& stones) {
 if (stones.empty()) {
 return 0;
 }

 // 计算石头总重量
 int sum = 0;
 for (int stone : stones) {
 sum += stone;
 }

 int target = sum / 2;
 int n = stones.size();

 // 创建二维 DP 数组, dp[i][j] 表示前 i 个石头是否能组成重量为 j 的子集
 vector<vector<bool>> dp(n + 1, vector<bool>(target + 1, false));

 // 初始状态: 空子集的重量为 0 是可以组成的
 for (int i = 0; i <= n; i++) {
 dp[i][0] = true;
 }

 // 填充 DP 数组
 for (int i = 1; i <= n; i++) {
 for (int j = 1; j <= target; j++) {
 // 不选第 i 个石头
 dp[i][j] = dp[i-1][j];

 // 选第 i 个石头 (如果 j >= stones[i-1])
 if (j >= stones[i-1]) {
 dp[i][j] = dp[i][j] || dp[i-1][j - stones[i-1]];
 }
 }
 }

 // 找到最大的 j, 使得 dp[n][j] 为 true
 int maxSubsetSum = 0;
 for (int j = target; j >= 0; j--) {
 if (dp[n][j]) {
 maxSubsetSum = j;
 break;
 }
 }
}

```

```

}

return sum - 2 * maxSubsetSum;
}

/***
 * 使用 DP 数组记录可达的重量集合
 * @param stones 石头的重量数组
 * @return 最后一块石头的最小可能重量
*/
int lastStoneWeightIIBitSet(vector<int>& stones) {
 if (stones.empty()) {
 return 0;
 }

 // 使用布尔数组模拟位集合，记录可达的重量
 vector<bool> dp(1501, false); // 根据约束，最大可能的总重量是 30 * 100 = 3000，所以 target 最多
 // 是 1500
 dp[0] = true;

 int sum = 0;

 for (int stone : stones) {
 sum += stone;
 // 逆序更新，避免重复使用同一块石头
 for (int j = min(sum, 1500); j >= stone; j--) {
 dp[j] = dp[j] || dp[j - stone];
 }
 }

 // 寻找最小可能的重量差
 int minWeight = sum;
 for (int j = 0; j <= sum / 2; j++) {
 if (dp[j]) {
 minWeight = min(minWeight, sum - 2 * j);
 }
 }

 return minWeight;
}

/***
 * 递归+记忆化搜索实现

```

```

* 这个方法对于较大的输入可能会超时，但展示了递归的思路
* @param stones 石头的重量数组
* @return 最后一块石头的最小可能重量
*/
int lastStoneWeightIIRecursive(vector<int>& stones) {
 if (stones.empty()) {
 return 0;
 }

 // 计算石头总重量
 int sum = 0;
 for (int stone : stones) {
 sum += stone;
 }

 int target = sum / 2;
 int n = stones.size();

 // 创建记忆化缓存，使用二维数组保存中间结果
 // memo[i][j]表示处理到第 i 个石头时，当前和为 j 的情况下，是否已经访问过
 vector<vector<bool>> visited(n, vector<bool>(target + 1, false));

 // 寻找最大的可能的子集和，使得该和不超过 target
 function<int(int, int)> dfs = [&](int index, int currentSum) -> int {
 // 基础情况：处理完所有石头或当前和已经超过 target
 if (index == n || currentSum > target) {
 return currentSum <= target ? currentSum : 0;
 }

 // 检查是否已经访问过
 if (visited[index][currentSum]) {
 return currentSum;
 }

 // 标记为已访问
 visited[index][currentSum] = true;

 // 选择当前石头
 int takeSum = dfs(index + 1, currentSum + stones[index]);

 // 不选择当前石头
 int notTakeSum = dfs(index + 1, currentSum);

 return max(takeSum, notTakeSum);
 };
}

```

```

 return max(takeSum, notTakeSum);
 };

 int maxSubsetSum = dfs(0, 0);
 return sum - 2 * maxSubsetSum;
}

int main() {
 // 测试用例 1
 vector<int> stones1 = {2, 7, 4, 1, 8, 1};
 cout << "测试用例 1 结果: " << lastStoneWeightII(stones1) << endl; // 预期输出: 1

 // 测试用例 2
 vector<int> stones2 = {31, 26, 33, 21, 40};
 cout << "测试用例 2 结果: " << lastStoneWeightII(stones2) << endl; // 预期输出: 5

 // 测试用例 3
 vector<int> stones3 = {1, 2};
 cout << "测试用例 3 结果: " << lastStoneWeightII(stones3) << endl; // 预期输出: 1

 // 测试二维 DP 版本
 cout << "测试用例 2 (二维 DP): " << lastStoneWeightII2D(stones2) << endl;

 // 测试位集合版本
 cout << "测试用例 2 (位集合): " << lastStoneWeightIIBitSet(stones2) << endl;

 // 测试递归版本
 cout << "测试用例 2 (递归): " << lastStoneWeightIIRecursive(stones2) << endl;

 return 0;
}
=====

文件: Code35_LastStoneWeightII.java
=====

package class073;

// LeetCode 1049. 最后一块石头的重量 II
// 题目描述: 有一堆石头, 每块石头的重量都是正整数。
// 每一回合, 从中选出任意两块石头, 然后将它们一起粉碎。假设石头的重量分别为 x 和 y, 且 x <= y。那么粉碎的可能结果如下:
// 如果 x == y, 那么两块石头都会被完全粉碎;
```

```

// LeetCode 1049. 最后一块石头的重量 II
// 题目描述: 有一堆石头, 每块石头的重量都是正整数。
// 每一回合, 从中选出任意两块石头, 然后将它们一起粉碎。假设石头的重量分别为 x 和 y, 且 x <= y。那么粉碎的可能结果如下:
// 如果 x == y, 那么两块石头都会被完全粉碎;
```

```

// 如果 x != y, 那么重量为 x 的石头将会完全粉碎, 而重量为 y 的石头新重量为 y-x。
// 最后, 最多只会剩下一块石头。返回此石头的最小可能重量。如果没有石头剩下, 就返回 0。
// 链接: https://leetcode.cn/problems/last-stone-weight-ii/
//
// 解题思路:
// 这是一个经典的 0-1 背包问题的变种。我们的目标是将石头分成两组, 使得两组的重量差最小。
// 设石头总重量为 sum, 我们希望找到一个子集, 其总重量尽可能接近 sum/2。
// 这样, 两组的重量差就是 sum - 2 * subsetSum, 我们需要最小化这个值。
//
// 状态定义: dp[j] 表示是否能组成重量为 j 的子集
// 状态转移方程: dp[j] = dp[j] || dp[j - stones[i]]
// 初始状态: dp[0] = true (空子集的重量为 0)
//
// 时间复杂度: O(n * target), 其中 n 是石头数量, target 是 sum/2
// 空间复杂度: O(target), 使用一维 DP 数组

public class Code35_LastStoneWeightII {

 // 主方法, 用于测试
 public static void main(String[] args) {
 // 测试用例 1
 int[] stones1 = {2, 7, 4, 1, 8, 1};
 System.out.println("测试用例 1 结果: " + lastStoneWeightII(stones1)); // 预期输出: 1

 // 测试用例 2
 int[] stones2 = {31, 26, 33, 21, 40};
 System.out.println("测试用例 2 结果: " + lastStoneWeightII(stones2)); // 预期输出: 5

 // 测试用例 3
 int[] stones3 = {1, 2};
 System.out.println("测试用例 3 结果: " + lastStoneWeightII(stones3)); // 预期输出: 1
 }

 /**
 * 计算最后一块石头的最小可能重量
 * @param stones 石头的重量数组
 * @return 最最后一块石头的最小可能重量
 */
 public static int lastStoneWeightII(int[] stones) {
 if (stones == null || stones.length == 0) {
 return 0;
 }
 }
}

```

```

// 计算石头总重量
int sum = 0;
for (int stone : stones) {
 sum += stone;
}

// 目标是找到尽可能接近 sum/2 的子集和
int target = sum / 2;

// 创建 DP 数组, dp[j] 表示是否能组成重量为 j 的子集
boolean[] dp = new boolean[target + 1];

// 初始状态: 空子集的重量为 0 是可以组成的
dp[0] = true;

// 遍历每一块石头
for (int stone : stones) {
 // 逆序遍历重量, 避免重复使用同一块石头
 for (int j = target; j >= stone; j--) {
 // 如果 j-stone 可以组成, 那么 j 也可以组成
 dp[j] = dp[j] || dp[j - stone];
 }
}

// 找到最大的 j, 使得 dp[j] 为 true, 且 j <= target
int maxSubsetSum = 0;
for (int j = target; j >= 0; j--) {
 if (dp[j]) {
 maxSubsetSum = j;
 break;
 }
}

// 两组的重量差就是 sum - 2 * maxSubsetSum
return sum - 2 * maxSubsetSum;
}

/**
 * 使用二维 DP 数组的版本
 * @param stones 石头的重量数组
 * @return 最后一块石头的最小可能重量
 */
public static int lastStoneWeightII2D(int[] stones) {

```

```

if (stones == null || stones.length == 0) {
 return 0;
}

// 计算石头总重量
int sum = 0;
for (int stone : stones) {
 sum += stone;
}

int target = sum / 2;
int n = stones.length;

// 创建二维 DP 数组, dp[i][j] 表示前 i 个石头是否能组成重量为 j 的子集
boolean[][] dp = new boolean[n + 1][target + 1];

// 初始状态: 空子集的重量为 0 是可以组成的
for (int i = 0; i <= n; i++) {
 dp[i][0] = true;
}

// 填充 DP 数组
for (int i = 1; i <= n; i++) {
 for (int j = 1; j <= target; j++) {
 // 不选第 i 个石头
 dp[i][j] = dp[i-1][j];

 // 选第 i 个石头 (如果 j >= stones[i-1])
 if (j >= stones[i-1]) {
 dp[i][j] = dp[i][j] || dp[i-1][j - stones[i-1]];
 }
 }
}

// 找到最大的 j, 使得 dp[n][j] 为 true
int maxSubsetSum = 0;
for (int j = target; j >= 0; j--) {
 if (dp[n][j]) {
 maxSubsetSum = j;
 break;
 }
}

```

```

 return sum - 2 * maxSubsetSum;
 }

/***
 * 使用 DP 数组记录可达的重量集合
 * @param stones 石头的重量数组
 * @return 最后一块石头的最小可能重量
 */
public static int lastStoneWeightIIBitSet(int[] stones) {
 if (stones == null || stones.length == 0) {
 return 0;
 }

 // 使用布尔数组模拟位集合，记录可达的重量
 boolean[] dp = new boolean[1501]; // 根据约束，最大可能的总重量是 30 * 100 = 3000，所以
 target 最多是 1500
 dp[0] = true;

 int sum = 0;

 for (int stone : stones) {
 sum += stone;
 // 逆序更新，避免重复使用同一块石头
 for (int j = Math.min(sum, 1500); j >= stone; j--) {
 dp[j] = dp[j] || dp[j - stone];
 }
 }

 // 寻找最小可能的重量差
 int minWeight = sum;
 for (int j = 0; j <= sum / 2; j++) {
 if (dp[j]) {
 minWeight = Math.min(minWeight, sum - 2 * j);
 }
 }

 return minWeight;
}

/***
 * 递归+记忆化搜索实现
 * 这个方法对于较大的输入可能会超时，但展示了递归的思路
 * @param stones 石头的重量数组
 */

```

```

* @return 最后一块石头的最小可能重量
*/
public static int lastStoneWeightIIRecursive(int[] stones) {
 if (stones == null || stones.length == 0) {
 return 0;
 }

 // 计算石头总重量
 int sum = 0;
 for (int stone : stones) {
 sum += stone;
 }

 // 创建记忆化缓存
 Boolean[][] memo = new Boolean[stones.length][sum + 1];

 // 寻找最大的可能的子集和，使得该和不超过 sum/2
 int maxSubsetSum = findMaxSubsetSum(stones, 0, 0, sum / 2, memo);

 return sum - 2 * maxSubsetSum;
}

/***
 * 递归辅助函数，寻找最大的子集和不超过 target
 * @param stones 石头数组
 * @param index 当前处理的石头索引
 * @param currentSum 当前子集和
 * @param target 目标值 (sum/2)
 * @param memo 记忆化缓存
 * @return 最大的子集和不超过 target
 */
private static int findMaxSubsetSum(int[] stones, int index, int currentSum, int target,
Boolean[][] memo) {
 // 基础情况：处理完所有石头或当前和已经超过 target
 if (index == stones.length || currentSum > target) {
 return currentSum <= target ? currentSum : 0;
 }

 // 检查缓存
 if (memo[index][currentSum] != null) {
 return memo[index][currentSum] ? currentSum : 0;
 }
}

```

```

// 选择当前石头
int takeSum = findMaxSubsetSum(stones, index + 1, currentSum + stones[index], target,
memo);

// 不选择当前石头
int notTakeSum = findMaxSubsetSum(stones, index + 1, currentSum, target, memo);

// 记录结果到缓存
int maxSum = Math.max(takeSum, notTakeSum);
memo[index][currentSum] = (maxSum == currentSum + stones[index]);

return maxSum;
}
}

```

=====

文件: Code35\_LastStoneWeightII.py

=====

```

LeetCode 1049. 最后一块石头的重量 II
题目描述: 有一堆石头, 每块石头的重量都是正整数。
每一回合, 从中选出任意两块石头, 然后将它们一起粉碎。假设石头的重量分别为 x 和 y, 且 x <= y。那么粉碎的可能结果如下:
如果 x == y, 那么两块石头都会被完全粉碎;
如果 x != y, 那么重量为 x 的石头将会完全粉碎, 而重量为 y 的石头新重量为 y-x。
最后, 最多只会剩下一块石头。返回此石头的最小可能重量。如果没有石头剩下, 就返回 0。
链接: https://leetcode.cn/problems/last-stone-weight-ii/
#
解题思路:
这是一个经典的 0-1 背包问题的变种。我们的目标是将石头分成两组, 使得两组的重量差最小。
设石头总重量为 sum, 我们希望找到一个子集, 其总重量尽可能接近 sum/2。
这样, 两组的重量差就是 sum - 2 * subsetSum, 我们需要最小化这个值。
#
状态定义: dp[j] 表示是否能组成重量为 j 的子集
状态转移方程: dp[j] = dp[j] or dp[j - stones[i]]
初始状态: dp[0] = True (空子集的重量为 0)
#
时间复杂度: O(n * target), 其中 n 是石头数量, target 是 sum/2
空间复杂度: O(target), 使用一维 DP 数组

```

```
def last_stone_weight_ii(stones: list[int]) -> int:
```

```
"""

```

计算最后一块石头的最小可能重量

参数:

stones: 石头的重量数组

返回:

最后一块石头的最小可能重量

"""

if not stones:

    return 0

# 计算石头总重量

total\_sum = sum(stones)

# 目标是找到尽可能接近 total\_sum/2 的子集和

target = total\_sum // 2

# 创建 DP 数组, dp[j] 表示是否能组成重量为 j 的子集

dp = [False] \* (target + 1)

# 初始状态: 空子集的重量为 0 是可以组成的

dp[0] = True

# 遍历每一块石头

for stone in stones:

# 逆序遍历重量, 避免重复使用同一块石头

for j in range(target, stone - 1, -1):

# 如果 j-stone 可以组成, 那么 j 也可以组成

dp[j] = dp[j] or dp[j - stone]

# 找到最大的 j, 使得 dp[j] 为 True, 且 j <= target

max\_subset\_sum = 0

for j in range(target, -1, -1):

if dp[j]:

max\_subset\_sum = j

break

# 两组的重量差就是 total\_sum - 2 \* max\_subset\_sum

return total\_sum - 2 \* max\_subset\_sum

def last\_stone\_weight\_ii\_2d(stones: list[int]) -> int:

"""

使用二维 DP 数组的版本

参数:

stones: 石头的重量数组

返回:

最后一块石头的最小可能重量

"""

if not stones:

    return 0

# 计算石头总重量

total\_sum = sum(stones)

target = total\_sum // 2

n = len(stones)

# 创建二维 DP 数组, dp[i][j] 表示前 i 个石头是否能组成重量为 j 的子集

dp = [[False] \* (target + 1) for \_ in range(n + 1)]

# 初始状态: 空子集的重量为 0 是可以组成的

for i in range(n + 1):

    dp[i][0] = True

# 填充 DP 数组

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

    for j in range(1, target + 1):

        # 不选第 i 个石头

        dp[i][j] = dp[i-1][j]

        # 选第 i 个石头 (如果  $j \geq stones[i-1]$ )

        if j >= stones[i-1]:

            dp[i][j] = dp[i][j] or dp[i-1][j - stones[i-1]]

# 找到最大的 j, 使得 dp[n][j] 为 True

max\_subset\_sum = 0

for j in range(target, -1, -1):

    if dp[n][j]:

        max\_subset\_sum = j

        break

return total\_sum - 2 \* max\_subset\_sum

def last\_stone\_weight\_i\_i\_bit\_set(stones: list[int]) -> int:

"""

使用 DP 数组记录可达的重量集合

参数:

stones: 石头的重量数组

返回:

最后一块石头的最小可能重量

"""

if not stones:

    return 0

# 使用布尔数组模拟位集合, 记录可达的重量

dp = [False] \* 1501 # 根据约束, 最大可能的总重量是  $30 * 100 = 3000$ , 所以 target 最多是 1500  
dp[0] = True

total\_sum = 0

for stone in stones:

    total\_sum += stone

    # 逆序更新, 避免重复使用同一块石头

    for j in range(min(total\_sum, 1500), stone - 1, -1):

        dp[j] = dp[j] or dp[j - stone]

# 寻找最小可能的重量差

min\_weight = total\_sum

for j in range(total\_sum // 2 + 1):

    if dp[j]:

        min\_weight = min(min\_weight, total\_sum - 2 \* j)

return min\_weight

from functools import lru\_cache

def last\_stone\_weight\_ii\_recursive(stones: list[int]) -> int:

"""

递归+记忆化搜索实现

这个方法对于较大的输入可能会超时, 但展示了递归的思路

参数:

stones: 石头的重量数组

返回:

最后一块石头的最小可能重量

```

"""
if not stones:
 return 0

计算石头总重量
total_sum = sum(stones)
target = total_sum // 2
n = len(stones)

@lru_cache(maxsize=None)
def dfs(index: int, current_sum: int) -> int:
 """
 递归辅助函数，寻找最大的子集和不超过 target

 参数：
 index: 当前处理的石头索引
 current_sum: 当前子集和

 返回：
 最大的子集和不超过 target
 """
 # 基础情况：处理完所有石头或当前和已经超过 target
 if index == n or current_sum > target:
 return current_sum if current_sum <= target else 0

 # 选择当前石头
 take_sum = dfs(index + 1, current_sum + stones[index])

 # 不选择当前石头
 not_take_sum = dfs(index + 1, current_sum)

 return max(take_sum, not_take_sum)

max_subset_sum = dfs(0, 0)
return total_sum - 2 * max_subset_sum

测试代码
if __name__ == "__main__":
 # 测试用例 1
 stones1 = [2, 7, 4, 1, 8, 1]
 print(f"测试用例 1 结果: {last_stone_weight_ii(stones1)}") # 预期输出: 1

 # 测试用例 2

```

```

stones2 = [31, 26, 33, 21, 40]
print(f"测试用例 2 结果: {last_stone_weight_ii(stones2)}") # 预期输出: 5

测试用例 3
stones3 = [1, 2]
print(f"测试用例 3 结果: {last_stone_weight_ii(stones3)}") # 预期输出: 1

测试二维 DP 版本
print(f"测试用例 2 (二维 DP): {last_stone_weight_ii_2d(stones2)}")

测试位集合版本
print(f"测试用例 2 (位集合): {last_stone_weight_ii_bit_set(stones2)}")

测试递归版本
print(f"测试用例 2 (递归): {last_stone_weight_ii_recursive(stones2)}")

```

=====

文件: Code36\_OnesAndZeroes.cpp

=====

```

// LeetCode 474. 一和零
// 题目描述: 给你一个二进制字符串数组 strs 和两个整数 m 和 n。
// 请你找出并返回 strs 的最大子集的长度，该子集中 最多 有 m 个 0 和 n 个 1。
// 如果 x 的所有元素也是 y 的元素，集合 x 是集合 y 的 子集 。
// 链接: https://leetcode.cn/problems/ones-and-zeroes/
//
// 解题思路:
// 这是一个二维费用的 0-1 背包问题。每个字符串可以看作是一个物品，它有两个费用：0 的数量和 1 的数量。
// 我们的背包有两个容量限制：最多可以使用 m 个 0 和 n 个 1。我们的目标是选择尽可能多的物品（字符串）。
//
// 状态定义: dp[i][j] 表示使用 i 个 0 和 j 个 1 时，最多可以选择的字符串数量
// 状态转移方程: dp[i][j] = max(dp[i][j], dp[i - zeros][j - ones] + 1)，其中 zeros 和 ones 是当前字符串的 0 和 1 的数量
// 初始状态: dp[0][0] = 0，其他初始化为 0
//
// 时间复杂度: O(l * m * n)，其中 l 是字符串数组的长度
// 空间复杂度: O(m * n)，使用二维 DP 数组

```

```

#include <iostream>
#include <vector>
#include <string>
#include <algorithm>

```

```

#include <cstring>
using namespace std;

/***
 * 统计字符串中 0 和 1 的数量
 * @param s 二进制字符串
 * @return 一个 vector, 第一个元素是 0 的数量, 第二个元素是 1 的数量
 */
vector<int> countZeroesOnes(const string& s) {
 vector<int> counts(2, 0);
 for (char c : s) {
 counts[c - '0']++;
 }
 return counts;
}

/***
 * 计算最大子集的长度
 * @param strs 二进制字符串数组
 * @param m 最多可以使用的 0 的数量
 * @param n 最多可以使用的 1 的数量
 * @return 最大子集的长度
 */
int findMaxForm(vector<string>& strs, int m, int n) {
 if (strs.empty()) {
 return 0;
 }

 // 创建二维 DP 数组, dp[i][j] 表示使用 i 个 0 和 j 个 1 时, 最多可以选择的字符串数量
 vector<vector<int>> dp(m + 1, vector<int>(n + 1, 0));

 // 遍历每个字符串
 for (const string& str : strs) {
 // 统计当前字符串中 0 和 1 的数量
 vector<int> counts = countZeroesOnes(str);
 int zeros = counts[0];
 int ones = counts[1];

 // 逆序遍历, 避免重复使用同一个字符串
 for (int i = m; i >= zeros; i--) {
 for (int j = n; j >= ones; j--) {
 // 更新状态: 不选当前字符串 或 选当前字符串 (如果可以的话)
 dp[i][j] = max(dp[i][j], dp[i - zeros][j - ones] + 1);
 }
 }
 }
}

```

```

 }
 }

}

return dp[m][n];
}

/***
 * 使用三维 DP 数组的版本（更直观但空间效率较低）
 * @param strs 二进制字符串数组
 * @param m 最多可以使用的 0 的数量
 * @param n 最多可以使用的 1 的数量
 * @return 最大子集的长度
*/
int findMaxForm3D(vector<string>& strs, int m, int n) {
 if (strs.empty()) {
 return 0;
 }

 int l = strs.size();
 // 创建三维 DP 数组，dp[k][i][j] 表示前 k 个字符串中，使用 i 个 0 和 j 个 1 时，最多可以选择的字符串数量
 vector<vector<vector<int>>> dp(l + 1, vector<vector<int>>(m + 1, vector<int>(n + 1, 0)));

 // 遍历每个字符串
 for (int k = 1; k <= l; k++) {
 string str = strs[k - 1];
 vector<int> counts = countZeroesOnes(str);
 int zeros = counts[0];
 int ones = counts[1];

 // 遍历 0 的数量
 for (int i = 0; i <= m; i++) {
 // 遍历 1 的数量
 for (int j = 0; j <= n; j++) {
 // 不选第 k 个字符串
 dp[k][i][j] = dp[k - 1][i][j];

 // 选第 k 个字符串（如果可以的话）
 if (i >= zeros && j >= ones) {
 dp[k][i][j] = max(dp[k][i][j], dp[k - 1][i - zeros][j - ones] + 1);
 }
 }
 }
 }
}

```

```

 }

}

return dp[1][m][n];
}

/***
 * 优化的二维 DP 版本，将统计 0 和 1 的过程提前
 * @param strs 二进制字符串数组
 * @param m 最多可以使用的 0 的数量
 * @param n 最多可以使用的 1 的数量
 * @return 最大子集的长度
*/
int findMaxFormOptimized(vector<string>& strs, int m, int n) {
 if (strs.empty()) {
 return 0;
 }

 // 预先统计所有字符串中 0 和 1 的数量
 vector<pair<int, int>> counts;
 for (const string& str : strs) {
 vector<int> cnt = countZeroesOnes(str);
 counts.push_back({cnt[0], cnt[1]});
 }

 // 创建二维 DP 数组
 vector<vector<int>> dp(m + 1, vector<int>(n + 1, 0));

 // 遍历每个字符串
 for (const auto& count : counts) {
 int zeros = count.first;
 int ones = count.second;

 // 逆序遍历
 for (int i = m; i >= zeros; i--) {
 for (int j = n; j >= ones; j--) {
 dp[i][j] = max(dp[i][j], dp[i - zeros][j - ones] + 1);
 }
 }
 }

 return dp[m][n];
}

```

```

// 记忆化搜索所需的辅助数组
template<typename T>
void printVector(const vector<T>& v) {
 for (const auto& item : v) {
 cout << item << " ";
 }
 cout << endl;
}

void printMatrix(const vector<vector<int>>& mat) {
 for (const auto& row : mat) {
 printVector(row);
 }
 cout << endl;
}

// 主函数，用于测试
int main() {
 // 测试用例 1
 vector<string> strs1 = {"10", "0001", "111001", "1", "0"};
 int m1 = 5, n1 = 3;
 cout << "测试用例 1 结果: " << findMaxForm(strs1, m1, n1) << endl; // 预期输出: 4
 cout << "三维 DP 版本: " << findMaxForm3D(strs1, m1, n1) << endl;
 cout << "优化版本: " << findMaxFormOptimized(strs1, m1, n1) << endl;

 // 测试用例 2
 vector<string> strs2 = {"10", "0", "1"};
 int m2 = 1, n2 = 1;
 cout << "测试用例 2 结果: " << findMaxForm(strs2, m2, n2) << endl; // 预期输出: 2
 cout << "三维 DP 版本: " << findMaxForm3D(strs2, m2, n2) << endl;
 cout << "优化版本: " << findMaxFormOptimized(strs2, m2, n2) << endl;

 // 测试用例 3
 vector<string> strs3 = {"001", "110", "0000", "0000"};
 int m3 = 9, n3 = 3;
 cout << "测试用例 3 结果: " << findMaxForm(strs3, m3, n3) << endl; // 预期输出: 4

 return 0;
}
=====
```

文件: Code36\_OnesAndZeroes.java

```
=====
package class073;

// LeetCode 474. 一和零
// 题目描述: 给你一个二进制字符串数组 strs 和两个整数 m 和 n。
// 请你找出并返回 strs 的最大子集的长度，该子集中 最多 有 m 个 0 和 n 个 1。
// 如果 x 的所有元素也是 y 的元素，集合 x 是集合 y 的 子集。
// 链接: https://leetcode.cn/problems/ones-and-zeroes/
//
// 解题思路:
// 这是一个二维费用的 0-1 背包问题。每个字符串可以看作是一个物品，它有两个费用：0 的数量和 1 的数量。
// 我们的背包有两个容量限制：最多可以使用 m 个 0 和 n 个 1。我们的目标是选择尽可能多的物品（字符串）。
//
// 状态定义: dp[i][j] 表示使用 i 个 0 和 j 个 1 时，最多可以选择的字符串数量
// 状态转移方程: dp[i][j] = max(dp[i][j], dp[i - zeros][j - ones] + 1)，其中 zeros 和 ones 是当前字符串的 0 和 1 的数量
// 初始状态: dp[0][0] = 0，其他初始化为 0
//
// 时间复杂度: O(l * m * n)，其中 l 是字符串数组的长度
// 空间复杂度: O(m * n)，使用二维 DP 数组

public class Code36_OnesAndZeroes {

 // 主方法，用于测试
 public static void main(String[] args) {
 // 测试用例 1
 String[] strs1 = {"10", "0001", "111001", "1", "0"};
 int m1 = 5, n1 = 3;
 System.out.println("测试用例 1 结果: " + findMaxForm(strs1, m1, n1)); // 预期输出: 4

 // 测试用例 2
 String[] strs2 = {"10", "0", "1"};
 int m2 = 1, n2 = 1;
 System.out.println("测试用例 2 结果: " + findMaxForm(strs2, m2, n2)); // 预期输出: 2
 }

 /**
 * 计算最大子集的长度
 * @param strs 二进制字符串数组
 * @param m 最多可以使用的 0 的数量
 * @param n 最多可以使用的 1 的数量
 */
}
```

```

* @return 最大子集的长度
*/
public static int findMaxForm(String[] strs, int m, int n) {
 if (strs == null || strs.length == 0) {
 return 0;
 }

 // 创建二维 DP 数组, dp[i][j] 表示使用 i 个 0 和 j 个 1 时, 最多可以选择的字符串数量
 int[][] dp = new int[m + 1][n + 1];

 // 遍历每个字符串
 for (String str : strs) {
 // 统计当前字符串中 0 和 1 的数量
 int[] counts = countZeroesOnes(str);
 int zeros = counts[0];
 int ones = counts[1];

 // 逆序遍历, 避免重复使用同一个字符串
 for (int i = m; i >= zeros; i--) {
 for (int j = n; j >= ones; j--) {
 // 更新状态: 不选当前字符串 或 选当前字符串 (如果可以的话)
 dp[i][j] = Math.max(dp[i][j], dp[i - zeros][j - ones] + 1);
 }
 }
 }

 return dp[m][n];
}

/***
 * 统计字符串中 0 和 1 的数量
 * @param s 二进制字符串
 * @return 一个数组, 第一个元素是 0 的数量, 第二个元素是 1 的数量
 */
private static int[] countZeroesOnes(String s) {
 int[] counts = new int[2];
 for (char c : s.toCharArray()) {
 counts[c - '0']++;
 }
 return counts;
}

/***

```

```

* 使用三维 DP 数组的版本（更直观但空间效率较低）
* @param strs 二进制字符串数组
* @param m 最多可以使用的 0 的数量
* @param n 最多可以使用的 1 的数量
* @return 最大子集的长度
*/
public static int findMaxForm3D(String[] strs, int m, int n) {
 if (strs == null || strs.length == 0) {
 return 0;
 }

 int l = strs.length;
 // 创建三维 DP 数组，dp[k][i][j] 表示前 k 个字符串中，使用 i 个 0 和 j 个 1 时，最多可以选择的字符串数量
 int[][][] dp = new int[l + 1][m + 1][n + 1];

 // 遍历每个字符串
 for (int k = 1; k <= l; k++) {
 String str = strs[k - 1];
 int[] counts = countZeroesOnes(str);
 int zeros = counts[0];
 int ones = counts[1];

 // 遍历 0 的数量
 for (int i = 0; i <= m; i++) {
 // 遍历 1 的数量
 for (int j = 0; j <= n; j++) {
 // 不选第 k 个字符串
 dp[k][i][j] = dp[k - 1][i][j];

 // 选第 k 个字符串（如果可以的话）
 if (i >= zeros && j >= ones) {
 dp[k][i][j] = Math.max(dp[k][i][j], dp[k - 1][i - zeros][j - ones] + 1);
 }
 }
 }
 }

 return dp[l][m][n];
}

/**
 * 优化的二维 DP 版本，将统计 0 和 1 的过程提前

```

```

* @param strs 二进制字符串数组
* @param m 最多可以使用的 0 的数量
* @param n 最多可以使用的 1 的数量
* @return 最大子集的长度
*/
public static int findMaxFormOptimized(String[] strs, int m, int n) {
 if (strs == null || strs.length == 0) {
 return 0;
 }

 // 预先统计所有字符串中 0 和 1 的数量
 int[][] counts = new int[strs.length][2];
 for (int i = 0; i < strs.length; i++) {
 counts[i] = countZeroesOnes(strs[i]);
 }

 // 创建二维 DP 数组
 int[][] dp = new int[m + 1][n + 1];

 // 遍历每个字符串
 for (int[] count : counts) {
 int zeros = count[0];
 int ones = count[1];

 // 逆序遍历
 for (int i = m; i >= zeros; i--) {
 for (int j = n; j >= ones; j--) {
 dp[i][j] = Math.max(dp[i][j], dp[i - zeros][j - ones] + 1);
 }
 }
 }

 return dp[m][n];
}

/**
 * 使用递归+记忆化搜索实现
 * 这个方法对于较大的输入可能会超时，但展示了递归的思路
 * @param strs 二进制字符串数组
 * @param m 最多可以使用的 0 的数量
 * @param n 最多可以使用的 1 的数量
 * @return 最大子集的长度
*/

```

```

public static int findMaxFormRecursive(String[] strs, int m, int n) {
 if (strs == null || strs.length == 0) {
 return 0;
 }

 // 预先统计所有字符串中 0 和 1 的数量
 int[][] counts = new int[strs.length][2];
 for (int i = 0; i < strs.length; i++) {
 counts[i] = countZeroesOnes(strs[i]);
 }

 // 创建三维记忆化缓存, memo[index][zeros][ones] 表示从 index 开始的字符串中, 剩余 zeros 和
 // ones 时能选的最大字符串数量
 Integer[][][] memo = new Integer[strs.length][m + 1][n + 1];

 return dfs(counts, 0, m, n, memo);
}

/***
 * 递归辅助函数
 * @param counts 每个字符串中 0 和 1 的数量
 * @param index 当前处理的字符串索引
 * @param zeros 剩余可用的 0 的数量
 * @param ones 剩余可用的 1 的数量
 * @param memo 记忆化缓存
 * @return 最大可以选择的字符串数量
 */
private static int dfs(int[][] counts, int index, int zeros, int ones, Integer[][][] memo) {
 // 基础情况: 处理完所有字符串
 if (index == counts.length) {
 return 0;
 }

 // 检查缓存
 if (memo[index][zeros][ones] != null) {
 return memo[index][zeros][ones];
 }

 // 不选当前字符串
 int notTake = dfs(counts, index + 1, zeros, ones, memo);

 // 选当前字符串 (如果可以的话)
 int take = 0;

```

```

int currentZeros = counts[index][0];
int currentOnes = counts[index][1];
if (zeros >= currentZeros && ones >= currentOnes) {
 take = 1 + dfs(counts, index + 1, zeros - currentZeros, ones - currentOnes, memo);
}
}

// 缓存结果
memo[index][zeros][ones] = Math.max(notTake, take);
return memo[index][zeros][ones];
}
}
=====
```

文件: Code36\_OnesAndZeroes.py

```

LeetCode 474. 一和零
题目描述: 给你一个二进制字符串数组 strs 和两个整数 m 和 n 。
请你找出并返回 strs 的最大子集的长度，该子集中 最多 有 m 个 0 和 n 个 1 。
如果 x 的所有元素也是 y 的元素，集合 x 是集合 y 的 子集 。
链接: https://leetcode.cn/problems/ones-and-zeroes/
#
解题思路:
这是一个二维费用的 0-1 背包问题。每个字符串可以看作是一个物品，它有两个费用：0 的数量和 1 的数量。
我们的背包有两个容量限制：最多可以使用 m 个 0 和 n 个 1。我们的目标是选择尽可能多的物品（字符串）。
#
状态定义: dp[i][j] 表示使用 i 个 0 和 j 个 1 时，最多可以选择的字符串数量
状态转移方程: dp[i][j] = max(dp[i][j], dp[i - zeros][j - ones] + 1)，其中 zeros 和 ones 是当前字符串的 0 和 1 的数量
初始状态: dp[0][0] = 0，其他初始化为 0
#
时间复杂度: O(1 * m * n)，其中 1 是字符串数组的长度
空间复杂度: O(m * n)，使用二维 DP 数组
```

```
def find_max_form(strs: list[str], m: int, n: int) -> int:
```

```
"""

```

计算最大子集的长度

参数:

- strs: 二进制字符串数组
- m: 最多可以使用的 0 的数量
- n: 最多可以使用的 1 的数量

返回：

    最大子集的长度

"""

if not strs:

    return 0

# 创建二维 DP 数组，dp[i][j] 表示使用 i 个 0 和 j 个 1 时，最多可以选择的字符串数量

dp = [[0] \* (n + 1) for \_ in range(m + 1)]

# 遍历每个字符串

for s in strs:

    # 统计当前字符串中 0 和 1 的数量

    zeros = s.count('0')

    ones = s.count('1')

    # 逆序遍历，避免重复使用同一个字符串

    for i in range(m, zeros - 1, -1):

        for j in range(n, ones - 1, -1):

            # 更新状态：不选当前字符串 或 选当前字符串（如果可以的话）

            dp[i][j] = max(dp[i][j], dp[i - zeros][j - ones] + 1)

return dp[m][n]

def find\_max\_form\_3d(strs: list[str], m: int, n: int) -> int:

"""

使用三维 DP 数组的版本（更直观但空间效率较低）

参数：

    strs: 二进制字符串数组

    m: 最多可以使用的 0 的数量

    n: 最多可以使用的 1 的数量

返回：

    最大子集的长度

"""

if not strs:

    return 0

l = len(strs)

# 创建三维 DP 数组，dp[k][i][j] 表示前 k 个字符串中，使用 i 个 0 和 j 个 1 时，最多可以选择的字符串数量

dp = [[[0] \* (n + 1) for \_ in range(m + 1)] for \_\_ in range(l + 1)]

```

遍历每个字符串
for k in range(1, l + 1):
 s = strs[k - 1]
 zeros = s.count('0')
 ones = s.count('1')

遍历 0 的数量
for i in range(m + 1):
 # 遍历 1 的数量
 for j in range(n + 1):
 # 不选第 k 个字符串
 dp[k][i][j] = dp[k - 1][i][j]

 # 选第 k 个字符串 (如果可以的话)
 if i >= zeros and j >= ones:
 dp[k][i][j] = max(dp[k][i][j], dp[k - 1][i - zeros][j - ones] + 1)

return dp[l][m][n]

```

```
def find_max_form_optimized(strs: list[str], m: int, n: int) -> int:
```

```
"""

```

优化的二维 DP 版本，将统计 0 和 1 的过程提前

参数：

strs: 二进制字符串数组  
 m: 最多可以使用的 0 的数量  
 n: 最多可以使用的 1 的数量

返回：

最大子集的长度

```
"""

```

```
if not strs:
 return 0
```

```
预先统计所有字符串中 0 和 1 的数量
counts = []
for s in strs:
 counts.append((s.count('0'), s.count('1')))
```

# 创建二维 DP 数组

```
dp = [[0] * (n + 1) for _ in range(m + 1)]
```

# 遍历每个字符串

```
for zeros, ones in counts:
 # 逆序遍历
 for i in range(m, zeros - 1, -1):
 for j in range(n, ones - 1, -1):
 dp[i][j] = max(dp[i][j], dp[i - zeros][j - ones] + 1)

return dp[m][n]
```

```
from functools import lru_cache
```

```
def find_max_form_recursive(strs: list[str], m: int, n: int) -> int:
```

```
"""
```

使用递归+记忆化搜索实现

这个方法对于较大的输入可能会超时，但展示了递归的思路

参数：

strs: 二进制字符串数组  
m: 最多可以使用的 0 的数量  
n: 最多可以使用的 1 的数量

返回：

最大子集的长度

```
"""
```

```
if not strs:
 return 0
```

# 预先统计所有字符串中 0 和 1 的数量

```
counts = []
for s in strs:
 counts.append((s.count('0'), s.count('1')))
```

# 使用 lru\_cache 装饰器进行记忆化

```
@lru_cache(maxsize=None)
def dfs(index: int, zeros_left: int, ones_left: int) -> int:
 """
```

递归辅助函数

参数：

index: 当前处理的字符串索引  
zeros\_left: 剩余可用的 0 的数量  
ones\_left: 剩余可用的 1 的数量

返回：

```

 最大可以选择的字符串数量
 """
基础情况：处理完所有字符串
if index == len(counts):
 return 0

不选当前字符串
not_take = dfs(index + 1, zeros_left, ones_left)

选当前字符串（如果可以的话）
take = 0
zeros, ones = counts[index]
if zeros_left >= zeros and ones_left >= ones:
 take = 1 + dfs(index + 1, zeros_left - zeros, ones_left - ones)

return max(not_take, take)

return dfs(0, m, n)

测试代码
if __name__ == "__main__":
 # 测试用例 1
 strs1 = ["10", "0001", "111001", "1", "0"]
 m1 = 5
 n1 = 3
 print(f"测试用例 1 结果: {find_max_form(strs1, m1, n1)}") # 预期输出: 4
 print(f"三维 DP 版本: {find_max_form_3d(strs1, m1, n1)}")
 print(f"优化版本: {find_max_form_optimized(strs1, m1, n1)}")
 print(f"递归版本: {find_max_form_recursive(strs1, m1, n1)}")

 # 测试用例 2
 strs2 = ["10", "0", "1"]
 m2 = 1
 n2 = 1
 print(f"测试用例 2 结果: {find_max_form(strs2, m2, n2)}") # 预期输出: 2
 print(f"三维 DP 版本: {find_max_form_3d(strs2, m2, n2)}")
 print(f"优化版本: {find_max_form_optimized(strs2, m2, n2)}")
 print(f"递归版本: {find_max_form_recursive(strs2, m2, n2)}")

 # 测试用例 3
 strs3 = ["001", "110", "0000", "0000"]
 m3 = 9
 n3 = 3

```

```
print(f"测试用例 3 结果: {find_max_form(strs3, m3, n3)}") # 预期输出: 4
```

```
=====
```

文件: Code37\_PartitionEqualSubsetSum.cpp

```
=====
```

// LeetCode 416. 分割等和子集

// 题目描述: 给你一个只包含正整数的非空数组 nums。请你判断是否可以将这个数组分割成两个子集，使得两个子集的元素和相等。

// 链接: <https://leetcode.cn/problems/partition-equal-subset-sum/>

//

// 解题思路:

// 这是一个 0-1 背包问题的变种。问题可以转化为: 是否存在一个子集，其和等于整个数组和的一半。

//

// 状态定义: dp[j] 表示是否能组成和为 j 的子集

// 状态转移方程: dp[j] = dp[j] || dp[j - nums[i]]

// 初始状态: dp[0] = true, 表示空子集的和为 0 是可以组成的

//

// 时间复杂度: O(n \* target)，其中 n 是数组长度，target 是数组和的一半

// 空间复杂度: O(target)，使用一维 DP 数组

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

/\*\*

\* 判断是否可以将数组分割成两个和相等的子集

\* @param nums 非空数组，只包含正整数

\* @return 是否可以分割

\*/

```
bool canPartition(vector<int>& nums) {
```

```
 if (nums.size() < 2) {
```

```
 return false;
```

```
}
```

// 计算数组总和

```
 int sum = accumulate(nums.begin(), nums.end(), 0);
```

// 如果总和是奇数，不可能分成两个和相等的子集

```
 if (sum % 2 != 0) {
```

```

 return false;
 }

 // 目标和为总和的一半
 int target = sum / 2;

 // 创建 DP 数组, dp[j] 表示是否能组成和为 j 的子集
 vector<bool> dp(target + 1, false);

 // 初始状态: 空子集的和为 0 是可以组成的
 dp[0] = true;

 // 遍历每个数字
 for (int num : nums) {
 // 逆序遍历, 避免重复使用同一个数字
 for (int j = target; j >= num; j--) {
 // 更新状态: 不选当前数字 或 选当前数字 (如果可以的话)
 dp[j] = dp[j] || dp[j - num];
 }
 }

 return dp[target];
}

/***
 * 使用二维 DP 数组的版本 (更直观但空间效率较低)
 * @param nums 非空数组, 只包含正整数
 * @return 是否可以分割
 */
bool canPartition2D(vector<int>& nums) {
 if (nums.size() < 2) {
 return false;
 }

 // 计算数组总和
 int sum = accumulate(nums.begin(), nums.end(), 0);

 // 如果总和是奇数, 不可能分成两个和相等的子集
 if (sum % 2 != 0) {
 return false;
 }

 // 目标和为总和的一半

```

```

int target = sum / 2;
int n = nums.size();

// 创建二维 DP 数组, dp[i][j] 表示前 i 个数字是否能组成和为 j 的子集
vector<vector<bool>> dp(n + 1, vector<bool>(target + 1, false));

// 初始状态: 空子集的和为 0 是可以组成的
for (int i = 0; i <= n; i++) {
 dp[i][0] = true;
}

// 填充 DP 数组
for (int i = 1; i <= n; i++) {
 for (int j = 1; j <= target; j++) {
 // 不选第 i 个数字
 dp[i][j] = dp[i-1][j];

 // 选第 i 个数字 (如果可以的话)
 if (j >= nums[i-1]) {
 dp[i][j] = dp[i][j] || dp[i-1][j - nums[i-1]];
 }
 }
}

return dp[n][target];
}

/***
 * 优化的一维 DP 版本, 提前处理一些边界情况
 * @param nums 非空数组, 只包含正整数
 * @return 是否可以分割
 */
bool canPartitionOptimized(vector<int>& nums) {
 if (nums.size() < 2) {
 return false;
 }

 // 计算数组总和
 int sum = 0;
 int maxNum = 0;
 for (int num : nums) {
 sum += num;
 maxNum = max(maxNum, num);
 }
}

```

```
}

// 如果总和是奇数，不可能分成两个和相等的子集
if (sum % 2 != 0) {
 return false;
}

// 目标和为总和的一半
int target = sum / 2;

// 如果最大的数字大于目标和，不可能分割
if (maxNum > target) {
 return false;
}

// 创建 DP 数组
vector<bool> dp(target + 1, false);
dp[0] = true;

for (int num : nums) {
 for (int j = target; j >= num; j--) {
 dp[j] = dp[j] || dp[j - num];
 }
}

return dp[target];
}

/***
 * 使用递归+记忆化搜索实现
 * 这个方法对于较大的输入可能会超时，但展示了递归的思路
 * @param nums 非空数组，只包含正整数
 * @return 是否可以分割
 */
bool canPartitionRecursive(vector<int>& nums) {
 if (nums.size() < 2) {
 return false;
 }

 // 计算数组总和
 int sum = accumulate(nums.begin(), nums.end(), 0);

 // 如果总和是奇数，不可能分成两个和相等的子集

```

```

if (sum % 2 != 0) {
 return false;
}

// 目标和为总和的一半
int target = sum / 2;
int n = nums.size();

// 创建记忆化缓存
vector<vector<int>> memo(n, vector<int>(target + 1, -1)); // -1 表示未计算

function<bool(int, int)> dfs = [&](int index, int currentSum) -> bool {
 // 找到目标和
 if (currentSum == target) {
 return true;
 }

 // 超过目标和或处理完所有元素
 if (currentSum > target || index == n) {
 return false;
 }

 // 检查缓存
 if (memo[index][currentSum] != -1) {
 return memo[index][currentSum];
 }

 // 递归调用：选当前元素 或 不选当前元素
 bool result = dfs(index + 1, currentSum + nums[index]) ||
 dfs(index + 1, currentSum);

 // 缓存结果
 memo[index][currentSum] = result;
 return result;
};

return dfs(0, 0);
}

/***
 * 使用位操作优化的版本
 * 对于较大的数组但元素值不大的情况，位操作可以更高效
 * @param nums 非空数组，只包含正整数
 */

```

```

* @return 是否可以分割
*/
bool canPartitionBitSet(vector<int>& nums) {
 if (nums.size() < 2) {
 return false;
 }

 // 计算数组总和
 int sum = accumulate(nums.begin(), nums.end(), 0);

 // 如果总和是奇数，不可能分成两个和相等的子集
 if (sum % 2 != 0) {
 return false;
 }

 // 目标和为总和的一半
 int target = sum / 2;

 // 使用位集表示可达的和
 // bitset 的第 i 位为 1 表示和为 i 是可达的
 unsigned long long bitset = 1; // 初始状态，和为 0 是可达的

 for (int num : nums) {
 // 位操作：当前可达的和 | (之前可达的和 + 当前数字)
 bitset |= bitset << num;
 }

 // 检查目标和是否可达
 return (bitset & (1ULL << target)) != 0;
}

// 打印数组函数
void printArray(const vector<int>& arr) {
 cout << "[";
 for (size_t i = 0; i < arr.size(); i++) {
 cout << arr[i];
 if (i < arr.size() - 1) {
 cout << ", ";
 }
 }
 cout << "]" << endl;
}

```

```

// 主函数，用于测试
int main() {
 // 测试用例 1
 vector<int> nums1 = {1, 5, 11, 5};
 cout << "测试用例 1 (";
 printArray(nums1);
 cout << ") 结果: " << (canPartition(nums1) ? "true" : "false") << endl; // 预期输出: true

 // 测试用例 2
 vector<int> nums2 = {1, 2, 3, 5};
 cout << "测试用例 2 (";
 printArray(nums2);
 cout << ") 结果: " << (canPartition(nums2) ? "true" : "false") << endl; // 预期输出: false

 // 测试不同实现
 cout << "\n 测试不同实现:\n";
 cout << "二维 DP 版本 (测试用例 1): " << (canPartition2D(nums1) ? "true" : "false") << endl;
 cout << "优化版本 (测试用例 1): " << (canPartitionOptimized(nums1) ? "true" : "false") <<
endl;
 cout << "递归版本 (测试用例 1): " << (canPartitionRecursive(nums1) ? "true" : "false") <<
endl;
 cout << "位操作版本 (测试用例 1): " << (canPartitionBitSet(nums1) ? "true" : "false") << endl;

 // 测试用例 3
 vector<int> nums3 = {1, 2, 3, 4, 5, 6, 7};
 cout << "\n 测试用例 3 (";
 printArray(nums3);
 cout << ") 结果: " << (canPartition(nums3) ? "true" : "false") << endl; // 预期输出: true

 // 测试用例 4
 vector<int> nums4 = {100, 100, 100, 100, 100, 100, 100, 100};
 cout << "\n 测试用例 4 (多个 100): " << (canPartition(nums4) ? "true" : "false") << endl; // 预
期输出: true

 return 0;
}

```

=====

文件: Code37\_PartitionEqualSubsetSum.java

=====

```

package class073;

```

```
// LeetCode 416. 分割等和子集
// 题目描述：给你一个只包含正整数的非空数组 nums。请你判断是否可以将这个数组分割成两个子集，使得
// 两个子集的元素和相等。
// 链接: https://leetcode.cn/problems/partition-equal-subset-sum/
//
// 解题思路：
// 这是一个 0-1 背包问题的变种。问题可以转化为：是否存在一个子集，其和等于整个数组和的一半。
// 状态定义：dp[j] 表示是否能组成和为 j 的子集
// 状态转移方程：dp[j] = dp[j] || dp[j - nums[i]]
// 初始状态：dp[0] = true，表示空子集的和为 0 是可以组成的
// 时间复杂度：O(n * target)，其中 n 是数组长度，target 是数组和的一半
// 空间复杂度：O(target)，使用一维 DP 数组
```

```
public class Code37_PartitionEqualSubsetSum {

 // 主方法，用于测试
 public static void main(String[] args) {
 // 测试用例 1
 int[] nums1 = {1, 5, 11, 5};
 System.out.println("测试用例 1 结果：" + canPartition(nums1)); // 预期输出: true

 // 测试用例 2
 int[] nums2 = {1, 2, 3, 5};
 System.out.println("测试用例 2 结果：" + canPartition(nums2)); // 预期输出: false
 }

 /**
 * 判断是否可以将数组分割成两个和相等的子集
 * @param nums 非空数组，只包含正整数
 * @return 是否可以分割
 */
 public static boolean canPartition(int[] nums) {
 if (nums == null || nums.length < 2) {
 return false;
 }

 // 计算数组总和
 int sum = 0;
 for (int num : nums) {
 sum += num;
 }
```

```

// 如果总和是奇数，不可能分成两个和相等的子集
if (sum % 2 != 0) {
 return false;
}

// 目标和为总和的一半
int target = sum / 2;

// 创建 DP 数组，dp[j] 表示是否能组成和为 j 的子集
boolean[] dp = new boolean[target + 1];

// 初始状态：空子集的和为 0 是可以组成的
dp[0] = true;

// 遍历每个数字
for (int num : nums) {
 // 逆序遍历，避免重复使用同一个数字
 for (int j = target; j >= num; j--) {
 // 更新状态：不选当前数字 或 选当前数字（如果可以的话）
 dp[j] = dp[j] || dp[j - num];
 }
}

return dp[target];
}

/**
 * 使用二维 DP 数组的版本（更直观但空间效率较低）
 * @param nums 非空数组，只包含正整数
 * @return 是否可以分割
 */
public static boolean canPartition2D(int[] nums) {
 if (nums == null || nums.length < 2) {
 return false;
 }

 // 计算数组总和
 int sum = 0;
 for (int num : nums) {
 sum += num;
 }
}

```

```

// 如果总和是奇数，不可能分成两个和相等的子集
if (sum % 2 != 0) {
 return false;
}

// 目标和为总和的一半
int target = sum / 2;
int n = nums.length;

// 创建二维 DP 数组，dp[i][j] 表示前 i 个数字是否能组成和为 j 的子集
boolean[][] dp = new boolean[n + 1][target + 1];

// 初始状态：空子集的和为 0 是可以组成的
for (int i = 0; i <= n; i++) {
 dp[i][0] = true;
}

// 填充 DP 数组
for (int i = 1; i <= n; i++) {
 for (int j = 1; j <= target; j++) {
 // 不选第 i 个数字
 dp[i][j] = dp[i-1][j];

 // 选第 i 个数字（如果可以的话）
 if (j >= nums[i-1]) {
 dp[i][j] = dp[i][j] || dp[i-1][j - nums[i-1]];
 }
 }
}

return dp[n][target];
}

/**
 * 优化的一维 DP 版本，提前处理一些边界情况
 * @param nums 非空数组，只包含正整数
 * @return 是否可以分割
 */
public static boolean canPartitionOptimized(int[] nums) {
 if (nums == null || nums.length < 2) {
 return false;
 }

 int sum = 0;
 for (int num : nums) {
 sum += num;
 }

 if (sum % 2 != 0) {
 return false;
 }

 int target = sum / 2;
 boolean[] dp = new boolean[target + 1];
 dp[0] = true;

 for (int i = 0; i < nums.length; i++) {
 for (int j = target; j >= 0; j--) {
 if (dp[j]) {
 dp[j] = dp[j];
 } else if (j >= nums[i]) {
 dp[j] = dp[j] || dp[j - nums[i]];
 }
 }
 }

 return dp[target];
}

```

```
// 计算数组总和
int sum = 0;
int maxNum = 0;
for (int num : nums) {
 sum += num;
 maxNum = Math.max(maxNum, num);
}

// 如果总和是奇数，不可能分成两个和相等的子集
if (sum % 2 != 0) {
 return false;
}

// 目标和为总和的一半
int target = sum / 2;

// 如果最大的数字大于目标和，不可能分割
if (maxNum > target) {
 return false;
}

// 创建 DP 数组
boolean[] dp = new boolean[target + 1];
dp[0] = true;

for (int num : nums) {
 for (int j = target; j >= num; j--) {
 dp[j] = dp[j] || dp[j - num];
 }
}

return dp[target];
}

/**
 * 使用递归+记忆化搜索实现
 * 这个方法对于较大的输入可能会超时，但展示了递归的思路
 * @param nums 非空数组，只包含正整数
 * @return 是否可以分割
*/
public static boolean canPartitionRecursive(int[] nums) {
 if (nums == null || nums.length < 2) {
 return false;
 }
```

```
}

// 计算数组总和
int sum = 0;
for (int num : nums) {
 sum += num;
}

// 如果总和是奇数，不可能分成两个和相等的子集
if (sum % 2 != 0) {
 return false;
}

// 目标和为总和的一半
int target = sum / 2;
int n = nums.length;

// 创建记忆化缓存
Boolean[][] memo = new Boolean[n][target + 1];

return dfs(nums, 0, 0, target, memo);
}

/**
 * 递归辅助函数
 * @param nums 数组
 * @param index 当前处理的索引
 * @param currentSum 当前子集和
 * @param target 目标和
 * @param memo 记忆化缓存
 * @return 是否能组成目标和
 */
private static boolean dfs(int[] nums, int index, int currentSum, int target, Boolean[][] memo) {
 // 找到目标和
 if (currentSum == target) {
 return true;
 }

 // 超过目标和或处理完所有元素
 if (currentSum > target || index == nums.length) {
 return false;
 }

 // 从当前索引开始遍历数组
 for (int i = index; i < n; i++) {
 // 将当前元素加入子集
 currentSum += nums[i];
 // 如果当前子集和等于目标和，返回 true
 if (currentSum == target) {
 return true;
 }
 // 递归调用，继续处理下一个元素
 if (dfs(nums, i + 1, currentSum, target, memo)) {
 return true;
 }
 // 回溯，移除当前元素
 currentSum -= nums[i];
 }
}

// 检查是否能组成目标和
public boolean canPartition(int[] nums) {
 if (nums == null || nums.length == 0) {
 return false;
 }
 int sum = 0;
 for (int num : nums) {
 sum += num;
 }
 if (sum % 2 != 0) {
 return false;
 }
 return dfs(nums, 0, 0, sum / 2, new Boolean[nums.length][sum / 2 + 1]);
}
```

```

// 检查缓存
if (memo[index][currentSum] != null) {
 return memo[index][currentSum];
}

// 递归调用：选当前元素 或 不选当前元素
boolean result = dfs(nums, index + 1, currentSum + nums[index], target, memo)
 || dfs(nums, index + 1, currentSum, target, memo);

// 缓存结果
memo[index][currentSum] = result;
return result;
}

/***
 * 使用位操作优化的版本
 * 对于较大的数组但元素值不大的情况，位操作可以更高效
 * @param nums 非空数组，只包含正整数
 * @return 是否可以分割
 */
public static boolean canPartitionBitSet(int[] nums) {
 if (nums == null || nums.length < 2) {
 return false;
 }

 // 计算数组总和
 int sum = 0;
 for (int num : nums) {
 sum += num;
 }

 // 如果总和是奇数，不可能分成两个和相等的子集
 if (sum % 2 != 0) {
 return false;
 }

 // 目标和为总和的一半
 int target = sum / 2;

 // 使用位集表示可达的和
 // bitset 的第 i 位为 1 表示和为 i 是可达的
 int bitset = 1; // 初始状态，和为 0 是可达的

```

```

 for (int num : nums) {
 // 位操作: 当前可达的和 | (之前可达的和 + 当前数字)
 bitset |= bitset << num;
 }

 // 检查目标和是否可达
 return (bitset & (1 << target)) != 0;
 }
}

```

=====

文件: Code37\_PartitionEqualSubsetSum.py

=====

```

LeetCode 416. 分割等和子集
题目描述: 给你一个只包含正整数的非空数组 nums。请你判断是否可以将这个数组分割成两个子集，使得两个子集的元素和相等。
链接: https://leetcode.cn/problems/partition-equal-subset-sum/
#
解题思路:
这是一个 0-1 背包问题的变种。问题可以转化为: 是否存在一个子集，其和等于整个数组和的一半。
#
状态定义: dp[j] 表示是否能组成和为 j 的子集
状态转移方程: dp[j] = dp[j] || dp[j - nums[i]]
初始状态: dp[0] = True, 表示空子集的和为 0 是可以组成的
#
时间复杂度: O(n * target)，其中 n 是数组长度，target 是数组和的一半
空间复杂度: O(target)，使用一维 DP 数组

```

def can\_partition(nums: list[int]) -> bool:

"""

判断是否可以将数组分割成两个和相等的子集

参数:

nums: 非空数组，只包含正整数

返回:

是否可以分割

"""

```

if len(nums) < 2:
 return False

```

```

计算数组总和
total_sum = sum(nums)

如果总和是奇数，不可能分成两个和相等的子集
if total_sum % 2 != 0:
 return False

目标和为总和的一半
target = total_sum // 2

创建 DP 数组，dp[j] 表示是否能组成和为 j 的子集
dp = [False] * (target + 1)

初始状态：空子集的和为 0 是可以组成的
dp[0] = True

遍历每个数字
for num in nums:
 # 逆序遍历，避免重复使用同一个数字
 for j in range(target, num - 1, -1):
 # 更新状态：不选当前数字 或 选当前数字（如果可以的话）
 dp[j] = dp[j] or dp[j - num]

return dp[target]

```

```
def can_partition_2d(nums: list[int]) -> bool:
```

```
"""

```

使用二维 DP 数组的版本（更直观但空间效率较低）

参数：

nums：非空数组，只包含正整数

返回：

是否可以分割

```
"""

```

```
if len(nums) < 2:
 return False
```

# 计算数组总和

```
total_sum = sum(nums)
```

# 如果总和是奇数，不可能分成两个和相等的子集

```
if total_sum % 2 != 0:
```

```

 return False

目标和为总和的一半
target = total_sum // 2
n = len(nums)

创建二维 DP 数组, dp[i][j] 表示前 i 个数字是否能组成和为 j 的子集
dp = [[False] * (target + 1) for _ in range(n + 1)]

初始状态: 空子集的和为 0 是可以组成的
for i in range(n + 1):
 dp[i][0] = True

填充 DP 数组
for i in range(1, n + 1):
 for j in range(1, target + 1):
 # 不选第 i 个数字
 dp[i][j] = dp[i-1][j]

 # 选第 i 个数字 (如果可以的话)
 if j >= nums[i-1]:
 dp[i][j] = dp[i][j] or dp[i-1][j - nums[i-1]]

return dp[n][target]

```

```
def can_partition_optimized(nums: list[int]) -> bool:
```

```
"""

```

优化的一维 DP 版本, 提前处理一些边界情况

参数:

nums: 非空数组, 只包含正整数

返回:

是否可以分割

```
"""

```

```
if len(nums) < 2:
 return False
```

# 计算数组总和

```
total_sum = sum(nums)
```

```
max_num = max(nums)
```

# 如果总和是奇数, 不可能分成两个和相等的子集

```
if total_sum % 2 != 0:
 return False

目标和为总和的一半
target = total_sum // 2

如果最大的数字大于目标和，不可能分割
if max_num > target:
 return False

创建 DP 数组
dp = [False] * (target + 1)
dp[0] = True

for num in nums:
 for j in range(target, num - 1, -1):
 dp[j] = dp[j] or dp[j - num]

return dp[target]
```

```
from functools import lru_cache

def can_partition_recursive(nums: list[int]) -> bool:
 """
 使用递归+记忆化搜索实现
 这个方法对于较大的输入可能会超时，但展示了递归的思路

 参数：
 nums: 非空数组，只包含正整数

 返回：
 是否可以分割
 """
```

```
if len(nums) < 2:
 return False

计算数组总和
total_sum = sum(nums)

如果总和是奇数，不可能分成两个和相等的子集
if total_sum % 2 != 0:
 return False
```

```
目标和为总和的一半
target = total_sum // 2
n = len(nums)
```

```
@lru_cache(maxsize=None)
def dfs(index: int, current_sum: int) -> bool:
 """
 递归辅助函数
```

参数:

index: 当前处理的索引  
current\_sum: 当前子集和

返回:

是否能组成目标和

"""

# 找到目标和

```
if current_sum == target:
 return True
```

# 超过目标和或处理完所有元素

```
if current_sum > target or index == n:
 return False
```

# 递归调用: 选当前元素 或 不选当前元素

```
return dfs(index + 1, current_sum + nums[index]) or dfs(index + 1, current_sum)
```

```
return dfs(0, 0)
```

```
def can_partition_bit_set(nums: list[int]) -> bool:
 """
```

使用位操作优化的版本

对于较大的数组但元素值不大的情况，位操作可以更高效

参数:

nums: 非空数组，只包含正整数

返回:

是否可以分割

"""

```
if len(nums) < 2:
 return False
```

```
计算数组总和
total_sum = sum(nums)

如果总和是奇数，不可能分成两个和相等的子集
if total_sum % 2 != 0:
 return False

目标和为总和的一半
target = total_sum // 2

使用位集表示可达的和
bitset 的第 i 位为 1 表示和为 i 是可达的
bitset = 1 # 初始状态，和为 0 是可达的

for num in nums:
 # 位操作：当前可达的和 | (之前可达的和 + 当前数字)
 # 使用位移操作，将之前的状态左移 num 位，然后与原来的状态进行或操作
 bitset |= bitset << num

 # 优化：如果已经能达到目标和，可以提前返回
 if bitset & (1 << target):
 return True

检查目标和是否可达
return bool(bitset & (1 << target))

测试代码
if __name__ == "__main__":
 # 测试用例 1
 nums1 = [1, 5, 11, 5]
 print(f"测试用例 1 结果: {can_partition(nums1)}") # 预期输出: True

 # 测试用例 2
 nums2 = [1, 2, 3, 5]
 print(f"测试用例 2 结果: {can_partition(nums2)}") # 预期输出: False

 # 测试不同实现
 print("\n测试不同实现:")
 print(f"二维 DP 版本 (测试用例 1): {can_partition_2d(nums1)}")
 print(f"优化版本 (测试用例 1): {can_partition_optimized(nums1)}")
 print(f"递归版本 (测试用例 1): {can_partition_recursive(nums1)}")
 print(f"位操作版本 (测试用例 1): {can_partition_bit_set(nums1)}")
```

```
测试用例 3
nums3 = [1, 2, 3, 4, 5, 6, 7]
print(f"\n 测试用例 3 结果: {can_partition(nums3)}") # 预期输出: True
```

```
测试用例 4
nums4 = [100, 100, 100, 100, 100, 100, 100]
print(f"\n 测试用例 4 结果: {can_partition(nums4)}") # 预期输出: True
```

```
测试用例 5
nums5 = [1, 2, 5]
print(f"\n 测试用例 5 结果: {can_partition(nums5)}") # 预期输出: False
```

=====

文件: Code38\_WordBreak.cpp

=====

```
// LeetCode 139. 单词拆分
// 题目描述: 给你一个字符串 s 和一个字符串列表 wordDict 作为字典。请你判断是否可以利用字典中出现的单词拼接出 s 。
// 注意: 不要求字典中出现的单词全部都使用, 并且字典中的单词可以重复使用。
// 链接: https://leetcode.cn/problems/word-break/
//
// 解题思路:
// 这是一个完全背包问题的变种。我们可以将字符串 s 看作是背包, 将字典中的单词看作是物品。
// 问题转化为: 是否可以从字典中选择一些单词 (可以重复选择), 使得它们的拼接恰好等于字符串 s。
//
// 状态定义: dp[i] 表示字符串 s 的前 i 个字符是否可以被拆分成字典中的单词
// 状态转移方程: 对于每个 i, 我们检查所有 j < i, 如果 dp[j] 为 true 且 s.substr(j, i-j) 在字典中, 则 dp[i] 为 true
// 初始状态: dp[0] = true, 表示空字符串可以被拆分
//
// 时间复杂度: O(n^3), 其中 n 是字符串 s 的长度
// 空间复杂度: O(n + m), 其中 m 是字典中所有单词的字符总数
```

```
#include <iostream>
#include <vector>
#include <string>
#include <unordered_set>
#include <queue>
using namespace std;

/***
 * 判断字符串是否可以被拆分成字典中的单词
```

```

* @param s 字符串
* @param wordDict 字典
* @return 是否可以拆分
*/
bool wordBreak(string s, vector<string>& wordDict) {
 if (s.empty()) {
 return false;
 }

 // 将字典转换为集合，提高查找效率
 unordered_set<string> wordSet(wordDict.begin(), wordDict.end());
 int n = s.length();

 // 创建 DP 数组，dp[i] 表示字符串 s 的前 i 个字符是否可以被拆分成字典中的单词
 vector<bool> dp(n + 1, false);

 // 初始状态：空字符串可以被拆分
 dp[0] = true;

 // 遍历字符串的每个位置
 for (int i = 1; i <= n; i++) {
 // 遍历所有可能的拆分点 j
 for (int j = 0; j < i; j++) {
 // 如果 dp[j] 为 true (前 j 个字符可以拆分)，且 s.substr(j, i - j) 在字典中，那么 dp[i] 为
 true
 if (dp[j] && wordSet.find(s.substr(j, i - j)) != wordSet.end()) {
 dp[i] = true;
 break; // 只要找到一个可行的拆分方式就可以提前结束内层循环
 }
 }
 }

 return dp[n];
}

/***
 * 优化的版本，限制 j 的范围为最大单词长度，避免不必要的检查
 * @param s 字符串
 * @param wordDict 字典
 * @return 是否可以拆分
*/
bool wordBreakOptimized(string s, vector<string>& wordDict) {
 if (s.empty()) {

```

```

 return false;
}

// 将字典转换为集合，提高查找效率
unordered_set<string> wordSet(wordDict.begin(), wordDict.end());
int n = s.length();

// 找出字典中最长单词的长度
int maxLength = 0;
for (const string& word : wordDict) {
 maxLength = max(maxLength, (int)word.length());
}

// 创建 DP 数组
vector<bool> dp(n + 1, false);
dp[0] = true;

// 遍历字符串的每个位置
for (int i = 1; i <= n; i++) {
 // 只检查 j >= i - maxLength 的情况，避免不必要的检查
 int start = max(0, i - maxLength);
 for (int j = start; j < i; j++) {
 if (dp[j] && wordSet.find(s.substr(j, i - j)) != wordSet.end()) {
 dp[i] = true;
 break;
 }
 }
}

return dp[n];
}

/***
 * 使用递归+记忆化搜索实现
 * 这个方法对于较大的输入可能会超时，但展示了递归的思路
 * @param s 字符串
 * @param wordDict 字典
 * @return 是否可以拆分
 */
bool wordBreakRecursive(string s, vector<string>& wordDict) {
 if (s.empty()) {
 return false;
 }

 // 将字典转换为集合，提高查找效率
 unordered_set<string> wordSet(wordDict.begin(), wordDict.end());
 int n = s.length();

 // 找出字典中最长单词的长度
 int maxLength = 0;
 for (const string& word : wordDict) {
 maxLength = max(maxLength, (int)word.length());
 }

 // 创建 DP 数组
 vector<bool> dp(n + 1, false);
 dp[0] = true;

 // 遍历字符串的每个位置
 for (int i = 1; i <= n; i++) {
 // 只检查 j >= i - maxLength 的情况，避免不必要的检查
 int start = max(0, i - maxLength);
 for (int j = start; j < i; j++) {
 if (dp[j] && wordSet.find(s.substr(j, i - j)) != wordSet.end()) {
 dp[i] = true;
 break;
 }
 }
 }

 return dp[n];
}

```

```

// 将字典转换为集合
unordered_set<string> wordSet(wordDict.begin(), wordDict.end());

// 创建记忆化缓存, memo[i]表示从位置 i 开始的子串是否可以被拆分
vector<int> memo(s.length(), -1); // -1 表示未计算, 0 表示 false, 1 表示 true

function<bool(int)> dfs = [&](int start) -> bool {
 // 基础情况: 已经处理到字符串末尾
 if (start == s.length()) {
 return true;
 }

 // 检查缓存
 if (memo[start] != -1) {
 return memo[start] == 1;
 }

 // 尝试所有可能的结束位置
 for (int end = start + 1; end <= s.length(); end++) {
 // 如果 s.substr(start, end-start) 在字典中, 且剩余部分可以拆分, 则返回 true
 if (wordSet.find(s.substr(start, end - start)) != wordSet.end() && dfs(end)) {
 memo[start] = 1;
 return true;
 }
 }

 // 如果所有可能性都尝试过仍无法拆分, 返回 false
 memo[start] = 0;
 return false;
};

return dfs(0);
}

/***
 * 使用 BFS 实现
 * @param s 字符串
 * @param wordDict 字典
 * @return 是否可以拆分
 */
bool wordBreakBFS(string s, vector<string>& wordDict) {
 if (s.empty()) {

```

```

 return false;
 }

unordered_set<string> wordSet(wordDict.begin(), wordDict.end());
vector<bool> visited(s.length(), false); // 记录哪些位置已经被访问过，避免重复处理

// 使用队列进行 BFS，队列中存储的是当前处理到的位置
queue<int> q;
q.push(0);
visited[0] = true;

while (!q.empty()) {
 int start = q.front();
 q.pop();

 // 尝试所有可能的结束位置
 for (int end = start + 1; end <= s.length(); end++) {
 // 如果当前子串在字典中，且结束位置尚未访问过，则继续 BFS
 if (wordSet.find(s.substr(start, end - start)) != wordSet.end() && !visited[end]) {
 if (end == s.length()) {
 return true; // 已经到达字符串末尾，找到了解决方案
 }
 q.push(end);
 visited[end] = true;
 }
 }
}

return false; // 无法拆分
}

// 打印测试结果
void printResult(const string& s, vector<string>& wordDict, bool result) {
 cout << "字符串: " << s << "\n";
 cout << "字典: [";
 for (size_t i = 0; i < wordDict.size(); i++) {
 cout << "\" " << wordDict[i] << "\"";
 if (i < wordDict.size() - 1) {
 cout << ", ";
 }
 }
 cout << "] \n";
 cout << "结果: " << (result ? "true" : "false") << "\n";
}

```

```

cout << "-----\n";
}

// 主函数, 用于测试
int main() {
 // 测试用例 1
 string s1 = "leetcode";
 vector<string> wordDict1 = {"leet", "code"};
 cout << "测试用例 1:\n";
 printResult(s1, wordDict1, wordBreak(s1, wordDict1));

 // 测试用例 2
 string s2 = "applepenapple";
 vector<string> wordDict2 = {"apple", "pen"};
 cout << "测试用例 2:\n";
 printResult(s2, wordDict2, wordBreak(s2, wordDict2));

 // 测试用例 3
 string s3 = "catsandog";
 vector<string> wordDict3 = {"cats", "dog", "sand", "and", "cat"};
 cout << "测试用例 3:\n";
 printResult(s3, wordDict3, wordBreak(s3, wordDict3));

 // 测试不同实现
 cout << "测试不同实现:\n";
 cout << "优化版本 (测试用例 1): " << (wordBreakOptimized(s1, wordDict1) ? "true" : "false") << "\n";
 cout << "递归版本 (测试用例 1): " << (wordBreakRecursive(s1, wordDict1) ? "true" : "false") << "\n";
 cout << "BFS 版本 (测试用例 1): " << (wordBreakBFS(s1, wordDict1) ? "true" : "false") << "\n";

 // 测试用例 4
 string s4 = "catsandog";
 vector<string> wordDict4 = {"cats", "dog", "sand", "and", "cat", "sando", "g"};
 cout << "测试用例 4:\n";
 printResult(s4, wordDict4, wordBreak(s4, wordDict4));

 return 0;
}
=====
```

```
=====
package class073;

import java.util.HashSet;
import java.util.List;
import java.util.Set;

// LeetCode 139. 单词拆分
// 题目描述：给你一个字符串 s 和一个字符串列表 wordDict 作为字典。请你判断是否可以利用字典中出现的单词拼接出 s 。
// 注意：不要求字典中出现的单词全部都使用，并且字典中的单词可以重复使用。
// 链接: https://leetcode.cn/problems/word-break/
//
// 解题思路：
// 这是一个完全背包问题的变种。我们可以将字符串 s 看作是背包，将字典中的单词看作是物品。
// 问题转化为：是否可以从字典中选择一些单词（可以重复选择），使得它们的拼接恰好等于字符串 s。
//
// 状态定义：dp[i] 表示字符串 s 的前 i 个字符是否可以被拆分成字典中的单词
// 状态转移方程：对于每个 i，我们检查所有 j < i，如果 dp[j] 为 true 且 s.substring(j, i) 在字典中，则 dp[i] 为 true
// 初始状态：dp[0] = true，表示空字符串可以被拆分
//
// 时间复杂度：O(n^3)，其中 n 是字符串 s 的长度
// 空间复杂度：O(n + m)，其中 m 是字典中所有单词的字符总数

public class Code38_WordBreak {

 // 主方法，用于测试
 public static void main(String[] args) {
 // 测试用例 1
 String s1 = "leetcode";
 List<String> wordDict1 = List.of("leet", "code");
 System.out.println("测试用例 1 结果：" + wordBreak(s1, wordDict1)); // 预期输出: true

 // 测试用例 2
 String s2 = "applepenapple";
 List<String> wordDict2 = List.of("apple", "pen");
 System.out.println("测试用例 2 结果：" + wordBreak(s2, wordDict2)); // 预期输出: true

 // 测试用例 3
 String s3 = "catsandog";
 List<String> wordDict3 = List.of("cats", "dog", "sand", "and", "cat");
 System.out.println("测试用例 3 结果：" + wordBreak(s3, wordDict3)); // 预期输出: false
 }
}
```

```

}

/**
 * 判断字符串是否可以被拆分成字典中的单词
 * @param s 字符串
 * @param wordDict 字典
 * @return 是否可以拆分
 */
public static boolean wordBreak(String s, List<String> wordDict) {
 if (s == null || s.isEmpty()) {
 return false;
 }

 // 将字典转换为集合，提高查找效率
 Set<String> wordSet = new HashSet<>(wordDict);
 int n = s.length();

 // 创建 DP 数组，dp[i] 表示字符串 s 的前 i 个字符是否可以被拆分成字典中的单词
 boolean[] dp = new boolean[n + 1];

 // 初始状态：空字符串可以被拆分
 dp[0] = true;

 // 遍历字符串的每个位置
 for (int i = 1; i <= n; i++) {
 // 遍历所有可能的拆分点 j
 for (int j = 0; j < i; j++) {
 // 如果 dp[j] 为 true (前 j 个字符可以拆分)，且 s.substring(j, i) 在字典中，那么 dp[i]
 // 为 true
 if (dp[j] && wordSet.contains(s.substring(j, i))) {
 dp[i] = true;
 break; // 只要找到一个可行的拆分方式就可以提前结束内层循环
 }
 }
 }

 return dp[n];
}

/**
 * 优化的版本，限制 j 的范围为最大单词长度，避免不必要的检查
 * @param s 字符串
 * @param wordDict 字典

```

```

* @return 是否可以拆分
*/
public static boolean wordBreakOptimized(String s, List<String> wordDict) {
 if (s == null || s.isEmpty()) {
 return false;
 }

 // 将字典转换为集合，提高查找效率
 Set<String> wordSet = new HashSet<>(wordDict);
 int n = s.length();

 // 找出字典中最长单词的长度
 int maxLength = 0;
 for (String word : wordDict) {
 maxLength = Math.max(maxLength, word.length());
 }

 // 创建 DP 数组
 boolean[] dp = new boolean[n + 1];
 dp[0] = true;

 // 遍历字符串的每个位置
 for (int i = 1; i <= n; i++) {
 // 只检查 j >= i - maxLength 的情况，避免不必要的检查
 int start = Math.max(0, i - maxLength);
 for (int j = start; j < i; j++) {
 if (dp[j] && wordSet.contains(s.substring(j, i))) {
 dp[i] = true;
 break;
 }
 }
 }

 return dp[n];
}

/**
 * 使用递归+记忆化搜索实现
 * 这个方法对于较大的输入可能会超时，但展示了递归的思路
 * @param s 字符串
 * @param wordDict 字典
 * @return 是否可以拆分
*/

```

```

public static boolean wordBreakRecursive(String s, List<String> wordDict) {
 if (s == null || s.isEmpty()) {
 return false;
 }

 // 将字典转换为集合
 Set<String> wordSet = new HashSet<>(wordDict);

 // 创建记忆化缓存, memo[i]表示从位置 i 开始的子串是否可以被拆分
 Boolean[] memo = new Boolean[s.length()];

 return dfs(s, 0, wordSet, memo);
}

/**
 * 递归辅助函数
 * @param s 字符串
 * @param start 起始位置
 * @param wordSet 字典集合
 * @param memo 记忆化缓存
 * @return 是否可以拆分
 */
private static boolean dfs(String s, int start, Set<String> wordSet, Boolean[] memo) {
 // 基础情况: 已经处理到字符串末尾
 if (start == s.length()) {
 return true;
 }

 // 检查缓存
 if (memo[start] != null) {
 return memo[start];
 }

 // 尝试所有可能的结束位置
 for (int end = start + 1; end <= s.length(); end++) {
 // 如果 s.substring(start, end) 在字典中, 且剩余部分可以拆分, 则返回 true
 if (wordSet.contains(s.substring(start, end)) && dfs(s, end, wordSet, memo)) {
 memo[start] = true;
 return true;
 }
 }

 // 如果所有可能性都尝试过仍无法拆分, 返回 false
}

```

```

memo[start] = false;
return false;
}

/***
 * 使用 BFS 实现
 * @param s 字符串
 * @param wordDict 字典
 * @return 是否可以拆分
*/
public static boolean wordBreakBFS(String s, List<String> wordDict) {
 if (s == null || s.isEmpty()) {
 return false;
 }

 Set<String> wordSet = new HashSet<>(wordDict);
 boolean[] visited = new boolean[s.length()]; // 记录哪些位置已经被访问过，避免重复处理

 // 使用队列进行 BFS，队列中存储的是当前处理到的位置
 java.util.Queue<Integer> queue = new java.util.LinkedList<>();
 queue.offer(0);
 visited[0] = true;

 while (!queue.isEmpty()) {
 int start = queue.poll();

 // 尝试所有可能的结束位置
 for (int end = start + 1; end <= s.length(); end++) {
 // 如果当前子串在字典中，且结束位置尚未访问过，则继续 BFS
 if (wordSet.contains(s.substring(start, end)) && !visited[end]) {
 if (end == s.length()) {
 return true; // 已经到达字符串末尾，找到了解决方案
 }
 queue.offer(end);
 visited[end] = true;
 }
 }
 }

 return false; // 无法拆分
}

```

文件: Code38\_WordBreak.py

```
LeetCode 139. 单词拆分
题目描述: 给你一个字符串 s 和一个字符串列表 wordDict 作为字典。请你判断是否可以利用字典中出现的单词拼接出 s 。
注意: 不要求字典中出现的单词全部都使用, 并且字典中的单词可以重复使用。
链接: https://leetcode.cn/problems/word-break/
#
解题思路:
这是一个完全背包问题的变种。我们可以将字符串 s 看作是背包, 将字典中的单词看作是物品。
问题转化为: 是否可以从字典中选择一些单词 (可以重复选择), 使得它们的拼接恰好等于字符串 s。
#
状态定义: dp[i] 表示字符串 s 的前 i 个字符是否可以被拆分成字典中的单词
状态转移方程: 对于每个 i, 我们检查所有 j < i, 如果 dp[j] 为 True 且 s[j:i] 在字典中, 则 dp[i] 为 True
初始状态: dp[0] = True, 表示空字符串可以被拆分
#
时间复杂度: O(n^3), 其中 n 是字符串 s 的长度
空间复杂度: O(n + m), 其中 m 是字典中所有单词的字符总数
```

```
def word_break(s: str, word_dict: list[str]) -> bool:
```

```
 """

```

判断字符串是否可以被拆分成字典中的单词

参数:

s: 要检查的字符串

word\_dict: 单词字典

返回:

是否可以拆分

```
 """

```

```
if not s:
```

```
 return False
```

```
将字典转换为集合, 提高查找效率
```

```
word_set = set(word_dict)
```

```
n = len(s)
```

```
创建 DP 数组, dp[i] 表示字符串 s 的前 i 个字符是否可以被拆分成字典中的单词
```

```
dp = [False] * (n + 1)
```

```
初始状态: 空字符串可以被拆分
```

```

dp[0] = True

遍历字符串的每个位置
for i in range(1, n + 1):
 # 遍历所有可能的拆分点 j
 for j in range(0, i):
 # 如果 dp[j] 为 True (前 j 个字符可以拆分), 且 s[j:i] 在字典中, 那么 dp[i] 为 True
 if dp[j] and s[j:i] in word_set:
 dp[i] = True
 break # 只要找到一个可行的拆分方式就可以提前结束内层循环

return dp[n]

```

```

def word_break_optimized(s: str, word_dict: list[str]) -> bool:
 """

```

优化的版本, 限制 j 的范围为最大单词长度, 避免不必要的检查

参数:

s: 要检查的字符串  
word\_dict: 单词字典

返回:

是否可以拆分

```

"""

```

```

if not s:
 return False

```

# 将字典转换为集合, 提高查找效率

```

word_set = set(word_dict)

```

```

n = len(s)

```

# 找出字典中最长单词的长度

```

max_length = 0

```

```

for word in word_dict:

```

```

 max_length = max(max_length, len(word))

```

# 创建 DP 数组

```

dp = [False] * (n + 1)

```

```

dp[0] = True

```

# 遍历字符串的每个位置

```

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

```

```

 # 只检查 $j \geq i - \text{max_length}$ 的情况, 避免不必要的检查

```

```
start = max(0, i - max_length)
for j in range(start, i):
 if dp[j] and s[j:i] in word_set:
 dp[i] = True
 break

return dp[n]

from functools import lru_cache

def word_break_recursive(s: str, word_dict: list[str]) -> bool:
 """
 使用递归+记忆化搜索实现

```

参数:

s: 要检查的字符串  
word\_dict: 单词字典

返回:

是否可以拆分

"""
if not s:
 return False

# 将字典转换为集合

```
word_set = set(word_dict)
n = len(s)
```

```
@lru_cache(maxsize=None)
def dfs(start: int) -> bool:
 """
 递归辅助函数

```

参数:

start: 起始位置

返回:

从 start 位置开始的子串是否可以被拆分

"""
# 基础情况: 已经处理到字符串末尾
if start == n:
 return True

```

尝试所有可能的结束位置
for end in range(start + 1, n + 1):
 # 如果 s[start:end] 在字典中，且剩余部分可以拆分，则返回 True
 if s[start:end] in word_set and dfs(end):
 return True

如果所有可能性都尝试过仍无法拆分，返回 False
return False

return dfs(0)

```

```
def word_break_bfs(s: str, word_dict: list[str]) -> bool:
```

```
"""
使用 BFS 实现
```

参数:

s: 要检查的字符串

word\_dict: 单词字典

返回:

是否可以拆分

```
"""
if not s:
 return False
```

```
word_set = set(word_dict)
```

```
n = len(s)
```

```
visited = [False] * n # 记录哪些位置已经被访问过，避免重复处理
```

# 使用队列进行 BFS，队列中存储的是当前处理到的位置

```
from collections import deque
```

```
queue = deque([0])
```

```
visited[0] = True
```

```
while queue:
```

```
 start = queue.popleft()
```

# 尝试所有可能的结束位置

```
for end in range(start + 1, n + 1):
```

# 如果当前子串在字典中，且结束位置尚未访问过，则继续 BFS

```
if s[start:end] in word_set and not visited[end]:
```

```
 if end == n:
```

```
 return True # 已经到达字符串末尾，找到了解决方案
```

```
queue.append(end)
visited[end] = True

return False # 无法拆分
```

```
def word_break_prefix_tree(s: str, word_dict: list[str]) -> bool:
 """
```

使用前缀树（字典树）优化实现

参数：

s：要检查的字符串

word\_dict：单词字典

返回：

是否可以拆分

```
"""
```

```
if not s:
```

```
 return False
```

# 构建前缀树

```
class TrieNode:
```

```
 def __init__(self):
```

```
 self.children = {}
```

```
 self.is_end_of_word = False
```

```
root = TrieNode()
```

```
for word in word_dict:
```

```
 node = root
```

```
 for char in word:
```

```
 if char not in node.children:
```

```
 node.children[char] = TrieNode()
```

```
 node = node.children[char]
```

```
 node.is_end_of_word = True
```

```
n = len(s)
```

```
dp = [False] * (n + 1)
```

```
dp[0] = True
```

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

```
 if not dp[i]:
```

```
 continue
```

# 从当前位置开始查找字典树



```
word_dict5 =
["a", "aa", "aaa", "aaaa", "aaaaa", "aaaaaa", "aaaaaaaa", "aaaaaaaaa", "aaaaaaaaaa"]
print(f"\n 测试用例 5 结果: {word_break_optimized(s5, word_dict5)}") # 预期输出: False
```

=====

文件: Code39\_WordBreakII.cpp

=====

```
// LeetCode 140. 单词拆分 II
// 题目描述: 给定一个字符串 s 和一个字符串字典 wordDict , 在字符串 s 中增加空格来构建一个句子，使得句子中所有的单词都在词典中。返回所有这样的可能句子。
// 链接: https://leetcode.cn/problems/word-break-ii/

// 解题思路:
// 这是一个完全背包问题的变种，同时也是一个组合问题。我们需要找到所有可能的单词组合，使得它们的拼接等于字符串 s。

// 我们可以使用递归+记忆化搜索来解决这个问题：
// 1. 使用记忆化缓存，避免重复计算
// 2. 对于每个位置 i，我们尝试所有可能的单词，如果 s.substr(i, j-i) 在字典中，我们递归处理剩余部分
// 3. 将当前单词与剩余部分的结果组合

// 时间复杂度: O(n^2 * 2^n)，其中 n 是字符串 s 的长度。在最坏情况下，每个字符之间都可以拆分，会产生 2^(n-1) 种拆分方式。
// 空间复杂度: O(n^2)，用于存储记忆化缓存。
```

```
#include <iostream>
#include <vector>
#include <string>
#include <unordered_set>
#include <unordered_map>
#include <algorithm>
using namespace std;

/**
 * 使用动态规划检查字符串是否可以拆分
 * @param s 字符串
 * @param wordSet 字典集合
 * @return 是否可以拆分
 */
bool canBreak(const string& s, const unordered_set<string>& wordSet) {
 int n = s.length();
 vector<bool> dp(n + 1, false);
 dp[0] = true;
 for (int i = 0; i < n; ++i) {
 for (int j = i + 1; j <= n; ++j) {
 if (dp[i] && wordSet.find(s.substr(i, j - i)) != wordSet.end()) {
 dp[j] = true;
 }
 }
 }
 return dp[n];
}
```

```

dp[0] = true; // 空字符串可以被拆分

// 找出字典中最长单词的长度
int maxLength = 0;
for (const string& word : wordSet) {
 maxLength = max(maxLength, (int)word.length());
}

for (int i = 1; i <= n; i++) {
 // 只检查 j >= i - maxLength 的情况，避免不必要的检查
 int start = max(0, i - maxLength);
 for (int j = start; j < i; j++) {
 if (dp[j] && wordSet.find(s.substr(j, i - j)) != wordSet.end()) {
 dp[i] = true;
 break;
 }
 }
}

return dp[n];
}

```

```

/**
 * 递归辅助函数，使用记忆化搜索找出所有可能的拆分方案
 * @param s 字符串
 * @param start 起始位置
 * @param wordSet 字典集合
 * @param memo 记忆化缓存
 * @return 从 start 位置开始的子串的所有可能拆分方案
 */

```

```

vector<string> dfs(const string& s, int start, const unordered_set<string>& wordSet,
 unordered_map<int, vector<string>>& memo) {
 // 如果已经计算过，直接返回缓存的结果
 if (memo.find(start) != memo.end()) {
 return memo[start];
 }

 vector<string> result;
 int n = s.length();

 // 基础情况：已经处理到字符串末尾
 if (start == n) {
 result.push_back("");
 // 添加空字符串作为递归终止条件
 }
}
```

```

 return result;
 }

// 尝试所有可能的结束位置
for (int end = start + 1; end <= n; end++) {
 // 获取当前子串
 string word = s.substr(start, end - start);

 // 如果当前子串在字典中，递归处理剩余部分
 if (wordSet.find(word) != wordSet.end()) {
 // 递归获取剩余部分的所有拆分方案
 vector<string> subList = dfs(s, end, wordSet, memo);

 // 将当前单词与剩余部分的拆分方案组合
 for (const string& sub : subList) {
 // 如果 sub 为空字符串，说明已经到达字符串末尾，不需要添加空格
 if (sub.empty()) {
 result.push_back(word);
 } else {
 result.push_back(word + " " + sub);
 }
 }
 }
}

// 缓存结果
memo[start] = result;
return result;
}

/***
 * 返回所有可能的单词拆分方案
 * @param s 字符串
 * @param wordDict 字典
 * @return 所有可能的拆分方案列表
 */
vector<string> wordBreak(string s, vector<string>& wordDict) {
 if (s.empty() || wordDict.empty()) {
 return {};
 }

 // 将字典转换为集合，提高查找效率
 unordered_set<string> wordSet(wordDict.begin(), wordDict.end());

```

```

// 首先使用动态规划检查是否可以拆分，如果不能拆分直接返回空列表
// 这一步可以避免不必要的递归计算
if (!canBreak(s, wordSet)) {
 return {};
}

// 创建记忆化缓存，memo[i]表示从位置 i 开始的子串的所有可能拆分方案
unordered_map<int, vector<string>> memo;

return dfs(s, 0, wordSet, memo);
}

/***
 * 另一种实现方式，使用动态规划来存储所有可能的拆分方案
 * @param s 字符串
 * @param wordDict 字典
 * @return 所有可能的拆分方案列表
 */
vector<string> wordBreakDP(string s, vector<string>& wordDict) {
 if (s.empty() || wordDict.empty()) {
 return {};
 }

 // 将字典转换为集合，提高查找效率
 unordered_set<string> wordSet(wordDict.begin(), wordDict.end());
 int n = s.length();

 // dp[i]存储前 i 个字符的所有可能拆分方案
 vector<vector<string>> dp(n + 1);

 // 初始状态：空字符串有一个拆分方案（空字符串）
 dp[0].push_back("");

 // 填充 dp 数组
 for (int i = 1; i <= n; i++) {
 for (int j = 0; j < i; j++) {
 string word = s.substr(j, i - j);
 if (wordSet.find(word) != wordSet.end() && !dp[j].empty()) {
 // 将当前单词与前 j 个字符的所有拆分方案组合
 for (const string& prev : dp[j]) {
 if (prev.empty()) {
 dp[i].push_back(word);
 }
 }
 }
 }
 }
}

```

```

 } else {
 dp[i].push_back(prev + " " + word);
 }
 }
}

return dp[n];
}

/***
 * 优化的递归辅助函数，使用最大单词长度来限制搜索范围
 * @param s 字符串
 * @param start 起始位置
 * @param wordSet 字典集合
 * @param memo 记忆化缓存
 * @param maxLength 字典中最长单词的长度
 * @return 从 start 位置开始的子串的所有可能拆分方案
 */
vector<string> dfsOptimized(const string& s, int start, const unordered_set<string>& wordSet,
 unordered_map<int, vector<string>>& memo, int maxLength) {
 // 如果已经计算过，直接返回缓存的结果
 if (memo.find(start) != memo.end()) {
 return memo[start];
 }

 vector<string> result;
 int n = s.length();

 // 基础情况：已经处理到字符串末尾
 if (start == n) {
 result.push_back("");
 return result;
 }

 // 限制 end 的范围为 start + maxLength，避免不必要的检查
 int endMax = min(start + maxLength, n);
 for (int end = start + 1; end <= endMax; end++) {
 string word = s.substr(start, end - start);

 if (wordSet.find(word) != wordSet.end()) {
 vector<string> subList = dfsOptimized(s, end, wordSet, memo, maxLength);

```

```

 for (const string& sub : subList) {
 if (sub.empty()) {
 result.push_back(word);
 } else {
 result.push_back(word + " " + sub);
 }
 }
 }

// 缓存结果
memo[start] = result;
return result;
}

/***
 * 优化的 DFS 实现，使用最大单词长度来限制搜索范围
 * @param s 字符串
 * @param wordDict 字典
 * @return 所有可能的拆分方案列表
 */
vector<string> wordBreakOptimized(string s, vector<string>& wordDict) {
 if (s.empty() || wordDict.empty()) {
 return {};
 }

 // 将字典转换为集合，提高查找效率
 unordered_set<string> wordSet(wordDict.begin(), wordDict.end());

 // 找出字典中最长单词的长度
 int maxLength = 0;
 for (const string& word : wordDict) {
 maxLength = max(maxLength, (int)word.length());
 }

 // 首先检查是否可以拆分
 if (!canBreak(s, wordSet)) {
 return {};
 }

 // 创建记忆化缓存
 unordered_map<int, vector<string>> memo;

```

```

return dfsOptimized(s, 0, wordSet, memo, maxLength);
}

// 打印测试结果
void printResult(const string& s, const vector<string>& wordDict, const vector<string>& result) {
 cout << "字符串: " << s << "\n";
 cout << "字典: [";
 for (size_t i = 0; i < wordDict.size(); i++) {
 cout << "\\" << wordDict[i] << "\\";
 if (i < wordDict.size() - 1) {
 cout << ", ";
 }
 }
 cout << "]\n";

 cout << "结果: [";
 for (size_t i = 0; i < result.size(); i++) {
 cout << "\\" << result[i] << "\\";
 if (i < result.size() - 1) {
 cout << ", ";
 }
 }
 cout << "]\n";
 cout << "-----\n";
}

// 主函数，用于测试
int main() {
 // 测试用例 1
 string s1 = "catsanddog";
 vector<string> wordDict1 = {"cat", "cats", "and", "sand", "dog"};
 cout << "测试用例 1:\n";
 printResult(s1, wordDict1, wordBreak(s1, wordDict1));

 // 测试用例 2
 string s2 = "pineapplepenapple";
 vector<string> wordDict2 = {"apple", "pen", "applepen", "pine", "pineapple"};
 cout << "测试用例 2:\n";
 printResult(s2, wordDict2, wordBreak(s2, wordDict2));

 // 测试用例 3
 string s3 = "catsandog";
}

```

```

vector<string> wordDict3 = {"cats", "dog", "sand", "and", "cat"};
cout << "测试用例 3:\n";
printResult(s3, wordDict3, wordBreak(s3, wordDict3));

// 测试不同实现
cout << "测试不同实现:\n";
cout << "动态规划版本 (测试用例 1):\n";
printResult(s1, wordDict1, wordBreakDP(s1, wordDict1));

cout << "优化 DFS 版本 (测试用例 1):\n";
printResult(s1, wordDict1, wordBreakOptimized(s1, wordDict1));

return 0;
}
=====
```

文件: Code39\_WordBreakII.java

```
=====
```

```

package class073;

import java.util.*;

// LeetCode 140. 单词拆分 II
// 题目描述: 给定一个字符串 s 和一个字符串字典 wordDict , 在字符串 s 中增加空格来构建一个句子，使得句子中所有的单词都在词典中。返回所有这样的可能句子。
// 链接: https://leetcode.cn/problems/word-break-ii/
//
// 解题思路:
// 这是一个完全背包问题的变种，同时也是一个组合问题。我们需要找到所有可能的单词组合，使得它们的拼接等于字符串 s。
//
// 我们可以使用递归+记忆化搜索来解决这个问题:
// 1. 使用记忆化缓存，避免重复计算
// 2. 对于每个位置 i，我们尝试所有可能的单词，如果 s.substring(i, j) 在字典中，我们递归处理剩余部分
// 3. 将当前单词与剩余部分的结果组合
//
// 时间复杂度: O(n^2 * 2^n)，其中 n 是字符串 s 的长度。在最坏情况下，每个字符之间都可以拆分，会产生 2^(n-1) 种拆分方式。
// 空间复杂度: O(n^2)，用于存储记忆化缓存。

public class Code39_WordBreakII {
```

```

// 主方法，用于测试
public static void main(String[] args) {
 // 测试用例 1
 String s1 = "catsanddog";
 List<String> wordDict1 = Arrays.asList("cat", "cats", "and", "sand", "dog");
 System.out.println("测试用例 1 结果: " + wordBreak(s1, wordDict1));
 // 预期输出: ["cats and dog", "cat sand dog"]

 // 测试用例 2
 String s2 = "pineapplepenapple";
 List<String> wordDict2 = Arrays.asList("apple", "pen", "applepen", "pine", "pineapple");
 System.out.println("测试用例 2 结果: " + wordBreak(s2, wordDict2));
 // 预期输出: ["pine apple pen apple", "pineapple pen apple", "pine applepen apple"]

 // 测试用例 3
 String s3 = "catsandog";
 List<String> wordDict3 = Arrays.asList("cats", "dog", "sand", "and", "cat");
 System.out.println("测试用例 3 结果: " + wordBreak(s3, wordDict3));
 // 预期输出: []

}

/**
 * 返回所有可能的单词拆分方案
 * @param s 字符串
 * @param wordDict 字典
 * @return 所有可能的拆分方案列表
 */
public static List<String> wordBreak(String s, List<String> wordDict) {
 if (s == null || s.isEmpty() || wordDict == null || wordDict.isEmpty()) {
 return new ArrayList<>();
 }

 // 将字典转换为集合，提高查找效率
 Set<String> wordSet = new HashSet<>(wordDict);

 // 首先使用动态规划检查是否可以拆分，如果不能拆分直接返回空列表
 // 这一步可以避免不必要的递归计算
 if (!canBreak(s, wordSet)) {
 return new ArrayList<>();
 }

 // 创建记忆化缓存，memo[i]表示从位置 i 开始的子串的所有可能拆分方案
 Map<Integer, List<String>> memo = new HashMap<>();

```

```

 return dfs(s, 0, wordSet, memo);
 }

/***
 * 使用动态规划检查字符串是否可以拆分
 * @param s 字符串
 * @param wordSet 字典集合
 * @return 是否可以拆分
 */
private static boolean canBreak(String s, Set<String> wordSet) {
 int n = s.length();
 boolean[] dp = new boolean[n + 1];
 dp[0] = true; // 空字符串可以被拆分

 // 找出字典中最长单词的长度
 int maxLength = 0;
 for (String word : wordSet) {
 maxLength = Math.max(maxLength, word.length());
 }

 for (int i = 1; i <= n; i++) {
 // 只检查 j >= i - maxLength 的情况，避免不必要的检查
 int start = Math.max(0, i - maxLength);
 for (int j = start; j < i; j++) {
 if (dp[j] && wordSet.contains(s.substring(j, i))) {
 dp[i] = true;
 break;
 }
 }
 }

 return dp[n];
}

/***
 * 递归辅助函数，使用记忆化搜索找出所有可能的拆分方案
 * @param s 字符串
 * @param start 起始位置
 * @param wordSet 字典集合
 * @param memo 记忆化缓存
 * @return 从 start 位置开始的子串的所有可能拆分方案
 */

```

```
private static List<String> dfs(String s, int start, Set<String> wordSet, Map<Integer, List<String>> memo) {
 // 如果已经计算过，直接返回缓存的结果
 if (memo.containsKey(start)) {
 return memo.get(start);
 }

 List<String> result = new ArrayList<>();
 int n = s.length();

 // 基础情况：已经处理到字符串末尾
 if (start == n) {
 result.add("");
 // 添加空字符串作为递归终止条件
 return result;
 }

 // 尝试所有可能的结束位置
 for (int end = start + 1; end <= n; end++) {
 // 获取当前子串
 String word = s.substring(start, end);

 // 如果当前子串在字典中，递归处理剩余部分
 if (wordSet.contains(word)) {
 // 递归获取剩余部分的所有拆分方案
 List<String> subList = dfs(s, end, wordSet, memo);

 // 将当前单词与剩余部分的拆分方案组合
 for (String sub : subList) {
 // 如果 sub 为空字符串，说明已经到达字符串末尾，不需要添加空格
 if (sub.isEmpty()) {
 result.add(word);
 } else {
 result.add(word + " " + sub);
 }
 }
 }
 }

 // 缓存结果
 memo.put(start, result);
 return result;
}
```

```

/**
 * 另一种实现方式，使用动态规划来存储所有可能的拆分方案
 * @param s 字符串
 * @param wordDict 字典
 * @return 所有可能的拆分方案列表
 */
public static List<String> wordBreakDP(String s, List<String> wordDict) {
 if (s == null || s.isEmpty() || wordDict == null || wordDict.isEmpty()) {
 return new ArrayList<>();
 }

 // 将字典转换为集合，提高查找效率
 Set<String> wordSet = new HashSet<>(wordDict);
 int n = s.length();

 // dp[i]存储前 i 个字符的所有可能拆分方案
 List<List<String>> dp = new ArrayList<>();
 for (int i = 0; i <= n; i++) {
 dp.add(new ArrayList<>());
 }

 // 初始状态：空字符串有一个拆分方案（空字符串）
 dp.get(0).add("");

 // 填充 dp 数组
 for (int i = 1; i <= n; i++) {
 for (int j = 0; j < i; j++) {
 String word = s.substring(j, i);
 if (wordSet.contains(word) && !dp.get(j).isEmpty()) {
 // 将当前单词与前 j 个字符的所有拆分方案组合
 for (String prev : dp.get(j)) {
 if (prev.isEmpty()) {
 dp.get(i).add(word);
 } else {
 dp.get(i).add(prev + " " + word);
 }
 }
 }
 }
 }

 return dp.get(n);
}

```

```

/**
 * 优化的 DFS 实现，使用最大单词长度来限制搜索范围
 * @param s 字符串
 * @param wordDict 字典
 * @return 所有可能的拆分方案列表
 */
public static List<String> wordBreakOptimized(String s, List<String> wordDict) {
 if (s == null || s.isEmpty() || wordDict == null || wordDict.isEmpty()) {
 return new ArrayList<>();
 }

 // 将字典转换为集合，提高查找效率
 Set<String> wordSet = new HashSet<>(wordDict);

 // 找出字典中最长单词的长度
 int maxLength = 0;
 for (String word : wordDict) {
 maxLength = Math.max(maxLength, word.length());
 }

 // 首先检查是否可以拆分
 if (!canBreak(s, wordSet)) {
 return new ArrayList<>();
 }

 // 创建记忆化缓存
 Map<Integer, List<String>> memo = new HashMap<>();

 return dfsOptimized(s, 0, wordSet, memo, maxLength);
}

/**
 * 优化的递归辅助函数，使用最大单词长度来限制搜索范围
 * @param s 字符串
 * @param start 起始位置
 * @param wordSet 字典集合
 * @param memo 记忆化缓存
 * @param maxLength 字典中最长单词的长度
 * @return 从 start 位置开始的子串的所有可能拆分方案
 */
private static List<String> dfsOptimized(String s, int start, Set<String> wordSet,
 Map<Integer, List<String>> memo, int maxLength) {

```

```

// 如果已经计算过，直接返回缓存的结果
if (memo.containsKey(start)) {
 return memo.get(start);
}

List<String> result = new ArrayList<>();
int n = s.length();

// 基础情况：已经处理到字符串末尾
if (start == n) {
 result.add("");
 return result;
}

// 限制 end 的范围为 start + maxLength，避免不必要的检查
int endMax = Math.min(start + maxLength, n);
for (int end = start + 1; end <= endMax; end++) {
 String word = s.substring(start, end);

 if (wordSet.contains(word)) {
 List<String> subList = dfsOptimized(s, end, wordSet, memo, maxLength);

 for (String sub : subList) {
 if (sub.isEmpty()) {
 result.add(word);
 } else {
 result.add(word + " " + sub);
 }
 }
 }
}

// 缓存结果
memo.put(start, result);
return result;
}

```

=====

文件: Code39\_WordBreakII.py

=====

# LeetCode 140. 单词拆分 II

```
题目描述：给定一个字符串 s 和一个字符串字典 wordDict ，在字符串 s 中增加空格来构建一个句子，使得句子中所有的单词都在词典中。返回所有这样的可能句子。
链接: https://leetcode.cn/problems/word-break-ii/
#
解题思路：
这是一个完全背包问题的变种，同时也是一个组合问题。我们需要找到所有可能的单词组合，使得它们的拼接等于字符串 s。
#
我们可以使用递归+记忆化搜索来解决这个问题：
1. 使用记忆化缓存，避免重复计算
2. 对于每个位置 i，我们尝试所有可能的单词，如果 s[i:j] 在字典中，我们递归处理剩余部分
3. 将当前单词与剩余部分的结果组合
#
时间复杂度：O(n^2 * 2^n)，其中 n 是字符串 s 的长度。在最坏情况下，每个字符之间都可以拆分，会产生 2^(n-1) 种拆分方式。
空间复杂度：O(n^2)，用于存储记忆化缓存。
```

```
from functools import lru_cache
from typing import List

def word_break(s: str, word_dict: List[str]) -> List[str]:
 """
 返回所有可能的单词拆分方案

```

参数：

```
s: 字符串
word_dict: 单词字典
```

返回：

```
所有可能的拆分方案列表
"""

if not s or not word_dict:
 return []
```

```
将字典转换为集合，提高查找效率
```

```
word_set = set(word_dict)
```

```
首先使用动态规划检查是否可以拆分，如果不能拆分直接返回空列表
```

```
这一步可以避免不必要的递归计算
```

```
if not can_break(s, word_set):
 return []
```

```
创建记忆化缓存，使用 lru_cache 装饰器
```

```

@lru_cache(maxsize=None)
def dfs(start: int) -> List[str]:
 """
 递归辅助函数，使用记忆化搜索找出所有可能的拆分方案

 参数:
 start: 起始位置

 返回:
 从 start 位置开始的子串的所有可能拆分方案
 """
 # 基础情况：已经处理到字符串末尾
 if start == len(s):
 return [''] # 返回空字符串作为递归终止条件

 result = []

 # 尝试所有可能的结束位置
 for end in range(start + 1, len(s) + 1):
 # 获取当前子串
 word = s[start:end]

 # 如果当前子串在字典中，递归处理剩余部分
 if word in word_set:
 # 递归获取剩余部分的所有拆分方案
 sub_list = dfs(end)

 # 将当前单词与剩余部分的拆分方案组合
 for sub in sub_list:
 # 如果 sub 为空字符串，说明已经到达字符串末尾，不需要添加空格
 if sub:
 result.append(word + ' ' + sub)
 else:
 result.append(word)

 return result

return dfs(0)

def can_break(s: str, word_set: set) -> bool:
 """
 使用动态规划检查字符串是否可以拆分

```

参数:

s: 字符串

word\_set: 字典集合

返回:

是否可以拆分

"""

n = len(s)

dp = [False] \* (n + 1)

dp[0] = True # 空字符串可以被拆分

# 找出字典中最长单词的长度

max\_length = 0

for word in word\_set:

    max\_length = max(max\_length, len(word))

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

    # 只检查  $j \geq i - \text{max\_length}$  的情况, 避免不必要的检查

    start = max(0, i - max\_length)

    for j in range(start, i):

        if dp[j] and s[j:i] in word\_set:

            dp[i] = True

            break

return dp[n]

def word\_break\_dp(s: str, word\_dict: List[str]) -> List[str]:

"""

使用动态规划来存储所有可能的拆分方案

参数:

s: 字符串

word\_dict: 单词字典

返回:

所有可能的拆分方案列表

"""

if not s or not word\_dict:

    return []

# 将字典转换为集合, 提高查找效率

word\_set = set(word\_dict)

n = len(s)

```

dp[i] 存储前 i 个字符的所有可能拆分方案
dp = [[] for _ in range(n + 1)]

初始状态：空字符串有一个拆分方案（空字符串）
dp[0].append('')

填充 dp 数组
for i in range(1, n + 1):
 for j in range(0, i):
 word = s[j:i]
 if word in word_set and dp[j]:
 # 将当前单词与前 j 个字符的所有拆分方案组合
 for prev in dp[j]:
 if prev:
 dp[i].append(prev + ' ' + word)
 else:
 dp[i].append(word)

return dp[n]

```

```

def word_break_optimized(s: str, word_dict: List[str]) -> List[str]:
 """

```

优化的 DFS 实现，使用最大单词长度来限制搜索范围

参数：

s: 字符串  
word\_dict: 单词字典

返回：

所有可能的拆分方案列表

"""

```

if not s or not word_dict:
 return []

```

# 将字典转换为集合，提高查找效率

```
word_set = set(word_dict)
```

# 找出字典中最长单词的长度

```
max_length = 0
for word in word_dict:
 max_length = max(max_length, len(word))
```

```

首先检查是否可以拆分
if not can_break(s, word_set):
 return []

创建记忆化缓存
@lru_cache(maxsize=None)
def dfs(start: int) -> List[str]:
 """
 递归辅助函数，使用记忆化搜索找出所有可能的拆分方案
 """

 # 参数:
 # start: 起始位置

 # 返回:
 # 从 start 位置开始的子串的所有可能拆分方案
 """

 # 基础情况: 已经处理到字符串末尾
 if start == len(s):
 return ['']

 result = []

 # 限制 end 的范围为 start + max_length, 避免不必要的检查
 end_max = min(start + max_length, len(s))
 for end in range(start + 1, end_max + 1):
 word = s[start:end]

 if word in word_set:
 sub_list = dfs(end)

 for sub in sub_list:
 if sub:
 result.append(word + ' ' + sub)
 else:
 result.append(word)

 return result

 return dfs(0)

def word_break_backtracking(s: str, word_dict: List[str]) -> List[str]:
 """
 使用回溯算法实现

```

参数:

s: 字符串

word\_dict: 单词字典

返回:

所有可能的拆分方案列表

"""

```
if not s or not word_dict:
 return []
```

```
word_set = set(word_dict)
result = []
path = []
```

# 首先检查是否可以拆分

```
if not can_break(s, word_set):
 return []
```

```
def backtrack(start: int):
```

"""回溯辅助函数"""

# 如果已经处理到字符串末尾, 将当前路径添加到结果中

```
if start == len(s):
 result.append(' '.join(path))
 return
```

# 尝试所有可能的结束位置

```
for end in range(start + 1, len(s) + 1):
 word = s[start:end]
```

```
 if word in word_set:
 # 选择当前单词
 path.append(word)
 # 递归处理剩余部分
 backtrack(end)
 # 回溯, 撤销选择
 path.pop()
```

```
backtrack(0)
```

```
return result
```

# 测试代码

```
if __name__ == "__main__":
```

```

测试用例 1
s1 = "catsanddog"
word_dict1 = ["cat", "cats", "and", "sand", "dog"]
print(f"测试用例 1 结果: {word_break(s1, word_dict1)}")
预期输出: ["cats and dog", "cat sand dog"]

测试用例 2
s2 = "pineapplepenapple"
word_dict2 = ["apple", "pen", "applepen", "pine", "pineapple"]
print(f"测试用例 2 结果: {word_break(s2, word_dict2)}")
预期输出: ["pine apple pen apple", "pineapple pen apple", "pine applepen apple"]

测试用例 3
s3 = "catsandog"
word_dict3 = ["cats", "dog", "sand", "and", "cat"]
print(f"测试用例 3 结果: {word_break(s3, word_dict3)}")
预期输出: []

测试不同实现
print("\n 测试不同实现:")
print(f"动态规划版本 (测试用例 1): {word_break_dp(s1, word_dict1)}")
print(f"优化 DFS 版本 (测试用例 1): {word_break_optimized(s1, word_dict1)}")
print(f"回溯算法版本 (测试用例 1): {word_break_backtracking(s1, word_dict1)}")

测试用例 4
s4 = "a"
word_dict4 = ["a"]
print(f"\n 测试用例 4 结果: {word_break(s4, word_dict4)}")
预期输出: ["a"]

测试用例 5
s5 = "aaaaaaaa"
word_dict5 = ["a", "aa", "aaa", "aaaa", "aaaaa", "aaaaaa", "aaaaaaa", "aaaaaaaa"]
print(f"\n 测试用例 5 结果: {word_break_optimized(s5, word_dict5)}")
应该输出所有可能的拆分方案
=====

文件: Code40_PerfectSquares.cpp
=====

#include <iostream>
#include <vector>
#include <queue>

```

文件: Code40\_PerfectSquares.cpp

```

=====

#include <iostream>
#include <vector>
#include <queue>
```

```

#include <cmath>
#include <climits>

// LeetCode 279. 完全平方数
// 题目描述：给你一个整数 n，返回和为 n 的完全平方数的最少数量。
// 完全平方数是一个整数，其值等于另一个整数的平方；换句话说，其值等于一个整数自乘的积。例如，1、4、9 和 16 都是完全平方数，而 3 和 11 不是。
// 链接：https://leetcode.cn/problems/perfect-squares/
//
// 解题思路：
// 这是一个完全背包问题。我们可以将问题转化为：使用最少量的物品（每个物品是一个完全平方数），恰好装满容量为 n 的背包。
//
// 状态定义：dp[i] 表示和为 i 的完全平方数的最少数量
// 状态转移方程：dp[i] = min(dp[i], dp[i - j * j] + 1)，其中 $j * j \leq i$
// 初始状态：dp[0] = 0，表示和为 0 的完全平方数的最少数量为 0
//
// 时间复杂度：O(n * sqrt(n))，其中 n 是给定的整数
// 空间复杂度：O(n)，使用一维 DP 数组

class Solution {
public:
 // 基础 DP 解法
 int numSquares(int n) {
 if (n < 1) {
 return 0;
 }

 // 创建 DP 数组，dp[i] 表示和为 i 的完全平方数的最少数量
 std::vector<int> dp(n + 1, 0);

 // 初始化 DP 数组，初始值设为最大可能值（即全部用 1 相加）
 for (int i = 1; i <= n; ++i) {
 dp[i] = i; // 最坏情况下，i 可以由 i 个 1 组成
 }

 // 填充 DP 数组
 for (int i = 2; i <= n; ++i) {
 // 尝试所有可能的完全平方数 j^2 ，其中 $j^2 \leq i$
 for (int j = 1; j * j <= i; ++j) {
 // 更新状态：选择当前完全平方数 j^2 ，那么问题转化为求 $dp[i - j * j] + 1$
 dp[i] = std::min(dp[i], dp[i - j * j] + 1);
 }
 }
 }
};

```

```

 }

 return dp[n];
}

// 优化版本，预生成所有可能的完全平方数
int numSquaresOptimized(int n) {
 if (n < 1) {
 return 0;
 }

 // 预生成所有可能的完全平方数
 int maxSquareRoot = std::sqrt(n);
 std::vector<int> squares;
 squares.reserve(maxSquareRoot);
 for (int i = 1; i <= maxSquareRoot; ++i) {
 squares.push_back(i * i);
 }

 // 创建 DP 数组
 std::vector<int> dp(n + 1, 0);

 // 初始化 DP 数组
 for (int i = 1; i <= n; ++i) {
 dp[i] = i;
 }

 // 填充 DP 数组
 for (int i = 2; i <= n; ++i) {
 // 尝试所有可能的完全平方数
 for (int square : squares) {
 if (square > i) {
 break; // 由于 squares 是递增的，当 square > i 时，后面的平方数也都大于 i，直接
break
 }
 dp[i] = std::min(dp[i], dp[i - square] + 1);
 }
 }

 return dp[n];
}

// 使用广度优先搜索(BFS)实现

```

```

int numSquaresBFS(int n) {
 if (n < 1) {
 return 0;
 }

 // 预先生成所有可能的完全平方数
 int maxSquareRoot = std::sqrt(n);
 std::vector<int> squares;
 squares.reserve(maxSquareRoot);
 for (int i = 1; i <= maxSquareRoot; ++i) {
 squares.push_back(i * i);
 }

 // 使用队列进行 BFS
 std::queue<int> q;
 std::vector<bool> visited(n + 1, false); // 记录哪些数字已经访问过，避免重复处理

 q.push(0); // 从 0 开始
 visited[0] = true;
 int level = 0; // 当前的层数，即使用的完全平方数的数量

 while (!q.empty()) {
 level++;
 int size = q.size();

 // 处理当前层的所有节点
 for (int i = 0; i < size; ++i) {
 int current = q.front();
 q.pop();

 // 尝试所有可能的完全平方数
 for (int square : squares) {
 int next = current + square;

 if (next == n) {
 return level; // 找到目标值，返回当前层数
 }

 if (next > n || visited[next]) {
 continue; // 超过目标值或者已经访问过，跳过
 }

 visited[next] = true;
 }
 }
 }
}

```

```

 q.push(next);
 }
}

}

return n; // 默认返回 n (实际上不应该到达这里)
}

// 使用数学方法优化，基于拉格朗日四平方定理
int numSquaresMath(int n) {
 if (n < 1) {
 return 0;
 }

 // 如果 n 是完全平方数，直接返回 1
 if (isPerfectSquare(n)) {
 return 1;
 }

 // 检查是否可以表示为两个完全平方数的和
 if (canBeExpressedAsSumOfTwoSquares(n)) {
 return 2;
 }

 // 检查是否可以表示为三个完全平方数的和
 // 根据 Legendre 三平方定理，如果 n 不是形如 $4^k \cdot (8m+7)$ ，则可以表示为三个平方数的和
 int temp = n;
 while (temp % 4 == 0) {
 temp /= 4;
 }
 if (temp % 8 != 7) {
 return 3;
 }

 // 根据四平方定理，所有自然数都可以表示为 4 个平方数的和
 return 4;
}

private:
 /**
 * 判断一个数是否是完全平方数
 */
 bool isPerfectSquare(int num) {

```

```

 int sqrtNum = std::sqrt(num);
 return sqrtNum * sqrtNum == num;
}

/***
 * 判断一个数是否可以表示为两个完全平方数的和
 */
bool canBeExpressedAsSumOfTwoSquares(int num) {
 for (int i = 0; i * i <= num; ++i) {
 int remainder = num - i * i;
 if (isPerfectSquare(remainder)) {
 return true;
 }
 }
 return false;
};

int main() {
 Solution solution;

 // 测试用例 1
 int n1 = 12;
 std::cout << "测试用例 1 结果: " << solution.numSquares(n1) << " (预期: 3)" << std::endl;
 std::cout << "优化版本结果: " << solution.numSquaresOptimized(n1) << std::endl;
 std::cout << "BFS 版本结果: " << solution.numSquaresBFS(n1) << std::endl;
 std::cout << "数学优化版本结果: " << solution.numSquaresMath(n1) << std::endl;

 std::cout << "-----" << std::endl;

 // 测试用例 2
 int n2 = 13;
 std::cout << "测试用例 2 结果: " << solution.numSquares(n2) << " (预期: 2)" << std::endl;
 std::cout << "优化版本结果: " << solution.numSquaresOptimized(n2) << std::endl;
 std::cout << "BFS 版本结果: " << solution.numSquaresBFS(n2) << std::endl;
 std::cout << "数学优化版本结果: " << solution.numSquaresMath(n2) << std::endl;

 std::cout << "-----" << std::endl;

 // 测试用例 3
 int n3 = 1;
 std::cout << "测试用例 3 结果: " << solution.numSquares(n3) << " (预期: 1)" << std::endl;
}

```

```
 std::cout << "-----" << std::endl;

 // 测试用例 4
 int n4 = 2;
 std::cout << "测试用例 4 结果: " << solution.numSquares(n4) << " (预期: 2)" << std::endl;

 return 0;
}
```

=====

文件: Code40\_PerfectSquares.java

```
package class073;

// LeetCode 279. 完全平方数
// 题目描述: 给你一个整数 n，返回和为 n 的完全平方数的最少数量。
// 完全平方数是一个整数，其值等于另一个整数的平方；换句话说，其值等于一个整数自乘的积。例如，1、4、9 和 16 都是完全平方数，而 3 和 11 不是。
// 链接: https://leetcode.cn/problems/perfect-squares/
//
// 解题思路:
// 这是一个完全背包问题。我们可以将问题转化为：使用最少量的物品（每个物品是一个完全平方数），恰好装满容量为 n 的背包。
//
// 状态定义: dp[i] 表示和为 i 的完全平方数的最少数量
// 状态转移方程: dp[i] = min(dp[i], dp[i - j * j] + 1)，其中 j * j <= i
// 初始状态: dp[0] = 0，表示和为 0 的完全平方数的最少数量为 0
//
// 时间复杂度: O(n * sqrt(n))，其中 n 是给定的整数
// 空间复杂度: O(n)，使用一维 DP 数组
```

```
public class Code40_PerfectSquares {

 // 主方法，用于测试
 public static void main(String[] args) {
 // 测试用例 1
 int n1 = 12;
 System.out.println("测试用例 1 结果: " + numSquares(n1)); // 预期输出: 3 (4 + 4 + 4)

 // 测试用例 2
 int n2 = 13;
 System.out.println("测试用例 2 结果: " + numSquares(n2)); // 预期输出: 2 (4 + 9)
```

```

// 测试用例 3
int n3 = 1;
System.out.println("测试用例 3 结果: " + numSquares(n3)); // 预期输出: 1 (1)

// 测试用例 4
int n4 = 2;
System.out.println("测试用例 4 结果: " + numSquares(n4)); // 预期输出: 2 (1 + 1)
}

/**
 * 找出和为 n 的完全平方数的最少数量
 * @param n 给定的整数
 * @return 最少数量
 */
public static int numSquares(int n) {
 if (n < 1) {
 return 0;
 }

 // 创建 DP 数组, dp[i] 表示和为 i 的完全平方数的最少数量
 int[] dp = new int[n + 1];

 // 初始化 DP 数组, 初始值设为最大可能值 (即全部用 1 相加)
 for (int i = 1; i <= n; i++) {
 dp[i] = i; // 最坏情况下, i 可以由 i 个 1 组成
 }

 // 填充 DP 数组
 for (int i = 2; i <= n; i++) {
 // 尝试所有可能的完全平方数 j^2, 其中 j^2 <= i
 for (int j = 1; j * j <= i; j++) {
 // 更新状态: 选择当前完全平方数 j^2, 那么问题转化为求 dp[i - j^2] + 1
 dp[i] = Math.min(dp[i], dp[i - j * j] + 1);
 }
 }

 return dp[n];
}

/**
 * 优化版本, 预先生成所有可能的完全平方数
 * @param n 给定的整数

```

```

* @return 最少数量
*/
public static int numSquaresOptimized(int n) {
 if (n < 1) {
 return 0;
 }

 // 预先生成所有可能的完全平方数
 int maxSquareRoot = (int) Math.sqrt(n);
 int[] squares = new int[maxSquareRoot];
 for (int i = 0; i < maxSquareRoot; i++) {
 squares[i] = (i + 1) * (i + 1);
 }

 // 创建 DP 数组
 int[] dp = new int[n + 1];

 // 初始化 DP 数组
 for (int i = 1; i <= n; i++) {
 dp[i] = i;
 }

 // 填充 DP 数组
 for (int i = 2; i <= n; i++) {
 // 尝试所有可能的完全平方数
 for (int square : squares) {
 if (square > i) {
 break; // 由于 squares 是递增的, 当 square > i 时, 后面的平方数也都大于 i, 直接
break
 }
 dp[i] = Math.min(dp[i], dp[i - square] + 1);
 }
 }

 return dp[n];
}

/**
* 使用广度优先搜索(BFS)实现
* 我们可以将每个数字看作一个节点, 如果两个数字之间相差一个完全平方数, 那么它们之间有一条边
* 问题转化为: 从 0 出发, 到 n 的最短路径长度
* @param n 给定的整数
* @return 最少数量

```

```
/*
public static int numSquaresBFS(int n) {
 if (n < 1) {
 return 0;
 }

 // 预先生成所有可能的完全平方数
 int maxSquareRoot = (int) Math.sqrt(n);
 int[] squares = new int[maxSquareRoot];
 for (int i = 0; i < maxSquareRoot; i++) {
 squares[i] = (i + 1) * (i + 1);
 }

 // 使用队列进行 BFS
 java.util.Queue<Integer> queue = new java.util.LinkedList<>();
 boolean[] visited = new boolean[n + 1]; // 记录哪些数字已经访问过，避免重复处理

 queue.offer(0); // 从 0 开始
 visited[0] = true;
 int level = 0; // 当前的层数，即使用的完全平方数的数量

 while (!queue.isEmpty()) {
 level++;
 int size = queue.size();

 // 处理当前层的所有节点
 for (int i = 0; i < size; i++) {
 int current = queue.poll();

 // 尝试所有可能的完全平方数
 for (int square : squares) {
 int next = current + square;

 if (next == n) {
 return level; // 找到目标值，返回当前层数
 }

 if (next > n || visited[next]) {
 continue; // 超过目标值或者已经访问过，跳过
 }

 visited[next] = true;
 queue.offer(next);
 }
 }
 }
}
```

```

 }
 }

}

return n; // 默认返回 n (实际上不应该到达这里)
}

/***
 * 使用数学方法优化，基于拉格朗日四平方定理
 * 四平方定理：每个自然数都可以表示为 4 个整数的平方和
 * @param n 给定的整数
 * @return 最少数量
 */
public static int numSquaresMath(int n) {
 if (n < 1) {
 return 0;
 }

 // 如果 n 是完全平方数，直接返回 1
 if (isPerfectSquare(n)) {
 return 1;
 }

 // 检查是否可以表示为两个完全平方数的和
 if (canBeExpressedAsSumOfTwoSquares(n)) {
 return 2;
 }

 // 检查是否可以表示为三个完全平方数的和
 // 根据 Legendre 三平方定理，如果 n 不是形如 $4^k \cdot (8m+7)$ ，则可以表示为三个平方数的和
 int temp = n;
 while (temp % 4 == 0) {
 temp /= 4;
 }
 if (temp % 8 != 7) {
 return 3;
 }

 // 根据四平方定理，所有自然数都可以表示为 4 个平方数的和
 return 4;
}

/***

```

```

* 判断一个数是否是完全平方数
* @param num 要判断的数
* @return 是否是完全平方数
*/
private static boolean isPerfectSquare(int num) {
 int sqrt = (int) Math.sqrt(num);
 return sqrt * sqrt == num;
}

/**
* 判断一个数是否可以表示为两个完全平方数的和
* @param num 要判断的数
* @return 是否可以表示为两个完全平方数的和
*/
private static boolean canBeExpressedAsSumOfTwoSquares(int num) {
 for (int i = 0; i * i <= num; i++) {
 if (isPerfectSquare(num - i * i)) {
 return true;
 }
 }
 return false;
}

```

文件: Code40\_PerfectSquares.py

```

LeetCode 279. 完全平方数
题目描述: 给你一个整数 n，返回和为 n 的完全平方数的最少数量。
完全平方数是一个整数，其值等于另一个整数的平方；换句话说，其值等于一个整数自乘的积。例如，1、4、9 和 16 都是完全平方数，而 3 和 11 不是。
链接: https://leetcode.cn/problems/perfect-squares/
#
解题思路:
这是一个完全背包问题。我们可以将问题转化为：使用最少量的物品（每个物品是一个完全平方数），恰好装满容量为 n 的背包。
#
状态定义: dp[i] 表示和为 i 的完全平方数的最少数量
状态转移方程: dp[i] = min(dp[i], dp[i - j * j] + 1)，其中 j * j <= i
初始状态: dp[0] = 0，表示和为 0 的完全平方数的最少数量为 0
#
时间复杂度: O(n * sqrt(n))，其中 n 是给定的整数

```

```
空间复杂度: O(n), 使用一维 DP 数组
```

```
from collections import deque
import math
```

```
def num_squares(n: int) -> int:
```

```
 """
```

```
 找出和为 n 的完全平方数的最少数量
```

```
参数:
```

```
 n: 给定的整数
```

```
返回:
```

```
 最少数量
```

```
 """
```

```
if n < 1:
```

```
 return 0
```

```
创建 DP 数组, dp[i] 表示和为 i 的完全平方数的最少数量
```

```
dp = [0] * (n + 1)
```

```
初始化 DP 数组, 初始值设为最大可能值 (即全部用 1 相加)
```

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

```
 dp[i] = i # 最坏情况下, i 可以由 i 个 1 组成
```

```
填充 DP 数组
```

```
for i in range(2, n + 1):
```

```
 # 尝试所有可能的完全平方数 j^2, 其中 j^2 <= i
```

```
 for j in range(1, int(math.sqrt(i)) + 1):
```

```
 # 更新状态: 选择当前完全平方数 j^2, 那么问题转化为求 dp[i - j * j] + 1
```

```
 dp[i] = min(dp[i], dp[i - j * j] + 1)
```

```
return dp[n]
```

```
def num_squares_optimized(n: int) -> int:
```

```
 """
```

```
优化版本, 预先生成所有可能的完全平方数
```

```
参数:
```

```
 n: 给定的整数
```

```
返回:
```

```
 最少数量
```

```

"""
if n < 1:
 return 0

预先生成所有可能的完全平方数
max_square_root = int(math.sqrt(n))
squares = [(i + 1) * (i + 1) for i in range(max_square_root)]

创建 DP 数组
dp = [0] * (n + 1)

初始化 DP 数组
for i in range(1, n + 1):
 dp[i] = i

填充 DP 数组
for i in range(2, n + 1):
 # 尝试所有可能的完全平方数
 for square in squares:
 if square > i:
 break # 由于 squares 是递增的, 当 square > i 时, 后面的平方数也都大于 i, 直接 break
 dp[i] = min(dp[i], dp[i - square] + 1)

return dp[n]

```

```
def num_squares_bfs(n: int) -> int:
```

```
"""

```

使用广度优先搜索(BFS)实现

我们可以将每个数字看作一个节点, 如果两个数字之间相差一个完全平方数, 那么它们之间有一条边  
问题转化为: 从 0 出发, 到 n 的最短路径长度

参数:

n: 给定的整数

返回:

最少数量

```
"""

```

```
if n < 1:
 return 0
```

```
预先生成所有可能的完全平方数
```

```
max_square_root = int(math.sqrt(n))
```

```
squares = [(i + 1) * (i + 1) for i in range(max_square_root)]
```

```

使用队列进行 BFS
queue = deque()
visited = [False] * (n + 1) # 记录哪些数字已经访问过，避免重复处理

queue.append(0) # 从 0 开始
visited[0] = True
level = 0 # 当前的层数，即使用的完全平方数的数量

while queue:
 level += 1
 size = len(queue)

 # 处理当前层的所有节点
 for _ in range(size):
 current = queue.popleft()

 # 尝试所有可能的完全平方数
 for square in squares:
 next_num = current + square

 if next_num == n:
 return level # 找到目标值，返回当前层数

 if next_num > n or visited[next_num]:
 continue # 超过目标值或者已经访问过，跳过

 visited[next_num] = True
 queue.append(next_num)

return n # 默认返回 n (实际上不应该到达这里)

def num_squares_math(n: int) -> int:
 """
 使用数学方法优化，基于拉格朗日四平方定理
 四平方定理：每个自然数都可以表示为 4 个整数的平方和
 """

 参数：
 n: 给定的整数

 返回：
 最少数量
 """

```

```
if n < 1:
 return 0

如果 n 是完全平方数，直接返回 1
if is_perfect_square(n):
 return 1

检查是否可以表示为两个完全平方数的和
if can_be_expressed_as_sum_of_two_squares(n):
 return 2

检查是否可以表示为三个完全平方数的和
根据 Legendre 三平方定理，如果 n 不是形如 $4^k \cdot (8m+7)$ ，则可以表示为三个平方数的和
temp = n
while temp % 4 == 0:
 temp //= 4
if temp % 8 != 7:
 return 3

根据四平方定理，所有自然数都可以表示为 4 个平方数的和
return 4
```

```
def is_perfect_square(num: int) -> bool:
```

```
"""
```

判断一个数是否是完全平方数

参数：

num：要判断的数

返回：

是否是完全平方数

```
"""
```

```
sqrt_num = int(math.sqrt(num))
```

```
return sqrt_num * sqrt_num == num
```

```
def can_be_expressed_as_sum_of_two_squares(num: int) -> bool:
```

```
"""
```

判断一个数是否可以表示为两个完全平方数的和

参数：

num：要判断的数

返回：

是否可以表示为两个完全平方数的和

```
"""
for i in range(int(math.sqrt(num)) + 1):
 remainder = num - i * i
 if is_perfect_square(remainder):
 return True
return False
```

```
def num_squares_dp_alt(n: int) -> int:
```

```
"""
另一种 DP 实现方式，从完全平方数的角度考虑
```

参数：

n：给定的整数

返回：

最少数量

```
"""
if n < 1:
 return 0
```

```
创建 DP 数组
```

```
dp = [float('inf')] * (n + 1)
```

```
dp[0] = 0 # 初始状态：和为 0 的完全平方数的最少数量为 0
```

```
对于每个完全平方数
```

```
for i in range(1, int(math.sqrt(n)) + 1):
```

```
 square = i * i
```

```
 # 对于所有可以表示为当前完全平方数加上另一个数的数
```

```
 for j in range(square, n + 1):
```

```
 # 更新状态
```

```
 dp[j] = min(dp[j], dp[j - square] + 1)
```

```
return dp[n]
```

```
测试代码
```

```
if __name__ == "__main__":
```

```
 # 测试用例 1
```

```
 n1 = 12
```

```
 print(f"测试用例 1 结果: {num_squares(n1)} (预期: 3)")
```

```
 print(f"优化版本结果: {num_squares_optimized(n1)}")
```

```
 print(f"BFS 版本结果: {num_squares_bfs(n1)}")
```

```
 print(f"数学优化版本结果: {num_squares_math(n1)}")
```

```

print(f"另一种 DP 实现结果: {num_squares_dp_alt(n1)}")

print("-----")

测试用例 2
n2 = 13
print(f"测试用例 2 结果: {num_squares(n2)} (预期: 2)")
print(f"优化版本结果: {num_squares_optimized(n2)}")
print(f"BFS 版本结果: {num_squares_bfs(n2)}")
print(f"数学优化版本结果: {num_squares_math(n2)}")

print("-----")

测试用例 3
n3 = 1
print(f"测试用例 3 结果: {num_squares(n3)} (预期: 1)")

print("-----")

测试用例 4
n4 = 2
print(f"测试用例 4 结果: {num_squares(n4)} (预期: 2)")

测试用例 5 - 边界情况
n5 = 0
print(f"\n测试用例 5 结果: {num_squares(n5)} (预期: 0)")

测试用例 6 - 较大的数
n6 = 1000
print(f"\n测试用例 6 结果: {num_squares(n6)}")
print(f"数学优化版本结果: {num_squares_math(n6)}")

```

=====

文件: Code41\_CoinChangeII.cpp

=====

```

#include <iostream>
#include <vector>
#include <algorithm>

// LeetCode 518. 零钱兑换 II
// 题目描述: 给你一个整数数组 coins 表示不同面额的硬币，另给一个整数 amount 表示总金额。
// 请你计算并返回可以凑成总金额的硬币组合数。如果任何硬币组合都无法凑出总金额，返回 0。

```

```

// 假设每一种面额的硬币有无限个。
// 链接: https://leetcode.cn/problems/coin-change-ii/
//
// 解题思路:
// 这是一个完全背包问题的变种。我们可以将问题转化为: 使用不同面额的硬币(可以重复使用), 恰好凑出总金额 amount 的方式有多少种。
//
// 状态定义: dp[j] 表示凑成总金额 j 的硬币组合数
// 状态转移方程: dp[j] += dp[j - coin], 其中 coin 是当前硬币的面额, 且 j >= coin
// 初始状态: dp[0] = 1, 表示凑成总金额 0 的方式有一种(不使用任何硬币)
//
// 时间复杂度: O(amount * n), 其中 n 是硬币的种类数
// 空间复杂度: O(amount), 使用一维 DP 数组

class Solution {
public:
 // 基础 DP 解法
 int change(int amount, std::vector<int>& coins) {
 if (amount < 0) {
 return 0;
 }

 // 创建 DP 数组, dp[j] 表示凑成总金额 j 的硬币组合数
 std::vector<int> dp(amount + 1, 0);

 // 初始状态: 凑成总金额 0 的方式有一种(不使用任何硬币)
 dp[0] = 1;

 // 填充 DP 数组
 // 注意: 这里我们先遍历硬币, 再遍历金额, 这样可以确保每个硬币只被考虑一次, 避免重复计算不同的排列
 for (int coin : coins) {
 for (int j = coin; j <= amount; j++) {
 dp[j] += dp[j - coin];
 }
 }

 return dp[amount];
 }

 // 错误实现示例: 先遍历金额, 再遍历硬币
 // 这种实现会将不同的排列视为不同的组合
 // 例如: [1, 2] 和 [2, 1] 会被视为两种不同的组合
}

```

```
int changeIncorrect(int amount, std::vector<int>& coins) {
 if (amount < 0) {
 return 0;
 }

 std::vector<int> dp(amount + 1, 0);
 dp[0] = 1;

 // 错误: 先遍历金额, 再遍历硬币
 for (int j = 1; j <= amount; j++) {
 for (int coin : coins) {
 if (j >= coin) {
 dp[j] += dp[j - coin];
 }
 }
 }

 return dp[amount];
}
```

```
// 优化实现: 提前过滤掉大于 amount 的硬币
int changeOptimized(int amount, std::vector<int>& coins) {
 if (amount < 0) {
 return 0;
 }
```

```
// 过滤掉大于 amount 的硬币
std::vector<int> filteredCoins;
for (int coin : coins) {
 if (coin <= amount) {
 filteredCoins.push_back(coin);
 }
}
```

```
// 创建 DP 数组
std::vector<int> dp(amount + 1, 0);
dp[0] = 1;
```

```
// 填充 DP 数组
for (int coin : filteredCoins) {
 for (int j = coin; j <= amount; j++) {
 dp[j] += dp[j - coin];
 }
}
```

```

 }

 return dp[amount];
}

// 打印出所有可能的硬币组合
void printAllCombinations(int amount, std::vector<int>& coins) {
 std::vector<std::vector<int>> result;
 std::vector<int> current;

 // 先对硬币排序，确保较小的面额在前
 std::sort(coins.begin(), coins.end());

 backtrack(amount, coins, 0, current, result);

 std::cout << "所有可能的硬币组合:" << std::endl;
 for (const auto& combination : result) {
 std::cout << "[";
 for (size_t i = 0; i < combination.size(); i++) {
 std::cout << combination[i];
 if (i < combination.size() - 1) {
 std::cout << ", ";
 }
 }
 std::cout << "]" << std::endl;
 }
}

// 动态规划方法打印所有可能的硬币组合
void printAllCombinationsDP(int amount, std::vector<int>& coins) {
 if (amount < 0 || coins.empty()) {
 return;
 }

 // dp[j]存储凑成总金额 j 的所有组合
 std::vector<std::vector<std::vector<int>>> dp(amount + 1);

 // 凑成总金额 0 的方式有一个空组合
 dp[0].push_back({});

 // 填充 dp 数组
 for (int coin : coins) {
 for (int j = coin; j <= amount; j++) {

```

```

 // 对于 dp[j - coin] 中的每个组合，添加当前硬币
 for (const auto& prev : dp[j - coin]) {
 std::vector<int> newCombination = prev;
 newCombination.push_back(coin);
 dp[j].push_back(newCombination);
 }
 }

}

std::cout << "所有可能的硬币组合 (DP 实现):" << std::endl;
for (const auto& combination : dp[amount]) {
 std::cout << "[";
 for (size_t i = 0; i < combination.size(); i++) {
 std::cout << combination[i];
 if (i < combination.size() - 1) {
 std::cout << ", ";
 }
 }
 std::cout << "]" << std::endl;
}
}

private:
 // 回溯辅助方法，用于找出所有可能的硬币组合
 void backtrack(int remaining, const std::vector<int>& coins, int index,
 std::vector<int>& current, std::vector<std::vector<int>>& result) {
 if (remaining == 0) {
 // 找到一个有效组合
 result.push_back(current);
 return;
 }

 if (remaining < 0 || index >= coins.size()) {
 return;
 }

 // 不使用当前硬币
 backtrack(remaining, coins, index + 1, current, result);

 // 使用当前硬币（可以重复使用）
 if (remaining >= coins[index]) {
 current.push_back(coins[index]);
 // 注意：这里 index 没有增加，因为可以重复使用当前硬币
 }
 }
}

```

```

 backtrack(remaining - coins[index], coins, index, current, result);
 current.pop_back(); // 回溯
 }
}
};

int main() {
 Solution solution;

 // 测试用例 1
 int amount1 = 5;
 std::vector<int> coins1 = {1, 2, 5};
 std::cout << "测试用例 1 结果: " << solution.change(amount1, coins1) << " (预期: 4)" <<
std::endl;
 solution.printAllCombinations(amount1, coins1);
 std::cout << "-----" << std::endl;

 // 测试用例 2
 int amount2 = 3;
 std::vector<int> coins2 = {2};
 std::cout << "测试用例 2 结果: " << solution.change(amount2, coins2) << " (预期: 0)" <<
std::endl;
 std::cout << "-----" << std::endl;

 // 测试用例 3
 int amount3 = 10;
 std::vector<int> coins3 = {10};
 std::cout << "测试用例 3 结果: " << solution.change(amount3, coins3) << " (预期: 1)" <<
std::endl;
 std::cout << "-----" << std::endl;

 // 测试用例 4
 int amount4 = 0;
 std::vector<int> coins4 = {1, 2, 5};
 std::cout << "测试用例 4 结果: " << solution.change(amount4, coins4) << " (预期: 1)" <<
std::endl;

 // 测试优化版本
 std::cout << "\n测试优化版本:" << std::endl;
 std::cout << "测试用例 1 结果: " << solution.changeOptimized(amount1, coins1) << " (预期: 4)" <<
std::endl;

 // 测试 DP 打印组合功能

```

```
 std::cout << "\n 测试 DP 打印组合功能:" << std::endl;
 solution.printAllCombinationsDP(amount1, coins1);

 return 0;
}
```

=====

文件: Code41\_CoinChangeII.java

=====

```
package class073;

// LeetCode 518. 零钱兑换 II
// 题目描述: 给你一个整数数组 coins 表示不同面额的硬币，另给一个整数 amount 表示总金额。
// 请你计算并返回可以凑成总金额的硬币组合数。如果任何硬币组合都无法凑出总金额，返回 0。
// 假设每一种面额的硬币有无限个。
// 链接: https://leetcode.cn/problems/coin-change-ii/
//
// 解题思路:
// 这是一个完全背包问题的变种。我们可以将问题转化为: 使用不同面额的硬币(可以重复使用)，恰好凑出总金额 amount 的方式有多少种。
//
// 状态定义: dp[j] 表示凑成总金额 j 的硬币组合数
// 状态转移方程: dp[j] += dp[j - coin]，其中 coin 是当前硬币的面额，且 j >= coin
// 初始状态: dp[0] = 1，表示凑成总金额 0 的方式有一种(不使用任何硬币)
//
// 时间复杂度: O(amount * n)，其中 n 是硬币的种类数
// 空间复杂度: O(amount)，使用一维 DP 数组
```

```
public class Code41_CoinChangeII {

 // 主方法，用于测试
 public static void main(String[] args) {
 // 测试用例 1
 int amount1 = 5;
 int[] coins1 = {1, 2, 5};
 System.out.println("测试用例 1 结果: " + change(amount1, coins1)); // 预期输出: 4

 // 测试用例 2
 int amount2 = 3;
 int[] coins2 = {2};
 System.out.println("测试用例 2 结果: " + change(amount2, coins2)); // 预期输出: 0
 }
}
```

```

// 测试用例 3
int amount3 = 10;
int[] coins3 = {10};
System.out.println("测试用例 3 结果: " + change(amount3, coins3)); // 预期输出: 1

// 测试用例 4
int amount4 = 0;
int[] coins4 = {1, 2, 5};
System.out.println("测试用例 4 结果: " + change(amount4, coins4)); // 预期输出: 1
}

/**
 * 计算可以凑成总金额的硬币组合数
 * @param amount 总金额
 * @param coins 硬币面额数组
 * @return 可以凑成总金额的硬币组合数
 */
public static int change(int amount, int[] coins) {
 if (amount < 0 || coins == null) {
 return 0;
 }

 // 创建 DP 数组, dp[j] 表示凑成总金额 j 的硬币组合数
 int[] dp = new int[amount + 1];

 // 初始状态: 凑成总金额 0 的方式有一种 (不使用任何硬币)
 dp[0] = 1;

 // 填充 DP 数组
 // 注意: 这里我们先遍历硬币, 再遍历金额, 这样可以确保每个硬币只被考虑一次, 避免重复计算不同的排列
 for (int coin : coins) {
 for (int j = coin; j <= amount; j++) {
 dp[j] += dp[j - coin];
 }
 }

 return dp[amount];
}

/**
 * 错误实现示例: 先遍历金额, 再遍历硬币
 * 这种实现会将不同的排列视为不同的组合

```

\* 例如: [1, 2] 和 [2, 1] 会被视为两种不同的组合

\*/

```

public static int changeIncorrect(int amount, int[] coins) {
 if (amount < 0 || coins == null) {
 return 0;
 }

 int[] dp = new int[amount + 1];
 dp[0] = 1;

 // 错误: 先遍历金额, 再遍历硬币
 for (int j = 1; j <= amount; j++) {
 for (int coin : coins) {
 if (j >= coin) {
 dp[j] += dp[j - coin];
 }
 }
 }

 return dp[amount];
}

```

/\*\*

\* 优化实现: 提前过滤掉大于 amount 的硬币

\*/

```

public static int changeOptimized(int amount, int[] coins) {
 if (amount < 0 || coins == null) {
 return 0;
 }

 // 过滤掉大于 amount 的硬币
 int[] filteredCoins = new int[coins.length];
 int filteredCount = 0;
 for (int coin : coins) {
 if (coin <= amount) {
 filteredCoins[filteredCount++] = coin;
 }
 }

 // 创建 DP 数组
 int[] dp = new int[amount + 1];
 dp[0] = 1;

```

```

// 填充 DP 数组
for (int i = 0; i < filteredCount; i++) {
 int coin = filteredCoins[i];
 for (int j = coin; j <= amount; j++) {
 dp[j] += dp[j - coin];
 }
}

return dp[amount];
}

/***
 * 打印出所有可能的硬币组合
 * 注意：这个方法仅用于教学目的，对于大额 amount 可能效率不高
 */
public static void printAllCombinations(int amount, int[] coins) {
 java.util.List<List<Integer>> result = new java.util.ArrayList<>();
 java.util.List<Integer> current = new java.util.ArrayList<>();

 // 先对硬币排序，确保较小的面额在前
 java.util.Arrays.sort(coins);

 backtrack(amount, coins, 0, current, result);

 System.out.println("所有可能的硬币组合:");
 for (List<Integer> combination : result) {
 System.out.println(combination);
 }
}

/***
 * 回溯辅助方法，用于找出所有可能的硬币组合
 */
private static void backtrack(int remaining, int[] coins, int index, List<Integer> current,
List<List<Integer>> result) {
 if (remaining == 0) {
 // 找到一个有效组合
 result.add(new java.util.ArrayList<>(current));
 return;
 }

 if (remaining < 0 || index >= coins.length) {
 return;
 }

 for (int i = index; i < coins.length; i++) {
 current.add(coins[i]);
 backtrack(remaining - coins[i], coins, i, current, result);
 current.remove(current.size() - 1);
 }
}

```

```
}

// 不使用当前硬币
backtrack(remaining, coins, index + 1, current, result);

// 使用当前硬币（可以重复使用）
if (remaining >= coins[index]) {
 current.add(coins[index]);
 // 注意：这里 index 没有增加，因为可以重复使用当前硬币
 backtrack(remaining - coins[index], coins, index, current, result);
 current.remove(current.size() - 1); // 回溯
}
}
```

```
/***
 * 动态规划方法打印所有可能的硬币组合
 * 注意：这个方法仅用于教学目的，对于大额 amount 可能效率不高
 */
```

```
public static void printAllCombinationsDP(int amount, int[] coins) {
 if (amount < 0 || coins == null || coins.length == 0) {
 return;
 }
}
```

```
// dp[j] 存储凑成总金额 j 的所有组合
java.util.List<List<List<Integer>>> dp = new java.util.ArrayList<>();
```

```
// 初始化 dp 数组
for (int j = 0; j <= amount; j++) {
 dp.add(new java.util.ArrayList<>());
}
```

```
// 凑成总金额 0 的方式有一个空组合
dp.get(0).add(new java.util.ArrayList<>());
```

```
// 填充 dp 数组
for (int coin : coins) {
 for (int j = coin; j <= amount; j++) {
 // 对于 dp[j - coin] 中的每个组合，添加当前硬币
 for (List<Integer> prev : dp.get(j - coin)) {
 List<Integer> newCombination = new java.util.ArrayList<>(prev);
 newCombination.add(coin);
 dp.get(j).add(newCombination);
 }
 }
}
```

```

 }
 }

System.out.println("所有可能的硬币组合 (DP 实现):");
for (List<Integer> combination : dp.get(amount)) {
 System.out.println(combination);
}
}

=====

```

文件: Code41\_CoinChangeII.py

```

LeetCode 518. 零钱兑换 II
题目描述: 给你一个整数数组 coins 表示不同面额的硬币，另给一个整数 amount 表示总金额。
请你计算并返回可以凑成总金额的硬币组合数。如果任何硬币组合都无法凑出总金额，返回 0。
假设每一种面额的硬币有无限个。
链接: https://leetcode.cn/problems/coin-change-ii/
#
解题思路:
这是一个完全背包问题的变种。我们可以将问题转化为: 使用不同面额的硬币（可以重复使用），恰好凑出总金额 amount 的方式有多少种。
#
状态定义: dp[j] 表示凑成总金额 j 的硬币组合数
状态转移方程: dp[j] += dp[j - coin]，其中 coin 是当前硬币的面额，且 j >= coin
初始状态: dp[0] = 1，表示凑成总金额 0 的方式有一种（不使用任何硬币）
#
时间复杂度: O(amount * n)，其中 n 是硬币的种类数
空间复杂度: O(amount)，使用一维 DP 数组

def change(amount: int, coins: list[int]) -> int:
 """
 计算可以凑成总金额的硬币组合数
 """

参数:
```

amount: 总金额

coins: 硬币面额数组

返回:

可以凑成总金额的硬币组合数

"""

if amount < 0 or not coins:

```

 return 0 if amount != 0 else 1

创建 DP 数组, dp[j] 表示凑成总金额 j 的硬币组合数
dp = [0] * (amount + 1)

初始状态: 凑成总金额 0 的方式有一种 (不使用任何硬币)
dp[0] = 1

填充 DP 数组
注意: 这里我们先遍历硬币, 再遍历金额, 这样可以确保每个硬币只被考虑一次, 避免重复计算不同的排列
for coin in coins:
 for j in range(coin, amount + 1):
 dp[j] += dp[j - coin]

return dp[amount]

```

```
def change_incorrect(amount: int, coins: list[int]) -> int:
 """

```

错误实现示例: 先遍历金额, 再遍历硬币  
这种实现会将不同的排列视为不同的组合  
例如: [1, 2] 和 [2, 1] 会被视为两种不同的组合

参数:

amount: 总金额  
coins: 硬币面额数组

返回:

排列数 (不是组合数)

```
"""

```

```
if amount < 0 or not coins:
 return 0 if amount != 0 else 1
```

```
dp = [0] * (amount + 1)
dp[0] = 1
```

# 错误: 先遍历金额, 再遍历硬币

```
for j in range(1, amount + 1):
 for coin in coins:
 if j >= coin:
 dp[j] += dp[j - coin]
```

```
return dp[amount]
```

```

def change_optimized(amount: int, coins: list[int]) -> int:
 """
 优化实现：提前过滤掉大于 amount 的硬币

 参数：
 amount: 总金额
 coins: 硬币面额数组

 返回：
 可以凑成总金额的硬币组合数
 """
 if amount < 0:
 return 0
 if not coins:
 return 1 if amount == 0 else 0

 # 过滤掉大于 amount 的硬币
 filtered_coins = [coin for coin in coins if coin <= amount]

 # 创建 DP 数组
 dp = [0] * (amount + 1)
 dp[0] = 1

 # 填充 DP 数组
 for coin in filtered_coins:
 for j in range(coin, amount + 1):
 dp[j] += dp[j - coin]

 return dp[amount]

def change_both_ways(amount: int, coins: list[int]) -> dict:
 """
 同时计算组合数和排列数，用于对比

 参数：
 amount: 总金额
 coins: 硬币面额数组

 返回：
 包含组合数和排列数的字典
 """
 if amount < 0:

```

```

 return {'combinations': 0, 'permutations': 0}
if not coins:
 return {'combinations': 1 if amount == 0 else 0, 'permutations': 1 if amount == 0 else 0}

计算组合数（先遍历硬币，再遍历金额）
combinations = [0] * (amount + 1)
combinations[0] = 1
for coin in coins:
 for j in range(coin, amount + 1):
 combinations[j] += combinations[j - coin]

计算排列数（先遍历金额，再遍历硬币）
permutations = [0] * (amount + 1)
permutations[0] = 1
for j in range(1, amount + 1):
 for coin in coins:
 if j >= coin:
 permutations[j] += permutations[j - coin]

return {'combinations': combinations[amount], 'permutations': permutations[amount]}

```

def print\_all\_combinations(amount: int, coins: list[int]) -> list[list[int]]:  
 """

打印出所有可能的硬币组合

注意：这个方法仅用于教学目的，对于大额 amount 可能效率不高

参数：

amount：总金额

coins：硬币面额数组

返回：

所有可能的硬币组合列表

"""

result = []

current = []

# 先对硬币排序，确保较小的面额在前  
 coins.sort()

```

def backtrack(remaining: int, index: int):
 if remaining == 0:
 # 找到一个有效组合
 result.append(current.copy())

```

```

 return

 if remaining < 0 or index >= len(coins):
 return

 # 不使用当前硬币
 backtrack(remaining, index + 1)

 # 使用当前硬币（可以重复使用）
 if remaining >= coins[index]:
 current.append(coins[index])
 # 注意：这里 index 没有增加，因为可以重复使用当前硬币
 backtrack(remaining - coins[index], index)
 current.pop() # 回溯

backtrack(amount, 0)

print("所有可能的硬币组合:")
for combo in result:
 print(combo)

return result

```

```
def print_all_combinations_dp(amount: int, coins: list[int]) -> list[list[int]]:
 """

```

动态规划方法打印所有可能的硬币组合

注意：这个方法仅用于教学目的，对于大额 amount 可能效率不高

参数：

amount：总金额

coins：硬币面额数组

返回：

所有可能的硬币组合列表

```
"""

```

```
if amount < 0 or not coins:
```

```
 return [] if amount != 0 else [[]]
```

```
dp[j]存储凑成总金额 j 的所有组合
```

```
dp = [[] for _ in range(amount + 1)]
```

```
凑成总金额 0 的方式有一个空组合
```

```
dp[0] = [[]]
```

```

填充 dp 数组
for coin in coins:
 for j in range(coin, amount + 1):
 # 对于 dp[j - coin] 中的每个组合，添加当前硬币
 for prev in dp[j - coin]:
 new_combination = prev.copy()
 new_combination.append(coin)
 dp[j].append(new_combination)

print("所有可能的硬币组合 (DP 实现):")
for combo in dp[amount]:
 print(combo)

return dp[amount]

```

```

def change_large_amount_optimized(amount: int, coins: list[int]) -> int:
"""

```

针对大额 amount 的优化实现，使用模运算避免整数溢出

注意：在 Python 中整数溢出不是问题，但这个方法演示了如何处理这种情况

参数：

amount：总金额

coins：硬币面额数组

返回：

可以凑成总金额的硬币组合数

"""

```
if amount < 0:
```

```
 return 0
```

```
if not coins:
```

```
 return 1 if amount == 0 else 0
```

# 过滤掉大于 amount 的硬币

```
filtered_coins = [coin for coin in coins if coin <= amount]
```

# 创建 DP 数组

```
dp = [0] * (amount + 1)
```

```
dp[0] = 1
```

# 填充 DP 数组

```
for coin in filtered_coins:
```

```
 for j in range(coin, amount + 1):
```

```
dp[j] += dp[j - coin]
在实际应用中，如果结果可能非常大，可以使用模运算
dp[j] %= MOD # MOD 可以是一个大质数，如 10^9+7

return dp[amount]

测试代码
if __name__ == "__main__":
 # 测试用例 1
 amount1 = 5
 coins1 = [1, 2, 5]
 print(f"测试用例 1 结果: {change(amount1, coins1)} (预期: 4)")
 print_all_combinations(amount1, coins1)
 print("-----")

 # 测试用例 2
 amount2 = 3
 coins2 = [2]
 print(f"测试用例 2 结果: {change(amount2, coins2)} (预期: 0)")
 print("-----")

 # 测试用例 3
 amount3 = 10
 coins3 = [10]
 print(f"测试用例 3 结果: {change(amount3, coins3)} (预期: 1)")
 print("-----")

 # 测试用例 4
 amount4 = 0
 coins4 = [1, 2, 5]
 print(f"测试用例 4 结果: {change(amount4, coins4)} (预期: 1)")

测试组合数与排列数的区别
print("\n 测试组合数与排列数的区别:")
result = change_both_ways(amount1, coins1)
print(f"总金额 {amount1}, 硬币 {coins1}:")
print(f"组合数: {result['combinations']} ")
print(f"排列数: {result['permutations']} ")

测试 DP 打印组合功能
print("\n 测试 DP 打印组合功能:")
print_all_combinations_dp(amount1, coins1)
```

```

测试优化版本
print("\n 测试优化版本:")
print(f"测试用例 1 结果: {change_optimized(amount1, coins1)} (预期: 4)")

测试更大的金额
print("\n 测试更大的金额:")
amount5 = 100
coins5 = [1, 5, 10, 25]
print(f"总金额 {amount5}, 硬币 {coins5} 的组合数: {change(amount5, coins5)}")

```

=====

文件: Code42\_TargetSum.cpp

=====

```

#include <iostream>
#include <vector>
#include <unordered_map>
#include <string>

// LeetCode 494. 目标和
// 题目描述: 给你一个整数数组 nums 和一个整数 target。
// 向数组中的每个整数前添加 '+' 或 '-'，然后串联起所有整数，可以构造一个 表达式：
// 例如，nums = [2, 1]，可以在 2 之前添加 '+'，在 1 之前添加 '-'，然后串联起来得到表达式 "+2-1"。
// 返回可以通过上述方法构造的、运算结果等于 target 的不同 表达式 的数目。
// 链接: https://leetcode.cn/problems/target-sum/
//
// 解题思路:
// 这是一个 0-1 背包问题的变种。我们可以将问题转化为：找到一个子集 P，使得 sum(P) - sum(N) = target，其中 N 是数组中不在 P 中的元素。
// 可以证明: sum(P) - sum(N) = target => sum(P) = (sum(nums) + target) / 2
// 因此，问题转化为：在数组 nums 中，有多少个子集的和等于 (sum(nums) + target) / 2。
//
// 状态定义: dp[j] 表示和为 j 的子集数目
// 状态转移方程: dp[j] += dp[j - nums[i]]，其中 nums[i] 是当前元素，且 j >= nums[i]
// 初始状态: dp[0] = 1，表示和为 0 的子集数目为 1 (空集)
//
// 时间复杂度: O(n * sum)，其中 n 是数组的长度，sum 是数组元素的和
// 空间复杂度: O(sum)，使用一维 DP 数组

class Solution {
public:
 // 基础 DP 解法

```

```

int findTargetSumWays(std::vector<int>& nums, int target) {
 int n = nums.size();
 if (n == 0) {
 return target == 0 ? 1 : 0;
 }

 // 计算数组元素的总和
 int sum = 0;
 for (int num : nums) {
 sum += num;
 }

 // 检查是否有解
 // 1. sum < abs(target): 总和小于目标值的绝对值, 无解
 // 2. (sum + target) % 2 != 0: sum + target 必须是偶数, 否则无法平均分成两部分
 if (sum < std::abs(target) || (sum + target) % 2 != 0) {
 return 0;
 }

 // 计算目标和: sum(P) = (sum + target) / 2
 int targetSum = (sum + target) / 2;
 if (targetSum < 0) {
 return 0; // 目标和为负数, 无解
 }

 // 创建 DP 数组, dp[j] 表示和为 j 的子集数目
 std::vector<int> dp(targetSum + 1, 0);

 // 初始状态: 和为 0 的子集数目为 1 (空集)
 dp[0] = 1;

 // 填充 DP 数组
 for (int num : nums) {
 // 注意: 这里我们从后往前遍历, 避免重复使用同一个元素
 for (int j = targetSum; j >= num; j--) {
 dp[j] += dp[j - num];
 }
 }

 return dp[targetSum];
}

// 使用二维 DP 数组的实现

```

```

int findTargetSumWays2D(std::vector<int>& nums, int target) {
 int n = nums.size();
 if (n == 0) {
 return target == 0 ? 1 : 0;
 }

 int sum = 0;
 for (int num : nums) {
 sum += num;
 }

 if (sum < std::abs(target) || (sum + target) % 2 != 0) {
 return 0;
 }

 int targetSum = (sum + target) / 2;
 if (targetSum < 0) {
 return 0;
 }

 // 创建二维 DP 数组
 std::vector<std::vector<int>> dp(n + 1, std::vector<int>(targetSum + 1, 0));

 // 初始状态: 使用前 0 个元素, 和为 0 的子集数目为 1 (空集)
 dp[0][0] = 1;

 // 填充 DP 数组
 for (int i = 1; i <= n; i++) {
 int num = nums[i - 1]; // 当前元素
 for (int j = 0; j <= targetSum; j++) {
 // 不选择当前元素
 dp[i][j] = dp[i - 1][j];
 // 选择当前元素 (如果 j >= num)
 if (j >= num) {
 dp[i][j] += dp[i - 1][j - num];
 }
 }
 }

 return dp[n][targetSum];
}

// 使用回溯法的实现

```

```

int findTargetSumWaysBacktrack(std::vector<int>& nums, int target) {
 int count = 0;
 backtrack(nums, target, 0, 0, count);
 return count;
}

// 使用记忆化递归的实现

int findTargetSumWaysMemo(std::vector<int>& nums, int target) {
 // 创建记忆化缓存，键为(index, currentSum)，值为对应的方法数
 std::unordered_map<std::string, int> memo;
 return backtrackMemo(nums, target, 0, 0, memo);
}

// 优化版本，考虑数组中包含 0 的情况

int findTargetSumWaysOptimized(std::vector<int>& nums, int target) {
 int n = nums.size();
 if (n == 0) {
 return target == 0 ? 1 : 0;
 }

 int sum = 0;
 int zeroCount = 0;

 // 计算总和和 0 的个数
 for (int num : nums) {
 sum += num;
 if (num == 0) {
 zeroCount++;
 }
 }

 // 检查是否有解
 if (sum < std::abs(target) || (sum + target) % 2 != 0) {
 return 0;
 }

 int targetSum = (sum + target) / 2;
 if (targetSum < 0) {
 return 0;
 }

 // 过滤掉 0，单独处理
 std::vector<int> nonZeroNums;

```

```

for (int num : nums) {
 if (num != 0) {
 nonZeroNums.push_back(num);
 }
}

// 创建 DP 数组
std::vector<int> dp(targetSum + 1, 0);
dp[0] = 1;

// 填充 DP 数组
for (int num : nonZeroNums) {
 for (int j = targetSum; j >= num; j--) {
 dp[j] += dp[j - num];
 }
}

// 每个 0 有两种选择 (+0 或-0), 所以总方法数乘以 2^zeroCount
return dp[targetSum] * static_cast<int>(std::pow(2, zeroCount));
}

// 打印所有可能的表达式
void printAllExpressions(std::vector<int>& nums, int target) {
 std::vector<std::string> result;
 std::string currentExpr;

 // 第一个数特殊处理, 不需要前面的符号
 currentExpr = std::to_string(nums[0]);
 backtrackExpressions(nums, target, 1, nums[0], currentExpr, result);

 // 尝试第一个数为负数的情况
 currentExpr = "-" + std::to_string(nums[0]);
 backtrackExpressions(nums, target, 1, -nums[0], currentExpr, result);

 std::cout << "所有可能的表达式:" << std::endl;
 for (const std::string& expr : result) {
 std::cout << expr << std::endl;
 }
 std::cout << "总共有 " << result.size() << " 种不同的表达式。" << std::endl;
}

private:
 // 回溯辅助方法

```

```

void backtrack(const std::vector<int>& nums, int target, int index, int currentSum, int& count) {
 // 已经处理完所有元素
 if (index == nums.size()) {
 if (currentSum == target) {
 count++;
 }
 return;
 }

 // 尝试加上当前元素
 backtrack(nums, target, index + 1, currentSum + nums[index], count);

 // 尝试减去当前元素
 backtrack(nums, target, index + 1, currentSum - nums[index], count);
}

// 记忆化递归辅助方法
int backtrackMemo(const std::vector<int>& nums, int target, int index, int currentSum,
 std::unordered_map<std::string, int>& memo) {
 // 已经处理完所有元素
 if (index == nums.size()) {
 return currentSum == target ? 1 : 0;
 }

 // 生成缓存键
 std::string key = std::to_string(index) + "," + std::to_string(currentSum);

 // 检查是否已经计算过
 if (memo.find(key) != memo.end()) {
 return memo[key];
 }

 // 计算两种情况的结果之和
 int add = backtrackMemo(nums, target, index + 1, currentSum + nums[index], memo);
 int subtract = backtrackMemo(nums, target, index + 1, currentSum - nums[index], memo);

 // 存储结果到缓存
 memo[key] = add + subtract;

 return add + subtract;
}

```

```

// 回溯辅助方法，用于生成所有可能的表达式
void backtrackExpressions(const std::vector<int>& nums, int target, int index, int
currentSum,
 std::string currentExpr, std::vector<std::string>& result) {
 if (index == nums.size()) {
 if (currentSum == target) {
 result.push_back(currentExpr);
 }
 return;
 }

 int num = nums[index];

 // 尝试加上当前元素
 backtrackExpressions(nums, target, index + 1, currentSum + num,
 currentExpr + "+" + std::to_string(num), result);

 // 尝试减去当前元素
 backtrackExpressions(nums, target, index + 1, currentSum - num,
 currentExpr + "-" + std::to_string(num), result);
}

};

int main() {
 Solution solution;

 // 测试用例 1
 std::vector<int> nums1 = {1, 1, 1, 1, 1};
 int target1 = 3;
 std::cout << "测试用例 1 结果: " << solution.findTargetSumWays(nums1, target1) << " (预期: 5)"
 << std::endl;
 solution.printAllExpressions(nums1, target1);
 std::cout << "-----" << std::endl;

 // 测试用例 2
 std::vector<int> nums2 = {1};
 int target2 = 1;
 std::cout << "测试用例 2 结果: " << solution.findTargetSumWays(nums2, target2) << " (预期: 1)"
 << std::endl;
 std::cout << "-----" << std::endl;

 // 测试用例 3
 std::vector<int> nums3 = {1, 2, 3, 4, 5};

```

```

int target3 = 3;
std::cout << "测试用例 3 结果: " << solution.findTargetSumWays(nums3, target3) << " (预期: 5)"
<< std::endl;
std::cout << "-----" << std::endl;

// 测试用例 4 - 无法满足的情况
std::vector<int> nums4 = {1, 2, 3};
int target4 = 7;
std::cout << "测试用例 4 结果: " << solution.findTargetSumWays(nums4, target4) << " (预期: 0)"
<< std::endl;

// 测试各种实现方法
std::cout << "\n 测试各种实现方法:" << std::endl;
std::cout << "二维 DP 版本: " << solution.findTargetSumWays2D(nums1, target1) << std::endl;
std::cout << "回溯版本: " << solution.findTargetSumWaysBacktrack(nums1, target1) <<
std::endl;
std::cout << "记忆化递归版本: " << solution.findTargetSumWaysMemo(nums1, target1) <<
std::endl;
std::cout << "优化版本: " << solution.findTargetSumWaysOptimized(nums1, target1) <<
std::endl;

return 0;
}
=====
```

文件: Code42\_TargetSum.java

```
=====
package class073;

// LeetCode 494. 目标和
// 题目描述: 给你一个整数数组 nums 和一个整数 target 。
// 向数组中的每个整数前添加 '+' 或 '-' ，然后串联起所有整数，可以构造一个 表达式 ：
// 例如，nums = [2, 1] ，可以在 2 之前添加 '+' ，在 1 之前添加 '-' ，然后串联起来得到表达式 "+2-1" 。
// 返回可以通过上述方法构造的、运算结果等于 target 的不同 表达式 的数目。
// 链接: https://leetcode.cn/problems/target-sum/
//
// 解题思路:
// 这是一个 0-1 背包问题的变种。我们可以将问题转化为：找到一个子集 P，使得 sum(P) - sum(N) = target，其中 N 是数组中不在 P 中的元素。
// 可以证明: sum(P) - sum(N) = target => sum(P) = (sum(nums) + target) / 2
// 因此，问题转化为：在数组 nums 中，有多少个子集的和等于 (sum(nums) + target) / 2。
```

```

// 状态定义: dp[j] 表示和为 j 的子集数目
// 状态转移方程: dp[j] += dp[j - nums[i]], 其中 nums[i] 是当前元素, 且 j >= nums[i]
// 初始状态: dp[0] = 1, 表示和为 0 的子集数目为 1 (空集)
//
// 时间复杂度: O(n * sum), 其中 n 是数组的长度, sum 是数组元素的和
// 空间复杂度: O(sum), 使用一维 DP 数组

public class Code42_TargetSum {

 // 主方法, 用于测试
 public static void main(String[] args) {
 // 测试用例 1
 int[] nums1 = {1, 1, 1, 1, 1};
 int target1 = 3;
 System.out.println("测试用例 1 结果: " + findTargetSumWays(nums1, target1)); // 预期输出:
 5

 // 测试用例 2
 int[] nums2 = {1};
 int target2 = 1;
 System.out.println("测试用例 2 结果: " + findTargetSumWays(nums2, target2)); // 预期输出:
 1

 // 测试用例 3
 int[] nums3 = {1, 2, 3, 4, 5};
 int target3 = 3;
 System.out.println("测试用例 3 结果: " + findTargetSumWays(nums3, target3)); // 预期输出:
 5

 // 测试用例 4 - 无法满足的情况
 int[] nums4 = {1, 2, 3};
 int target4 = 7;
 System.out.println("测试用例 4 结果: " + findTargetSumWays(nums4, target4)); // 预期输出:
 0
 }

 /**
 * 计算有多少种不同的方法构造运算结果等于 target 的表达式
 * @param nums 整数数组
 * @param target 目标值
 * @return 不同表达式的数目
 */
}

```

```

public static int findTargetSumWays(int[] nums, int target) {
 int n = nums.length;
 if (n == 0) {
 return target == 0 ? 1 : 0;
 }

 // 计算数组元素的总和
 int sum = 0;
 for (int num : nums) {
 sum += num;
 }

 // 检查是否有解
 // 1. sum < Math.abs(target): 总和小于目标值的绝对值, 无解
 // 2. (sum + target) % 2 != 0: sum + target 必须是偶数, 否则无法平均分成两部分
 if (sum < Math.abs(target) || (sum + target) % 2 != 0) {
 return 0;
 }

 // 计算目标和: sum(P) = (sum + target) / 2
 int targetSum = (sum + target) / 2;
 if (targetSum < 0) {
 return 0; // 目标和为负数, 无解
 }

 // 创建 DP 数组, dp[j] 表示和为 j 的子集数目
 int[] dp = new int[targetSum + 1];

 // 初始状态: 和为 0 的子集数目为 1 (空集)
 dp[0] = 1;

 // 填充 DP 数组
 for (int num : nums) {
 // 注意: 这里我们从后往前遍历, 避免重复使用同一个元素
 for (int j = targetSum; j >= num; j--) {
 dp[j] += dp[j - num];
 }
 }

 return dp[targetSum];
}

/***

```

\* 使用二维 DP 数组的实现

\* dp[i][j] 表示使用前 i 个元素，和为 j 的子集数目

\*/

```
public static int findTargetSumWays2D(int[] nums, int target) {
```

```
 int n = nums.length;
```

```
 if (n == 0) {
```

```
 return target == 0 ? 1 : 0;
```

```
}
```

```
 int sum = 0;
```

```
 for (int num : nums) {
```

```
 sum += num;
```

```
}
```

```
 if (sum < Math.abs(target) || (sum + target) % 2 != 0) {
```

```
 return 0;
```

```
}
```

```
 int targetSum = (sum + target) / 2;
```

```
 if (targetSum < 0) {
```

```
 return 0;
```

```
}
```

// 创建二维 DP 数组

```
int[][] dp = new int[n + 1][targetSum + 1];
```

// 初始状态：使用前 0 个元素，和为 0 的子集数目为 1（空集）

```
dp[0][0] = 1;
```

// 填充 DP 数组

```
for (int i = 1; i <= n; i++) {
```

```
 int num = nums[i - 1]; // 当前元素
```

```
 for (int j = 0; j <= targetSum; j++) {
```

```
 // 不选择当前元素
```

```
 dp[i][j] = dp[i - 1][j];
```

```
 // 选择当前元素（如果 j >= num）
```

```
 if (j >= num) {
```

```
 dp[i][j] += dp[i - 1][j - num];
```

```
 }
```

```
}
```

```
}
```

```
return dp[n][targetSum];
```

```

}

/**
 * 使用回溯法的实现
 * 注意：这种方法对于大数组可能效率不高
 */
public static int findTargetSumWaysBacktrack(int[] nums, int target) {
 int[] count = new int[1]; // 使用数组来存储结果，以便在递归中修改
 backtrack(nums, target, 0, 0, count);
 return count[0];
}

/**
 * 回溯辅助方法
 * @param nums 整数数组
 * @param target 目标值
 * @param index 当前处理的索引
 * @param current 当前的和
 * @param count 结果计数器
 */
private static void backtrack(int[] nums, int target, int index, int current, int[] count) {
 // 已经处理完所有元素
 if (index == nums.length) {
 if (current == target) {
 count[0]++;
 }
 return;
 }

 // 尝试加上当前元素
 backtrack(nums, target, index + 1, current + nums[index], count);

 // 尝试减去当前元素
 backtrack(nums, target, index + 1, current - nums[index], count);
}

/**
 * 使用记忆化递归的实现
 */
public static int findTargetSumWaysMemo(int[] nums, int target) {
 // 创建记忆化缓存，键为(index, currentSum)，值为对应的方法数
 java.util.Map<String, Integer> memo = new java.util.HashMap<>();
 return backtrackMemo(nums, target, 0, 0, memo);
}

```

```
}

/**
 * 记忆化递归辅助方法
 */
private static int backtrackMemo(int[] nums, int target, int index, int currentSum,
java.util.Map<String, Integer> memo) {
 // 已经处理完所有元素
 if (index == nums.length) {
 return currentSum == target ? 1 : 0;
 }

 // 生成缓存键
 String key = index + "," + currentSum;

 // 检查是否已经计算过
 if (memo.containsKey(key)) {
 return memo.get(key);
 }

 // 计算两种情况的结果之和
 int add = backtrackMemo(nums, target, index + 1, currentSum + nums[index], memo);
 int subtract = backtrackMemo(nums, target, index + 1, currentSum - nums[index], memo);

 // 存储结果到缓存
 memo.put(key, add + subtract);

 return add + subtract;
}

/**
 * 优化版本，考虑数组中包含 0 的情况
 */
public static int findTargetSumWaysOptimized(int[] nums, int target) {
 int n = nums.length;
 if (n == 0) {
 return target == 0 ? 1 : 0;
 }

 int sum = 0;
 int zeroCount = 0;

 // 计算总和和 0 的个数
```

```

for (int num : nums) {
 sum += num;
 if (num == 0) {
 zeroCount++;
 }
}

// 检查是否有解
if (sum < Math.abs(target) || (sum + target) % 2 != 0) {
 return 0;
}

int targetSum = (sum + target) / 2;
if (targetSum < 0) {
 return 0;
}

// 过滤掉 0，单独处理
int[] nonZeroNums = new int[n - zeroCount];
int idx = 0;
for (int num : nums) {
 if (num != 0) {
 nonZeroNums[idx++] = num;
 }
}

// 创建 DP 数组
int[] dp = new int[targetSum + 1];
dp[0] = 1;

// 填充 DP 数组
for (int num : nonZeroNums) {
 for (int j = targetSum; j >= num; j--) {
 dp[j] += dp[j - num];
 }
}

// 每个 0 有两种选择 (+0 或 -0)，所以总方法数乘以 2^zeroCount
return dp[targetSum] * (int) Math.pow(2, zeroCount);
}

/**
 * 打印所有可能的表达式

```

```

* 注意: 这个方法仅用于教学目的, 对于大数据可能效率不高
*/
public static void printAllExpressions(int[] nums, int target) {
 java.util.List<String> result = new java.util.ArrayList<>();
 StringBuilder currentExpr = new StringBuilder();

 // 第一个数特殊处理, 不需要前面的符号
 currentExpr.append(nums[0]);
 backtrackExpressions(nums, target, 1, nums[0], currentExpr, result);

 // 尝试第一个数为负数的情况
 currentExpr = new StringBuilder();
 currentExpr.append("-").append(nums[0]);
 backtrackExpressions(nums, target, 1, -nums[0], currentExpr, result);

 System.out.println("所有可能的表达式:");
 for (String expr : result) {
 System.out.println(expr);
 }
 System.out.println("总共有 " + result.size() + " 种不同的表达式.");
}

/**
 * 回溯辅助方法, 用于生成所有可能的表达式
*/
private static void backtrackExpressions(int[] nums, int target, int index, int currentSum,
 StringBuilder currentExpr, java.util.List<String>
result) {
 if (index == nums.length) {
 if (currentSum == target) {
 result.add(currentExpr.toString());
 }
 return;
 }

 int num = nums[index];
 int length = currentExpr.length();

 // 尝试加上当前元素
 currentExpr.append("+").append(num);
 backtrackExpressions(nums, target, index + 1, currentSum + num, currentExpr, result);
 currentExpr.setLength(length); // 回溯
}

```

```

 // 尝试减去当前元素
 currentExpr.append("-").append(num);
 backtrackExpressions(nums, target, index + 1, currentSum - num, currentExpr, result);
 currentExpr.setLength(length); // 回溯
}
}

```

=====

文件: Code42\_TargetSum.py

=====

```

LeetCode 494. 目标和
题目描述: 给你一个整数数组 nums 和一个整数 target 。
向数组中的每个整数前添加 '+' 或 '-'，然后串联起所有整数，可以构造一个 表达式 ：
例如，nums = [2, 1]，可以在 2 之前添加 '+'，在 1 之前添加 '-'，然后串联起来得到表达式 "+2-1" 。
返回可以通过上述方法构造的、运算结果等于 target 的不同 表达式 的数目。
链接: https://leetcode.cn/problems/target-sum/
#
解题思路:
这是一个 0-1 背包问题的变种。我们可以将问题转化为：找到一个子集 P，使得 $\text{sum}(P) - \text{sum}(N) = \text{target}$ ，其中 N 是数组中不在 P 中的元素。
可以证明： $\text{sum}(P) - \text{sum}(N) = \text{target} \Rightarrow \text{sum}(P) = (\text{sum}(nums) + \text{target}) / 2$
因此，问题转化为：在数组 nums 中，有多少个子集的和等于 $(\text{sum}(nums) + \text{target}) / 2$ 。
#
状态定义：dp[j] 表示和为 j 的子集数目
状态转移方程：dp[j] += dp[j - nums[i]]，其中 nums[i] 是当前元素，且 $j \geq \text{sum}(nums)$
初始状态：dp[0] = 1，表示和为 0 的子集数目为 1（空集）
#
时间复杂度：O(n * sum)，其中 n 是数组的长度，sum 是数组元素的和
空间复杂度：O(sum)，使用一维 DP 数组

```

```
def find_target_sum_ways(nums: list[int], target: int) -> int:
```

```
"""

```

计算有多少种不同的方法构造运算结果等于 target 的表达式

参数:

nums: 整数数组

target: 目标值

返回:

不同表达式的数目

```
"""

```

```

n = len(nums)
if n == 0:
 return 1 if target == 0 else 0

计算数组元素的总和
total_sum = sum(nums)

检查是否有解
1. total_sum < abs(target): 总和小于目标值的绝对值, 无解
2. (total_sum + target) % 2 != 0: sum + target 必须是偶数, 否则无法平均分成两部分
if total_sum < abs(target) or (total_sum + target) % 2 != 0:
 return 0

计算目标和: sum(P) = (total_sum + target) // 2
target_sum = (total_sum + target) // 2
if target_sum < 0:
 return 0 # 目标和为负数, 无解

创建 DP 数组, dp[j] 表示和为 j 的子集数目
dp = [0] * (target_sum + 1)

初始状态: 和为 0 的子集数目为 1 (空集)
dp[0] = 1

填充 DP 数组
for num in nums:
 # 注意: 这里我们从后往前遍历, 避免重复使用同一个元素
 for j in range(target_sum, num - 1, -1):
 dp[j] += dp[j - num]

return dp[target_sum]

```

```

def find_target_sum_ways_2d(nums: list[int], target: int) -> int:
 """

```

使用二维 DP 数组的实现

$dp[i][j]$  表示使用前  $i$  个元素, 和为  $j$  的子集数目

```

 """

```

```

n = len(nums)
if n == 0:
 return 1 if target == 0 else 0
```

```

total_sum = sum(nums)
```

```

if total_sum < abs(target) or (total_sum + target) % 2 != 0:
 return 0

target_sum = (total_sum + target) // 2
if target_sum < 0:
 return 0

创建二维 DP 数组
dp = [[0] * (target_sum + 1) for _ in range(n + 1)]

初始状态: 使用前 0 个元素, 和为 0 的子集数目为 1 (空集)
dp[0][0] = 1

填充 DP 数组
for i in range(1, n + 1):
 num = nums[i - 1] # 当前元素
 for j in range(target_sum + 1):
 # 不选择当前元素
 dp[i][j] = dp[i - 1][j]
 # 选择当前元素 (如果 j >= num)
 if j >= num:
 dp[i][j] += dp[i - 1][j - num]

return dp[n][target_sum]

```

```
def find_target_sum_ways_backtrack(nums: list[int], target: int) -> int:
```

使用回溯法的实现

注意: 这种方法对于大数组可能效率不高

```
count = [0] # 使用列表来存储结果, 以便在递归中修改
```

```

def backtrack(index, current_sum):
 # 已经处理完所有元素
 if index == len(nums):
 if current_sum == target:
 count[0] += 1
 return

 # 尝试加上当前元素
 backtrack(index + 1, current_sum + nums[index])

 # 尝试减去当前元素

```

```

backtrack(index + 1, current_sum - nums[index])

backtrack(0, 0)
return count[0]

def find_target_sum_ways_memo(nums: list[int], target: int) -> int:
"""
使用记忆化递归的实现
"""

创建记忆化缓存，键为(index, current_sum)，值为对应的方法数
memo = {}

def backtrack_memo(index, current_sum):
 # 已经处理完所有元素
 if index == len(nums):
 return 1 if current_sum == target else 0

 # 生成缓存键
 key = (index, current_sum)

 # 检查是否已经计算过
 if key in memo:
 return memo[key]

 # 计算两种情况的结果之和
 add = backtrack_memo(index + 1, current_sum + nums[index])
 subtract = backtrack_memo(index + 1, current_sum - nums[index])

 # 存储结果到缓存
 memo[key] = add + subtract

 return add + subtract

return backtrack_memo(0, 0)

def find_target_sum_ways_optimized(nums: list[int], target: int) -> int:
"""
优化版本，考虑数组中包含 0 的情况
"""

n = len(nums)
if n == 0:
 return 1 if target == 0 else 0

```

```

total_sum = 0
zero_count = 0

计算总和和 0 的个数
for num in nums:
 total_sum += num
 if num == 0:
 zero_count += 1

检查是否有解
if total_sum < abs(target) or (total_sum + target) % 2 != 0:
 return 0

target_sum = (total_sum + target) // 2
if target_sum < 0:
 return 0

过滤掉 0, 单独处理
non_zero_nums = [num for num in nums if num != 0]

创建 DP 数组
dp = [0] * (target_sum + 1)
dp[0] = 1

填充 DP 数组
for num in non_zero_nums:
 for j in range(target_sum, num - 1, -1):
 dp[j] += dp[j - num]

每个 0 有两种选择 (+0 或 -0), 所以总方法数乘以 2^zero_count
return dp[target_sum] * (2 ** zero_count)

```

```

def print_all_expressions(nums: list[int], target: int) -> list[str]:
 """

```

打印所有可能的表达式

注意：这个方法仅用于教学目的，对于大数组可能效率不高

参数：

nums：整数数组

target：目标值

返回：

所有可能的表达式列表

```

"""
result = []

def backtrack_expressions(index, current_sum, current_expr):
 if index == len(nums):
 if current_sum == target:
 result.append(current_expr)
 return

 num = nums[index]

 # 尝试加上当前元素
 backtrack_expressions(index + 1, current_sum + num,
 current_expr + '+' + str(num))

 # 尝试减去当前元素
 backtrack_expressions(index + 1, current_sum - num,
 current_expr + '-' + str(num))

 # 第一个数特殊处理，不需要前面的符号
 if nums:
 backtrack_expressions(1, nums[0], str(nums[0]))

 # 尝试第一个数为负数的情况
 backtrack_expressions(1, -nums[0], '-' + str(nums[0]))

print("所有可能的表达式:")
for expr in result:
 print(expr)
print(f"总共有 {len(result)} 种不同的表达式。")

return result

def find_target_sum_ways_bfs(nums: list[int], target: int) -> int:
"""
使用广度优先搜索(BFS)的实现
"""

from collections import defaultdict

BFS 队列，存储当前索引和当前和
queue = defaultdict(int)
queue[0] = 1 # 初始状态：和为 0，方法数为 1

```

```
逐个处理数组元素
for num in nums:
 next_queue = defaultdict(int)
 for current_sum, count in queue.items():
 # 加上当前元素
 next_queue[current_sum + num] += count
 # 减去当前元素
 next_queue[current_sum - num] += count
 queue = next_queue

返回目标和对应的方法数
return queue.get(target, 0)

测试代码
if __name__ == "__main__":
 # 测试用例 1
 nums1 = [1, 1, 1, 1, 1]
 target1 = 3
 print(f"测试用例 1 结果: {find_target_sum_ways(nums1, target1)} (预期: 5)")
 print_all_expressions(nums1, target1)
 print("-----")

 # 测试用例 2
 nums2 = [1]
 target2 = 1
 print(f"测试用例 2 结果: {find_target_sum_ways(nums2, target2)} (预期: 1)")
 print("-----")

 # 测试用例 3
 nums3 = [1, 2, 3, 4, 5]
 target3 = 3
 print(f"测试用例 3 结果: {find_target_sum_ways(nums3, target3)} (预期: 5)")
 print("-----")

 # 测试用例 4 - 无法满足的情况
 nums4 = [1, 2, 3]
 target4 = 7
 print(f"测试用例 4 结果: {find_target_sum_ways(nums4, target4)} (预期: 0)")

 # 测试用例 5 - 包含 0 的情况
 nums5 = [0, 0, 0, 0, 0, 0, 0, 0, 1]
 target5 = 1
 print(f"\n测试用例 5 结果 (包含 0): {find_target_sum_ways(nums5, target5)} (预期: 256)")
```

```

测试各种实现方法
print("\n 测试各种实现方法:")
print(f"二维 DP 版本: {find_target_sum_ways_2d(nums1, target1)}")
print(f"回溯版本: {find_target_sum_ways_backtrack(nums1, target1)}")
print(f"记忆化递归版本: {find_target_sum_ways_memo(nums1, target1)}")
print(f"优化版本: {find_target_sum_ways_optimized(nums1, target1)}")
print(f"BFS 版本: {find_target_sum_ways_bfs(nums1, target1)}")

测试大数组性能对比
print("\n 测试大数组性能对比:")
import time

创建一个较大的测试用例
large_nums = [1] * 20 # 20 个 1
large_target = 10

测试 DP 版本
start_time = time.time()
dp_result = find_target_sum_ways(large_nums, large_target)
dp_time = time.time() - start_time
print(f"DP 版本结果: {dp_result}, 耗时: {dp_time:.6f} 秒")

测试优化版本
start_time = time.time()
optimized_result = find_target_sum_ways_optimized(large_nums, large_target)
optimized_time = time.time() - start_time
print(f"优化版本结果: {optimized_result}, 耗时: {optimized_time:.6f} 秒")

注意: 回溯版本在大数组上可能需要很长时间, 这里不进行测试
=====

文件: Code43_CoinChange.cpp
=====

#include <iostream>
#include <vector>
#include <queue>
#include <algorithm>
#include <climits>

// LeetCode 322. 零钱兑换
// 题目描述: 给你一个整数数组 coins , 表示不同面额的硬币; 以及一个整数 amount , 表示总金额。

```

```

// LeetCode 322. 零钱兑换
// 题目描述: 给你一个整数数组 coins , 表示不同面额的硬币; 以及一个整数 amount , 表示总金额。

```

```
// 计算并返回可以凑成总金额所需的 最少的硬币个数 。如果没有任何一种硬币组合能组成总金额，返回 -1 。
// 你可以认为每种硬币的数量是无限的。
// 链接: https://leetcode.cn/problems/coin-change/
//
// 解题思路:
// 这是一个完全背包问题。我们需要找到凑成总金额 amount 所需的最少硬币数量，每种硬币可以重复使用。
//
// 状态定义: dp[j] 表示凑成总金额 j 所需的最少硬币数量
// 状态转移方程: dp[j] = min(dp[j], dp[j - coin] + 1)，其中 coin 是当前硬币的面额，且 j >= coin
// 初始状态: dp[0] = 0，表示凑成总金额 0 所需的最少硬币数量为 0；对于其他 j，初始化为一个较大的值 (如 amount + 1)
//
// 时间复杂度: O(amount * n)，其中 amount 是总金额，n 是硬币的种类数
// 空间复杂度: O(amount)，使用一维 DP 数组

class Solution {
public:
 // 基础 DP 解法
 int coinChange(std::vector<int>& coins, int amount) {
 if (amount < 0 || coins.empty()) {
 return amount == 0 ? 0 : -1;
 }

 // 创建 DP 数组，dp[j] 表示凑成总金额 j 所需的最少硬币数量
 // 初始化为一个较大的值 (amount + 1)，因为最多需要 amount 个 1 元硬币
 std::vector<int> dp(amount + 1, amount + 1);

 // 初始状态：凑成总金额 0 所需的最少硬币数量为 0
 dp[0] = 0;

 // 填充 DP 数组
 // 遍历硬币
 for (int coin : coins) {
 // 遍历金额
 for (int j = coin; j <= amount; j++) {
 // 更新凑成总金额 j 所需的最少硬币数量
 dp[j] = std::min(dp[j], dp[j - coin] + 1);
 }
 }

 // 如果 dp[amount] 仍然是初始值，说明无法凑出总金额
 }
}
```

```

 return dp[amount] > amount ? -1 : dp[amount];
 }

// 从金额角度出发的实现
int coinChange2(std::vector<int>& coins, int amount) {
 if (amount < 0 || coins.empty()) {
 return amount == 0 ? 0 : -1;
 }

 std::vector<int> dp(amount + 1, amount + 1);
 dp[0] = 0;

 // 遍历金额
 for (int j = 1; j <= amount; j++) {
 // 遍历硬币
 for (int coin : coins) {
 if (coin <= j) {
 dp[j] = std::min(dp[j], dp[j - coin] + 1);
 }
 }
 }

 return dp[amount] > amount ? -1 : dp[amount];
}

// 广度优先搜索(BFS)实现
int coinChangeBFS(std::vector<int>& coins, int amount) {
 if (amount == 0) {
 return 0;
 }
 if (coins.empty() || amount < 0) {
 return -1;
 }

 // 使用队列进行 BFS
 std::queue<int> queue;
 // 使用数组记录访问过的金额，避免重复计算
 std::vector<bool> visited(amount + 1, false);

 // 将 0 加入队列，表示初始金额
 queue.push(0);
 visited[0] = true;

```

```

int level = 0; // 记录层数，即硬币数量

while (!queue.empty()) {
 int size = queue.size();
 level++;

 for (int i = 0; i < size; i++) {
 int current = queue.front();
 queue.pop();

 // 尝试每种硬币
 for (int coin : coins) {
 int next = current + coin;

 // 如果达到目标金额，返回当前层数
 if (next == amount) {
 return level;
 }

 // 如果没有超过目标金额且未访问过，则加入队列
 if (next < amount && !visited[next]) {
 visited[next] = true;
 queue.push(next);
 }
 }
 }
}

// 无法凑出总金额
return -1;
}

// 贪心 + 回溯 实现（使用引用传递 result）
int coinChangeGreedy(std::vector<int>& coins, int amount) {
 if (amount == 0) {
 return 0;
 }

 if (coins.empty() || amount < 0) {
 return -1;
 }

 // 按面额降序排序
 std::sort(coins.begin(), coins.end(), std::greater<int>());

```

```

int result = INT_MAX;

// 回溯搜索
backtrack(coins, amount, 0, 0, result);

return result == INT_MAX ? -1 : result;
}

// 记忆化搜索实现
int coinChangeMemo(std::vector<int>& coins, int amount) {
 if (amount == 0) {
 return 0;
 }
 if (coins.empty() || amount < 0) {
 return -1;
 }

 // 按面额降序排序，有助于剪枝
 std::sort(coins.begin(), coins.end(), std::greater<int>());

 // 创建记忆化缓存
 std::vector<int> memo(amount + 1, 0);
 return backtrackMemo(coins, amount, memo);
}

// 优化版本，提前过滤掉大于 amount 的硬币
int coinChangeOptimized(std::vector<int>& coins, int amount) {
 if (amount < 0 || coins.empty()) {
 return amount == 0 ? 0 : -1;
 }

 // 过滤掉大于 amount 的硬币
 std::vector<int> filteredCoins;
 for (int coin : coins) {
 if (coin <= amount) {
 filteredCoins.push_back(coin);
 }
 }

 if (filteredCoins.empty()) {
 return amount == 0 ? 0 : -1;
 }
}

```

```

std::vector<int> dp(amount + 1, amount + 1);
dp[0] = 0;

for (int coin : filteredCoins) {
 for (int j = coin; j <= amount; j++) {
 dp[j] = std::min(dp[j], dp[j - coin] + 1);
 }
}

return dp[amount] > amount ? -1 : dp[amount];
}

// 打印出一种最优的硬币组合
void printOptimalCoins(std::vector<int>& coins, int amount) {
 if (amount == 0) {
 std::cout << "无需硬币" << std::endl;
 return;
 }

 if (coins.empty() || amount < 0) {
 std::cout << "无法凑出总金额" << std::endl;
 return;
 }

 // 计算最少硬币数量
 std::vector<int> dp(amount + 1, amount + 1);
 dp[0] = 0;

 // 额外创建一个数组，用于记录每个金额使用的最后一个硬币
 std::vector<int> lastCoin(amount + 1, -1);

 // 填充 DP 数组
 for (int coin : coins) {
 for (int j = coin; j <= amount; j++) {
 if (dp[j] > dp[j - coin] + 1) {
 dp[j] = dp[j - coin] + 1;
 lastCoin[j] = coin;
 }
 }
 }

 if (dp[amount] > amount) {
 std::cout << "无法凑出总金额" << std::endl;
 return;
 }
}

```

```

}

// 回溯构建最优硬币组合
std::vector<int> result;
int current = amount;
while (current > 0) {
 int coin = lastCoin[current];
 result.push_back(coin);
 current -= coin;
}

// 输出结果
std::cout << "最优硬币组合: ";
for (size_t i = 0; i < result.size(); i++) {
 std::cout << result[i];
 if (i < result.size() - 1) {
 std::cout << " + ";
 }
}
std::cout << " = " << amount << std::endl;
std::cout << "最少硬币数量: " << dp[amount] << std::endl;
}

private:
// 回溯辅助方法 (使用引用传递 result)
void backtrack(const std::vector<int>& coins, int amount, int index, int count, int& result)
{
 // 已经找到一个解，或者无法继续使用更大的硬币
 if (amount == 0) {
 result = std::min(result, count);
 return;
 }
 if (index >= coins.size()) {
 return;
 }

 // 尝试使用当前硬币的最大可能数量
 for (int i = amount / coins[index]; i >= 0 && count + i < result; i--) {
 backtrack(coins, amount - i * coins[index], index + 1, count + i, result);
 }
}

// 记忆化搜索辅助方法

```

```

int backtrackMemo(const std::vector<int>& coins, int amount, std::vector<int>& memo) {
 if (amount == 0) {
 return 0;
 }
 if (amount < 0) {
 return -1;
 }

 // 检查是否已经计算过
 if (memo[amount] != 0) {
 return memo[amount];
 }

 int minCount = INT_MAX;

 // 尝试每种硬币
 for (int coin : coins) {
 int subResult = backtrackMemo(coins, amount - coin, memo);
 if (subResult >= 0 && subResult < minCount) {
 minCount = subResult + 1;
 }
 }

 // 记录结果
 memo[amount] = (minCount == INT_MAX) ? -1 : minCount;

 return memo[amount];
}

int main() {
 Solution solution;

 // 测试用例 1
 std::vector<int> coins1 = {1, 2, 5};
 int amount1 = 11;
 std::cout << "测试用例 1 结果: " << solution.coinChange(coins1, amount1) << " (预期: 3)" << std::endl;
 solution.printOptimalCoins(coins1, amount1);
 std::cout << "-----" << std::endl;

 // 测试用例 2
 std::vector<int> coins2 = {2};

```

```
int amount2 = 3;
std::cout << "测试用例 2 结果: " << solution.coinChange(coins2, amount2) << " (预期: -1)" <<
std::endl;
std::cout << "-----" << std::endl;

// 测试用例 3
std::vector<int> coins3 = {1};
int amount3 = 0;
std::cout << "测试用例 3 结果: " << solution.coinChange(coins3, amount3) << " (预期: 0)" <<
std::endl;
std::cout << "-----" << std::endl;

// 测试用例 4
std::vector<int> coins4 = {186, 419, 83, 408};
int amount4 = 6249;
std::cout << "测试用例 4 结果: " << solution.coinChange(coins4, amount4) << " (预期: 20)" <<
std::endl;

// 测试各种实现方法
std::cout << "\n 测试各种实现方法:" << std::endl;
std::cout << "方法 2: " << solution.coinChange2(coins1, amount1) << std::endl;
std::cout << "BFS 方法: " << solution.coinChangeBFS(coins1, amount1) << std::endl;
std::cout << "贪心回溯方法: " << solution.coinChangeGreedy(coins1, amount1) << std::endl;
std::cout << "记忆化搜索方法: " << solution.coinChangeMemo(coins1, amount1) << std::endl;
std::cout << "优化方法: " << solution.coinChangeOptimized(coins1, amount1) << std::endl;

// 测试特殊情况 - 硬币包含 0
std::cout << "\n 测试特殊情况 - 硬币包含 0:" << std::endl;
std::vector<int> coins5 = {0, 1, 2, 5};
std::cout << "结果: " << solution.coinChange(coins5, amount1) << std::endl;
std::cout << "优化方法结果: " << solution.coinChangeOptimized(coins5, amount1) << std::endl;

// 测试大额硬币
std::cout << "\n 测试大额硬币:" << std::endl;
std::vector<int> coins6 = {500, 100, 50, 10, 5, 1};
int amount6 = 12345;
std::cout << "大额硬币结果: " << solution.coinChange(coins6, amount6) << std::endl;

return 0;
}
```

---

文件: Code43\_CoinChange.java

```
=====
package class073;

// LeetCode 322. 零钱兑换
// 题目描述: 给你一个整数数组 coins，表示不同面额的硬币；以及一个整数 amount，表示总金额。
// 计算并返回可以凑成总金额所需的 最少的硬币个数。如果没有任何一种硬币组合能组成总金额，返回 -1。
// 你可以认为每种硬币的数量是无限的。
// 链接: https://leetcode.cn/problems/coin-change/
//
// 解题思路:
// 这是一个经典完全背包问题。我们需要找到凑成总金额 amount 所需的最少硬币数量，每种硬币可以重复使用。
//
// 状态定义: dp[j] 表示凑成总金额 j 所需的最少硬币数量
// 状态转移方程: dp[j] = min(dp[j], dp[j - coin] + 1)，其中 coin 是当前硬币的面额，且 j >= coin
// 初始状态: dp[0] = 0，表示凑成总金额 0 所需的最少硬币数量为 0；对于其他 j，初始化为一个较大的值（如 amount + 1）
//
// 时间复杂度: O(amount * n)，其中 amount 是总金额，n 是硬币的种类数
// 空间复杂度: O(amount)，使用一维 DP 数组

public class Code43_CoinChange {

 // 主方法，用于测试
 public static void main(String[] args) {
 // 测试用例 1
 int[] coins1 = {1, 2, 5};
 int amount1 = 11;
 System.out.println("测试用例 1 结果: " + coinChange(coins1, amount1)); // 预期输出: 3

 // 测试用例 2
 int[] coins2 = {2};
 int amount2 = 3;
 System.out.println("测试用例 2 结果: " + coinChange(coins2, amount2)); // 预期输出: -1

 // 测试用例 3
 int[] coins3 = {1};
 int amount3 = 0;
 System.out.println("测试用例 3 结果: " + coinChange(coins3, amount3)); // 预期输出: 0

 // 测试用例 4
 }
}
```

```

int[] coins4 = {186, 419, 83, 408};
int amount4 = 6249;
System.out.println("测试用例 4 结果: " + coinChange(coins4, amount4)); // 预期输出: 20
}

/***
 * 计算凑成总金额所需的最少硬币数量
 * @param coins 硬币面额数组
 * @param amount 总金额
 * @return 最少硬币数量, 如果无法凑出则返回-1
 */
public static int coinChange(int[] coins, int amount) {
 if (amount < 0 || coins == null || coins.length == 0) {
 return amount == 0 ? 0 : -1;
 }

 // 创建 DP 数组, dp[j] 表示凑成总金额 j 所需的最少硬币数量
 // 初始化为一个较大的值 (amount + 1), 因为最多需要 amount 个 1 元硬币
 int[] dp = new int[amount + 1];
 for (int j = 1; j <= amount; j++) {
 dp[j] = amount + 1;
 }

 // 初始状态: 凑成总金额 0 所需的最少硬币数量为 0
 dp[0] = 0;

 // 填充 DP 数组
 // 遍历硬币
 for (int coin : coins) {
 // 遍历金额
 for (int j = coin; j <= amount; j++) {
 // 更新凑成总金额 j 所需的最少硬币数量
 dp[j] = Math.min(dp[j], dp[j - coin] + 1);
 }
 }

 // 如果 dp[amount] 仍然是初始值, 说明无法凑出总金额
 return dp[amount] > amount ? -1 : dp[amount];
}

/***
 * 从金额角度出发的实现
 * 这种实现方式与上面的类似, 只是遍历顺序不同
*/

```

```

*/
public static int coinChange2(int[] coins, int amount) {
 if (amount < 0 || coins == null || coins.length == 0) {
 return amount == 0 ? 0 : -1;
 }

 int[] dp = new int[amount + 1];
 for (int j = 1; j <= amount; j++) {
 dp[j] = amount + 1;
 }
 dp[0] = 0;

 // 遍历金额
 for (int j = 1; j <= amount; j++) {
 // 遍历硬币
 for (int coin : coins) {
 if (coin <= j) {
 dp[j] = Math.min(dp[j], dp[j - coin] + 1);
 }
 }
 }
}

return dp[amount] > amount ? -1 : dp[amount];
}

/***
 * 广度优先搜索(BFS)实现
 * 将问题视为图的最短路径问题：从 0 到 amount 的最短路径
 */
public static int coinChangeBFS(int[] coins, int amount) {
 if (amount == 0) {
 return 0;
 }
 if (coins == null || coins.length == 0 || amount < 0) {
 return -1;
 }

 // 使用队列进行 BFS
 java.util.Queue<Integer> queue = new java.util.LinkedList<>();
 // 使用数组记录访问过的金额，避免重复计算
 boolean[] visited = new boolean[amount + 1];

 // 将 0 加入队列，表示初始金额

```

```

queue.offer(0);
visited[0] = true;

int level = 0; // 记录层数，即硬币数量

while (!queue.isEmpty()) {
 int size = queue.size();
 level++;

 for (int i = 0; i < size; i++) {
 int current = queue.poll();

 // 尝试每种硬币
 for (int coin : coins) {
 int next = current + coin;

 // 如果达到目标金额，返回当前层数
 if (next == amount) {
 return level;
 }

 // 如果没有超过目标金额且未访问过，则加入队列
 if (next < amount && !visited[next]) {
 visited[next] = true;
 queue.offer(next);
 }
 }
 }

 // 无法凑出总金额
 return -1;
}

/***
 * 贪心 + 回溯 实现
 * 注意：这种方法不一定能得到正确的结果，因为贪心策略不一定适用于所有硬币组合
 * 例如，对于 coins = [1, 3, 4], amount = 6，贪心会选择 [4, 1, 1] (3个硬币)，但最优解是 [3, 3] (2个硬币)
 */
public static int coinChangeGreedy(int[] coins, int amount) {
 if (amount == 0) {
 return 0;
}

```

```

 }

 if (coins == null || coins.length == 0 || amount < 0) {
 return -1;
 }

 // 按面额降序排序
 java.util.Arrays.sort(coins);
 int result = Integer.MAX_VALUE;

 // 回溯搜索
 backtrack(coins, amount, coins.length - 1, 0, result);

 return result == Integer.MAX_VALUE ? -1 : result;
}

/***
 * 回溯辅助方法
 */
private static void backtrack(int[] coins, int amount, int index, int count, int result) {
 // 已经找到一个解，或者无法继续使用更大的硬币
 if (amount == 0) {
 result = Math.min(result, count);
 return;
 }

 if (index < 0) {
 return;
 }

 // 尝试使用当前硬币的最大可能数量
 for (int i = amount / coins[index]; i >= 0 && count + i < result; i--) {
 backtrack(coins, amount - i * coins[index], index - 1, count + i, result);
 }
}

/***
 * 优化的回溯方法，使用记忆化搜索
 */
public static int coinChangeMemo(int[] coins, int amount) {
 if (amount == 0) {
 return 0;
 }

 if (coins == null || coins.length == 0 || amount < 0) {
 return -1;
 }
}

```

```
}

// 按面额降序排序，有助于剪枝
java.util.Arrays.sort(coins);

// 创建记忆化缓存
int[] memo = new int[amount + 1];
return backtrackMemo(coins, amount, memo);
}

/**
 * 记忆化搜索辅助方法
 */
private static int backtrackMemo(int[] coins, int amount, int[] memo) {
 if (amount == 0) {
 return 0;
 }
 if (amount < 0) {
 return -1;
 }

 // 检查是否已经计算过
 if (memo[amount] != 0) {
 return memo[amount];
 }

 int minCount = Integer.MAX_VALUE;

 // 尝试每种硬币
 for (int coin : coins) {
 int subResult = backtrackMemo(coins, amount - coin, memo);
 if (subResult >= 0 && subResult < minCount) {
 minCount = subResult + 1;
 }
 }

 // 记录结果
 memo[amount] = (minCount == Integer.MAX_VALUE) ? -1 : minCount;

 return memo[amount];
}

/**
```

```
* 打印出一种最优的硬币组合
*/
public static void printOptimalCoins(int[] coins, int amount) {
 if (amount == 0) {
 System.out.println("无需硬币");
 return;
 }
 if (coins == null || coins.length == 0 || amount < 0) {
 System.out.println("无法凑出总金额");
 return;
 }

 // 计算最少硬币数量
 int[] dp = new int[amount + 1];
 for (int j = 1; j <= amount; j++) {
 dp[j] = amount + 1;
 }
 dp[0] = 0;

 // 额外创建一个数组，用于记录每个金额使用的最后一个硬币
 int[] lastCoin = new int[amount + 1];

 // 填充 DP 数组
 for (int coin : coins) {
 for (int j = coin; j <= amount; j++) {
 if (dp[j] > dp[j - coin] + 1) {
 dp[j] = dp[j - coin] + 1;
 lastCoin[j] = coin;
 }
 }
 }

 if (dp[amount] > amount) {
 System.out.println("无法凑出总金额");
 return;
 }

 // 回溯构建最优硬币组合
 java.util.List<Integer> result = new java.util.ArrayList<>();
 int current = amount;
 while (current > 0) {
 int coin = lastCoin[current];
 result.add(coin);
 current -= coin;
 }
}
```

```

 current -= coin;
 }

// 输出结果
System.out.print("最优硬币组合: ");
for (int i = 0; i < result.size(); i++) {
 System.out.print(result.get(i));
 if (i < result.size() - 1) {
 System.out.print(" + ");
 }
}
System.out.println(" = " + amount);
System.out.println("最少硬币数量: " + dp[amount]);
}

/**
 * 优化版本，提前过滤掉大于 amount 的硬币
 */
public static int coinChangeOptimized(int[] coins, int amount) {
 if (amount < 0 || coins == null || coins.length == 0) {
 return amount == 0 ? 0 : -1;
 }

 // 过滤掉大于 amount 的硬币
 java.util.List<Integer> filteredCoins = new java.util.ArrayList<>();
 for (int coin : coins) {
 if (coin <= amount) {
 filteredCoins.add(coin);
 }
 }

 if (filteredCoins.isEmpty()) {
 return amount == 0 ? 0 : -1;
 }

 // 转换回数组
 int[] filteredCoinsArray = new int[filteredCoins.size()];
 for (int i = 0; i < filteredCoins.size(); i++) {
 filteredCoinsArray[i] = filteredCoins.get(i);
 }

 int[] dp = new int[amount + 1];
 for (int j = 1; j <= amount; j++) {

```

```

 dp[j] = amount + 1;
 }
 dp[0] = 0;

 for (int coin : filteredCoinsArray) {
 for (int j = coin; j <= amount; j++) {
 dp[j] = Math.min(dp[j], dp[j - coin] + 1);
 }
 }

 return dp[amount] > amount ? -1 : dp[amount];
}
}

```

=====

文件: Code43\_CoinChange.py

=====

```

LeetCode 322. 零钱兑换
题目描述: 给你一个整数数组 coins，表示不同面额的硬币；以及一个整数 amount，表示总金额。
计算并返回可以凑成总金额所需的 最少的硬币个数。如果没有任何一种硬币组合能组成总金额，返回 -1。
你可以认为每种硬币的数量是无限的。
链接: https://leetcode.cn/problems/coin-change/
#
解题思路:
这是一个经典完全背包问题。我们需要找到凑成总金额 amount 所需的最少硬币数量，每种硬币可以重复使用。
#
状态定义: dp[j] 表示凑成总金额 j 所需的最少硬币数量
状态转移方程: dp[j] = min(dp[j], dp[j - coin] + 1)，其中 coin 是当前硬币的面额，且 j >= coin
初始状态: dp[0] = 0，表示凑成总金额 0 所需的最少硬币数量为 0；对于其他 j，初始化为一个较大的值（如 amount + 1）
#
时间复杂度: O(amount * n)，其中 amount 是总金额，n 是硬币的种类数
空间复杂度: O(amount)，使用一维 DP 数组

```

```

def coin_change(coins: list[int], amount: int) -> int:
 """

```

计算凑成总金额所需的最少硬币数量

参数:

coins: 硬币面额数组

```
amount: 总金额
```

返回：

最少硬币数量，如果无法凑出则返回-1

```
"""
```

```
if amount < 0 or not coins:
```

```
 return 0 if amount == 0 else -1
```

```
创建 DP 数组，dp[j] 表示凑成总金额 j 所需的最少硬币数量
```

```
初始化为一个较大的值 (amount + 1)，因为最多需要 amount 个 1 元硬币
```

```
dp = [amount + 1] * (amount + 1)
```

```
初始状态：凑成总金额 0 所需的最少硬币数量为 0
```

```
dp[0] = 0
```

```
填充 DP 数组
```

```
遍历硬币
```

```
for coin in coins:
```

```
 # 遍历金额
```

```
 for j in range(coin, amount + 1):
```

```
 # 更新凑成总金额 j 所需的最少硬币数量
```

```
 dp[j] = min(dp[j], dp[j - coin] + 1)
```

```
如果 dp[amount] 仍然是初始值，说明无法凑出总金额
```

```
return dp[amount] if dp[amount] <= amount else -1
```

```
def coin_change2(coins: list[int], amount: int) -> int:
```

```
"""
```

从金额角度出发的实现

这种实现方式与上面的类似，只是遍历顺序不同

```
"""
```

```
if amount < 0 or not coins:
```

```
 return 0 if amount == 0 else -1
```

```
dp = [amount + 1] * (amount + 1)
```

```
dp[0] = 0
```

```
遍历金额
```

```
for j in range(1, amount + 1):
```

```
 # 遍历硬币
```

```
 for coin in coins:
```

```
 if coin <= j:
```

```
 dp[j] = min(dp[j], dp[j - coin] + 1)
```

```

return dp[amount] if dp[amount] <= amount else -1

def coin_change_bfs(coins: list[int], amount: int) -> int:
 """
 广度优先搜索(BFS)实现
 将问题视为图的最短路径问题：从 0 到 amount 的最短路径
 """
 if amount == 0:
 return 0
 if not coins or amount < 0:
 return -1

 # 使用集合记录访问过的金额，避免重复计算
 visited = set()
 visited.add(0)

 # BFS 队列，存储当前金额和步数
 queue = [(0, 0)] # (current_amount, steps)

 while queue:
 current, steps = queue.pop(0)
 steps += 1

 # 尝试每种硬币
 for coin in coins:
 next_amount = current + coin

 # 如果达到目标金额，返回当前步数
 if next_amount == amount:
 return steps

 # 如果没有超过目标金额且未访问过，则加入队列
 if next_amount < amount and next_amount not in visited:
 visited.add(next_amount)
 queue.append((next_amount, steps))

 # 无法凑出总金额
 return -1

def coin_change_greedy(coins: list[int], amount: int) -> int:
 """
 贪心 + 回溯 实现

```

注意：这种方法不一定能得到正确的结果，因为贪心策略不一定适用于所有硬币组合

例如，对于 coins = [1, 3, 4]，amount = 6，贪心会选择 [4, 1, 1] (3个硬币)，但最优解是 [3, 3] (2个硬币)

"""

```
if amount == 0:
```

```
 return 0
```

```
if not coins or amount < 0:
```

```
 return -1
```

# 按面额降序排序

```
coins.sort(reverse=True)
```

```
result = float('inf')
```

```
def backtrack(amount_remaining, coin_index, count):
```

```
 nonlocal result
```

# 已经找到一个解，或者无法继续使用更大的硬币

```
 if amount_remaining == 0:
```

```
 result = min(result, count)
```

```
 return
```

```
 if coin_index >= len(coins):
```

```
 return
```

# 尝试使用当前硬币的最大可能数量

```
 max_usage = amount_remaining // coins[coin_index]
```

# 从最大可能数量开始尝试，直到 0

```
 for i in range(max_usage, -1, -1):
```

# 剪枝：如果当前数量已经超过了已知的最优解，则停止尝试

```
 if count + i >= result:
```

```
 break
```

```
 backtrack(amount_remaining - i * coins[coin_index], coin_index + 1, count + i)
```

```
backtrack(amount, 0, 0)
```

```
return result if result != float('inf') else -1
```

```
def coin_change_memo(coins: list[int], amount: int) -> int:
```

"""

优化的回溯方法，使用记忆化搜索

"""

```
if amount == 0:
```

```
 return 0
```

```
if not coins or amount < 0:
```

```
 return -1
```

```

创建记忆化缓存，初始化为-2（表示未计算过）
memo = [-2] * (amount + 1)
memo[0] = 0 # 基础情况：凑成 0 元需要 0 个硬币

def backtrack_memo(remaining):
 if remaining == 0:
 return 0
 if remaining < 0:
 return -1

 # 检查是否已经计算过
 if memo[remaining] != -2:
 return memo[remaining]

 min_count = float('inf')

 # 尝试每种硬币
 for coin in coins:
 sub_result = backtrack_memo(remaining - coin)
 if sub_result >= 0 and sub_result < min_count:
 min_count = sub_result + 1

 # 记录结果
 memo[remaining] = min_count if min_count != float('inf') else -1

 return memo[remaining]

return backtrack_memo(amount)

def print_optimal_coins(coins: list[int], amount: int):
 """
 打印出一种最优的硬币组合
 """
 if amount == 0:
 print("无需硬币")
 return
 if not coins or amount < 0:
 print("无法凑出总金额")
 return

 # 计算最少硬币数量
 dp = [amount + 1] * (amount + 1)

```

```

dp[0] = 0

额外创建一个数组，用于记录每个金额使用的最后一个硬币
last_coin = [0] * (amount + 1)

填充 DP 数组
for coin in coins:
 for j in range(coin, amount + 1):
 if dp[j] > dp[j - coin] + 1:
 dp[j] = dp[j - coin] + 1
 last_coin[j] = coin

if dp[amount] > amount:
 print("无法凑出总金额")
 return

回溯构建最优硬币组合
result = []
current = amount
while current > 0:
 coin = last_coin[current]
 result.append(coin)
 current -= coin

输出结果
print("最优硬币组合: ", end="")
print(" + ".join(map(str, result)), end=" = ")
print(amount)
print(f"最少硬币数量: {dp[amount]}")

def coin_change_optimized(coins: list[int], amount: int) -> int:
 """
 优化版本，提前过滤掉大于 amount 的硬币
 """

 if amount < 0 or not coins:
 return 0 if amount == 0 else -1

 # 过滤掉大于 amount 的硬币
 filtered_coins = [coin for coin in coins if coin <= amount]

 if not filtered_coins:
 return 0 if amount == 0 else -1

```

```
dp = [amount + 1] * (amount + 1)
dp[0] = 0

for coin in filtered_coins:
 for j in range(coin, amount + 1):
 dp[j] = min(dp[j], dp[j - coin] + 1)

return dp[amount] if dp[amount] <= amount else -1

测试代码
if __name__ == "__main__":
 # 测试用例 1
 coins1 = [1, 2, 5]
 amount1 = 11
 print(f"测试用例 1 结果: {coin_change(coins1, amount1)} (预期: 3)")
 print_optimal_coins(coins1, amount1)
 print("-----")

 # 测试用例 2
 coins2 = [2]
 amount2 = 3
 print(f"测试用例 2 结果: {coin_change(coins2, amount2)} (预期: -1)")
 print("-----")

 # 测试用例 3
 coins3 = [1]
 amount3 = 0
 print(f"测试用例 3 结果: {coin_change(coins3, amount3)} (预期: 0)")
 print("-----")

 # 测试用例 4
 coins4 = [186, 419, 83, 408]
 amount4 = 6249
 print(f"测试用例 4 结果: {coin_change(coins4, amount4)} (预期: 20)")

测试各种实现方法
print("\n 测试各种实现方法:")
print(f"方法 2: {coin_change2(coins1, amount1)}")
print(f"BFS 方法: {coin_change_bfs(coins1, amount1)}")
print(f"贪心回溯方法: {coin_change_greedy(coins1, amount1)}")
print(f"记忆化搜索方法: {coin_change_memo(coins1, amount1)}")
print(f"优化方法: {coin_change_optimized(coins1, amount1)})")
```

```

测试特殊情况 - 硬币包含 0
print("\n 测试特殊情况 - 硬币包含 0:")
coins5 = [0, 1, 2, 5]
print(f"结果: {coin_change(coins5, amount1)}")
print(f"优化方法结果: {coin_change_optimized(coins5, amount1)}")

测试大额硬币
print("\n 测试大额硬币:")
coins6 = [500, 100, 50, 10, 5, 1]
amount6 = 12345
print(f"大额硬币结果: {coin_change(coins6, amount6)}")

性能测试
print("\n 性能测试:")
import time

创建一个较大的测试用例
large_amount = 1000
start_time = time.time()
dp_result = coin_change(coins6, large_amount)
dp_time = time.time() - start_time
print(f"DP 方法结果: {dp_result}, 耗时: {dp_time:.6f} 秒")

测试 BFS 方法
start_time = time.time()
bfs_result = coin_change_bfs(coins6, large_amount)
bfs_time = time.time() - start_time
print(f"BFS 方法结果: {bfs_result}, 耗时: {bfs_time:.6f} 秒")

测试记忆化搜索方法
start_time = time.time()
memo_result = coin_change_memo(coins6, large_amount)
memo_time = time.time() - start_time
print(f"记忆化搜索方法结果: {memo_result}, 耗时: {memo_time:.6f} 秒")
=====
```

文件: Code44\_PerfectSquares.cpp

```
=====
#include <iostream>
#include <vector>
#include <algorithm>
#include <climits>
```

```

#include <cmath>
#include <queue>
#include <unordered_set>

using namespace std;

// LeetCode 279. 完全平方数
// 题目描述：给定正整数 n，找到若干个完全平方数（比如 1, 4, 9, 16, ...）使得它们的和等于 n。
// 你需要让组成和的完全平方数的个数最少。
// 链接: https://leetcode.cn/problems/perfect-squares/
//
// 解题思路：
// 这是一个完全背包问题，其中：
// - 背包容量：正整数 n
// - 物品：完全平方数 (1, 4, 9, 16, ...)
// - 每个物品可以无限次使用（完全背包）
// - 目标：使用最少数量的物品（完全平方数）装满背包
//
// 状态定义：dp[i] 表示和为 i 的完全平方数的最少数量
// 状态转移方程：dp[i] = min(dp[i], dp[i - j*j] + 1)，其中 j*j <= i
// 初始状态：dp[0] = 0, dp[i] = INT_MAX (表示不可达)
//
// 时间复杂度：O(n * √n)，其中 n 是给定的正整数
// 空间复杂度：O(n)，使用一维 DP 数组
//
// 工程化考量：
// 1. 异常处理：处理 n <= 0 的情况
// 2. 边界条件：n=0 时返回 0, n=1 时返回 1
// 3. 性能优化：预生成完全平方数列表
// 4. 可读性：清晰的变量命名和注释

class Solution {
public:
 /**
 * 动态规划解法 - 完全背包问题
 * @param n 目标正整数
 * @return 组成 n 的最少完全平方数个数
 */
 int numSquares(int n) {
 // 参数验证
 if (n <= 0) {
 throw invalid_argument("n must be positive");
 }

```

```

// 特殊情况处理
if (n == 1) return 1;

// 创建 DP 数组, dp[i] 表示和为 i 的最少完全平方数个数
vector<int> dp(n + 1, INT_MAX);
dp[0] = 0;

// 遍历所有可能的完全平方数
for (int i = 1; i * i <= n; i++) {
 int square = i * i;
 // 完全背包: 正序遍历容量
 for (int j = square; j <= n; j++) {
 // 避免整数溢出
 if (dp[j - square] != INT_MAX) {
 dp[j] = min(dp[j], dp[j - square] + 1);
 }
 }
}

return dp[n];
}

/**
 * 优化的动态规划解法 - 预先生成完全平方数列表
 * @param n 目标正整数
 * @return 组成 n 的最少完全平方数个数
 */
int numSquaresOptimized(int n) {
 if (n <= 0) throw invalid_argument("n must be positive");

 // 预先生成所有可能的完全平方数
 int maxSquareRoot = static_cast<int>(sqrt(n));
 vector<int> squares;
 for (int i = 1; i <= maxSquareRoot; i++) {
 squares.push_back(i * i);
 }

 vector<int> dp(n + 1, INT_MAX);
 dp[0] = 0;

 // 先遍历物品 (完全平方数), 再遍历容量
 for (int square : squares) {

```

```

 for (int j = square; j <= n; j++) {
 if (dp[j - square] != INT_MAX) {
 dp[j] = min(dp[j], dp[j - square] + 1);
 }
 }

 }

 return dp[n];
}

/***
 * 数学解法 - 利用四平方定理
 * 拉格朗日四平方定理: 每个正整数都可以表示为 4 个整数的平方和
 * 勒让德三平方定理: 当且仅当 $n \neq 4^a(8b+7)$ 时, n 可以表示为 3 个整数的平方和
 * @param n 目标正整数
 * @return 组成 n 的最少完全平方数个数
 */
int numSquaresMath(int n) {
 // 检查 n 是否是完全平方数
 if (isPerfectSquare(n)) {
 return 1;
 }

 // 检查是否满足勒让德三平方定理的排除条件
 if (checkLegendreThreeSquare(n)) {
 return 4;
 }

 // 检查是否可以表示为两个平方数之和
 for (int i = 1; i * i <= n; i++) {
 int j = n - i * i;
 if (isPerfectSquare(j)) {
 return 2;
 }
 }

 // 其他情况返回 3
 return 3;
}

/***
 * 判断一个数是否是完全平方数
 */

```

```

bool isPerfectSquare(int x) {
 int sqrt_val = static_cast<int>(sqrt(x));
 return sqrt_val * sqrt_val == x;
}

/***
 * 检查是否满足勒让德三平方定理的排除条件
 * 即 $n = 4^a(8b+7)$
 */
bool checkLegendreThreeSquare(int n) {
 while (n % 4 == 0) {
 n /= 4;
 }
 return n % 8 == 7;
}

/***
 * BFS 解法 - 将问题转化为图的最短路径问题
 * 每个数字是一个节点，如果两个数字相差一个完全平方数，则它们之间有边
 */
int numSquaresBFS(int n) {
 if (n <= 0) throw invalid_argument("n must be positive");

 // 使用队列进行 BFS
 queue<int> q;
 // 记录到达每个数字的最短步数
 vector<int> steps(n + 1, -1);

 // 从 0 开始
 q.push(0);
 steps[0] = 0;

 while (!q.empty()) {
 int current = q.front();
 q.pop();

 // 尝试所有可能的完全平方数
 for (int i = 1; i * i <= n - current; i++) {
 int next = current + i * i;

 // 如果超出范围或已经访问过，跳过
 if (next > n || steps[next] != -1) {
 continue;
 }

 steps[next] = steps[current] + 1;
 q.push(next);
 }
 }

 return steps[n];
}

```

```

 }

 steps[next] = steps[current] + 1;

 // 如果到达目标，直接返回
 if (next == n) {
 return steps[next];
 }

 q.push(next);
}

return steps[n];
}

};

// 测试函数
void testPerfectSquares() {
 Solution sol;

 // 测试用例
 vector<int> testCases = {12, 13, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13};

 cout << "完全平方数问题测试: " << endl;
 for (int n : testCases) {
 int result1 = sol.numSquares(n);
 int result2 = sol.numSquaresOptimized(n);
 int result3 = sol.numSquaresBFS(n);
 int result4 = sol.numSquaresMath(n);

 cout << "n=" << n << ": DP=" << result1
 << ", Optimized=" << result2
 << ", BFS=" << result3
 << ", Math=" << result4 << endl;
 }

 // 验证所有方法结果一致
 if (result1 != result2 || result2 != result3 || result3 != result4) {
 cout << "警告: 不同方法结果不一致! " << endl;
 }
}

// 性能测试

```

```

auto startTime = chrono::high_resolution_clock::now();
int largeResult = sol.numSquares(10000);
auto endTime = chrono::high_resolution_clock::now();
auto duration = chrono::duration_cast<chrono::milliseconds>(endTime - startTime);

cout << "n=10000 的结果: " << largeResult << ", 耗时: " << duration.count() << "ms" << endl;
}

int main() {
 try {
 testPerfectSquares();
 } catch (const exception& e) {
 cerr << "错误: " << e.what() << endl;
 return 1;
 }
 return 0;
}

/*
 * 复杂度分析:
 *
 * 方法 1: 动态规划 (完全背包)
 * - 时间复杂度: O(n * √n)
 * - 外层循环: √n 次 (完全平方数的个数)
 * - 内层循环: n 次 (背包容量)
 * - 空间复杂度: O(n)
 *
 * 方法 2: 优化的动态规划
 * - 时间复杂度: O(n * √n) (与方法 1 相同, 但常数更小)
 * - 空间复杂度: O(n)
 *
 * 方法 3: 数学解法
 * - 时间复杂度: O(√n)
 * - 检查完全平方数: O(1)
 * - 检查勒让德条件: O(log n)
 * - 检查两个平方数之和: O(√n)
 * - 空间复杂度: O(1)
 *
 * 方法 4: BFS 解法
 * - 时间复杂度: O(n * √n) (最坏情况)
 * - 空间复杂度: O(n)
 *
 * 最优解分析:

```

```
* - 对于小规模 n (n < 1000): 所有方法都很快
* - 对于大规模 n (n >= 10000): 数学解法最优, 时间复杂度最低
* - 在实际工程中: 推荐使用动态规划, 代码清晰易懂
*
* C++特定优化:
* 1. 使用 vector 代替数组, 更安全
* 2. 使用 chrono 进行精确性能测试
* 3. 异常处理使用 C++标准异常
* 4. 使用 static_cast 进行安全的类型转换
*/
=====
```

文件: Code44\_PerfectSquares.java

```
=====
package class073;

import java.util.Arrays;

// LeetCode 279. 完全平方数
// 题目描述: 给定正整数 n, 找到若干个完全平方数 (比如 1, 4, 9, 16, ...) 使得它们的和等于 n。
// 你需要让组成和的完全平方数的个数最少。
// 链接: https://leetcode.cn/problems/perfect-squares/
//
// 解题思路:
// 这是一个完全背包问题, 其中:
// - 背包容量: 正整数 n
// - 物品: 完全平方数 (1, 4, 9, 16, ...)
// - 每个物品可以无限次使用 (完全背包)
// - 目标: 使用最少数量的物品 (完全平方数) 装满背包
//
// 状态定义: dp[i] 表示和为 i 的完全平方数的最少数量
// 状态转移方程: dp[i] = min(dp[i], dp[i - j*j] + 1), 其中 j*j <= i
// 初始状态: dp[0] = 0, dp[i] = Integer.MAX_VALUE (表示不可达)
//
// 时间复杂度: O(n * √n), 其中 n 是给定的正整数
// 空间复杂度: O(n), 使用一维 DP 数组
//
// 工程化考量:
// 1. 异常处理: 处理 n <= 0 的情况
// 2. 边界条件: n=0 时返回 0, n=1 时返回 1
// 3. 性能优化: 预先生成完全平方数列表
// 4. 可读性: 清晰的变量命名和注释
=====
```

```
public class Code44_PerfectSquares {

 /**
 * 动态规划解法 - 完全背包问题
 * @param n 目标正整数
 * @return 组成 n 的最少完全平方数个数
 */

 public static int numSquares(int n) {
 // 参数验证
 if (n <= 0) {
 throw new IllegalArgumentException("n must be positive");
 }

 // 特殊情况处理
 if (n == 1) return 1;

 // 创建 DP 数组, dp[i] 表示和为 i 的最少完全平方数个数
 int[] dp = new int[n + 1];

 // 初始化 DP 数组, 除了 dp[0]=0 外, 其他初始化为最大值
 Arrays.fill(dp, Integer.MAX_VALUE);
 dp[0] = 0;

 // 遍历所有可能的完全平方数
 for (int i = 1; i * i <= n; i++) {
 int square = i * i;
 // 完全背包: 正序遍历容量
 for (int j = square; j <= n; j++) {
 // 避免整数溢出
 if (dp[j - square] != Integer.MAX_VALUE) {
 dp[j] = Math.min(dp[j], dp[j - square] + 1);
 }
 }
 }

 return dp[n];
 }

 /**
 * 优化的动态规划解法 - 预先生成完全平方数列表
 * @param n 目标正整数
 * @return 组成 n 的最少完全平方数个数
 */
}
```

```

*/
public static int numSquaresOptimized(int n) {
 if (n <= 0) throw new IllegalArgumentException("n must be positive");

 // 预先生成所有可能的完全平方数
 int maxSquareRoot = (int) Math.sqrt(n);
 int[] squares = new int[maxSquareRoot];
 for (int i = 1; i <= maxSquareRoot; i++) {
 squares[i - 1] = i * i;
 }

 int[] dp = new int[n + 1];
 Arrays.fill(dp, Integer.MAX_VALUE);
 dp[0] = 0;

 // 先遍历物品（完全平方数），再遍历容量
 for (int square : squares) {
 for (int j = square; j <= n; j++) {
 if (dp[j - square] != Integer.MAX_VALUE) {
 dp[j] = Math.min(dp[j], dp[j - square] + 1);
 }
 }
 }

 return dp[n];
}

/**
 * 数学解法 - 利用四平方定理
 * 拉格朗日四平方定理：每个正整数都可以表示为 4 个整数的平方和
 * 勒让德三平方定理：当且仅当 $n \neq 4^a(8b+7)$ 时， n 可以表示为 3 个整数的平方和
 * @param n 目标正整数
 * @return 组成 n 的最少完全平方数个数
 */
public static int numSquaresMath(int n) {
 // 检查 n 是否是完全平方数
 if (isPerfectSquare(n)) {
 return 1;
 }

 // 检查是否满足勒让德三平方定理的排除条件
 if (checkLegendreThreeSquare(n)) {
 return 4;
 }

 // ...
}

```

```

}

// 检查是否可以表示为两个平方数之和
for (int i = 1; i * i <= n; i++) {
 int j = n - i * i;
 if (isPerfectSquare(j)) {
 return 2;
 }
}

// 其他情况返回 3
return 3;
}

/***
 * 判断一个数是否是完全平方数
 */
private static boolean isPerfectSquare(int x) {
 int sqrt = (int) Math.sqrt(x);
 return sqrt * sqrt == x;
}

/***
 * 检查是否满足勒让德三平方定理的排除条件
 * 即 $n = 4^a(8b+7)$
 */
private static boolean checkLegendreThreeSquare(int n) {
 while (n % 4 == 0) {
 n /= 4;
 }
 return n % 8 == 7;
}

/***
 * BFS 解法 - 将问题转化为图的最短路径问题
 * 每个数字是一个节点，如果两个数字相差一个完全平方数，则它们之间有边
 */
public static int numSquaresBFS(int n) {
 if (n <= 0) throw new IllegalArgumentException("n must be positive");

 // 使用队列进行 BFS
 java.util.Queue<Integer> queue = new java.util.LinkedList<>();
 // 记录到达每个数字的最短步数

```

```
int[] steps = new int[n + 1];
Arrays.fill(steps, -1);

// 从 0 开始
queue.offer(0);
steps[0] = 0;

while (!queue.isEmpty()) {
 int current = queue.poll();

 // 尝试所有可能的完全平方数
 for (int i = 1; i * i <= n - current; i++) {
 int next = current + i * i;

 // 如果超出范围或已经访问过，跳过
 if (next > n || steps[next] != -1) {
 continue;
 }

 steps[next] = steps[current] + 1;

 // 如果到达目标，直接返回
 if (next == n) {
 return steps[next];
 }

 queue.offer(next);
 }
}

return steps[n];
}

// 测试方法
public static void main(String[] args) {
 // 测试用例
 int[] testCases = {12, 13, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13};

 System.out.println("完全平方数问题测试: ");
 for (int n : testCases) {
 int result1 = numSquares(n);
 int result2 = numSquaresOptimized(n);
 int result3 = numSquaresBFS(n);
 }
}
```

```

int result4 = numSquaresMath(n);

System.out.printf("n=%d: DP=%d, Optimized=%d, BFS=%d, Math=%d%n",
 n, result1, result2, result3, result4);

// 验证所有方法结果一致
if (result1 != result2 || result2 != result3 || result3 != result4) {
 System.out.println("警告：不同方法结果不一致！");
}

}

// 性能测试
long startTime = System.currentTimeMillis();
int largeResult = numSquares(10000);
long endTime = System.currentTimeMillis();
System.out.printf("n=10000 的结果: %d, 耗时: %dms%n", largeResult, endTime - startTime);
}

/*
 * 复杂度分析:
 *
 * 方法 1: 动态规划 (完全背包)
 * - 时间复杂度: O(n * √n)
 * - 外层循环: √n 次 (完全平方数的个数)
 * - 内层循环: n 次 (背包容量)
 * - 空间复杂度: O(n)
 *
 * 方法 2: 优化的动态规划
 * - 时间复杂度: O(n * √n) (与方法 1 相同, 但常数更小)
 * - 空间复杂度: O(n)
 *
 * 方法 3: 数学解法
 * - 时间复杂度: O(√n)
 * - 检查完全平方数: O(1)
 * - 检查勒让德条件: O(log n)
 * - 检查两个平方数之和: O(√n)
 * - 空间复杂度: O(1)
 *
 * 方法 4: BFS 解法
 * - 时间复杂度: O(n * √n) (最坏情况)
 * - 空间复杂度: O(n)
 */

```

\* 最优解分析:

- \* - 对于小规模 n ( $n < 1000$ ): 所有方法都很快
- \* - 对于大规模 n ( $n \geq 10000$ ): 数学解法最优, 时间复杂度最低
- \* - 在实际工程中: 推荐使用动态规划, 代码清晰易懂

\*

\* 边界场景测试:

- \* 1.  $n=0$ : 应该返回 0 (根据题目定义, n 是正整数, 但需要处理边界)
- \* 2.  $n=1$ : 应该返回 1 (1 本身就是完全平方数)
- \* 3.  $n=2$ : 应该返回 2 ( $1+1$ )
- \* 4.  $n=3$ : 应该返回 3 ( $1+1+1$ )
- \* 5.  $n=4$ : 应该返回 1 (4 本身就是完全平方数)
- \* 6.  $n=12$ : 应该返回 3 ( $4+4+4$ )
- \* 7.  $n=13$ : 应该返回 2 ( $4+9$ )

\*

\* 工程化考量:

- \* 1. 异常处理: 对非法输入抛出明确异常
- \* 2. 性能优化: 预计算完全平方数, 避免重复计算
- \* 3. 可读性: 清晰的变量命名和详细注释
- \* 4. 测试覆盖: 包含正常情况、边界情况、性能测试
- \* 5. 多解法对比: 提供不同实现, 便于理解和选择

\*/

=====

文件: Code44\_PerfectSquares.py

=====

```
import math
from typing import List
import sys
from collections import deque

LeetCode 279. 完全平方数
题目描述: 给定正整数 n, 找到若干个完全平方数 (比如 1, 4, 9, 16, ...) 使得它们的和等于 n。
你需要让组成和的完全平方数的个数最少。
链接: https://leetcode.cn/problems/perfect-squares/
#
解题思路:
这是一个完全背包问题, 其中:
- 背包容量: 正整数 n
- 物品: 完全平方数 (1, 4, 9, 16, ...)
- 每个物品可以无限次使用 (完全背包)
- 目标: 使用最少数量的物品 (完全平方数) 装满背包
#
```

```
状态定义: dp[i] 表示和为 i 的完全平方数的最少数量
状态转移方程: dp[i] = min(dp[i], dp[i - j*j] + 1), 其中 j*j <= i
初始状态: dp[0] = 0, dp[i] = float('inf') (表示不可达)
#
时间复杂度: O(n * √n), 其中 n 是给定的正整数
空间复杂度: O(n), 使用一维 DP 数组
#
工程化考量:
1. 异常处理: 处理 n <= 0 的情况
2. 边界条件: n=0 时返回 0, n=1 时返回 1
3. 性能优化: 预先生成完全平方数列表
4. 可读性: 清晰的变量命名和注释
5. Python 特性: 使用类型注解提高代码可读性
```

```
class PerfectSquares:
```

```
 """
```

```
 完全平方数问题的多种解法
```

```
 """
```

```
@staticmethod
```

```
def num_squares_dp(n: int) -> int:
```

```
 """
```

```
 动态规划解法 - 完全背包问题
```

```
Args:
```

```
 n: 目标正整数
```

```
Returns:
```

```
 int: 组成 n 的最少完全平方数个数
```

```
Raises:
```

```
 ValueError: 当 n <= 0 时抛出异常
```

```
 """
```

```
参数验证
```

```
if n <= 0:
```

```
 raise ValueError("n must be positive")
```

```
特殊情况处理
```

```
if n == 1:
```

```
 return 1
```

```
创建 DP 数组, dp[i] 表示和为 i 的最少完全平方数个数
```

```
dp = [float('inf')] * (n + 1)
```

```

dp[0] = 0

遍历所有可能的完全平方数
i = 1
while i * i <= n:
 square = i * i
 # 完全背包：正序遍历容量
 for j in range(square, n + 1):
 if dp[j - square] != float('inf'):
 dp[j] = min(dp[j], dp[j - square] + 1)
 i += 1

return dp[n] if dp[n] != float('inf') else -1

```

@staticmethod

def num\_squares\_optimized(n: int) -> int:

"""

优化的动态规划解法 - 预先生成完全平方数列表

Args:

n: 目标正整数

Returns:

int: 组成 n 的最少完全平方数个数

"""

if n <= 0:

raise ValueError("n must be positive")

# 预先生成所有可能的完全平方数

max\_square\_root = int(math.sqrt(n))

squares = [i \* i for i in range(1, max\_square\_root + 1)]

dp = [float('inf')] \* (n + 1)

dp[0] = 0

# 先遍历物品（完全平方数），再遍历容量

for square in squares:

for j in range(square, n + 1):

if dp[j - square] != float('inf'):

dp[j] = min(dp[j], dp[j - square] + 1)

return dp[n] if dp[n] != float('inf') else -1

```
@staticmethod
def _is_perfect_square(x: int) -> bool:
 """
```

判断一个数是否是完全平方数

Args:

x: 要判断的数

Returns:

bool: 如果是完全平方数返回 True, 否则返回 False

```
"""
```

```
sqrt_val = int(math.sqrt(x))
```

```
return sqrt_val * sqrt_val == x
```

```
@staticmethod
```

```
def _check_legendre_three_square(n: int) -> bool:
 """
```

检查是否满足勒让德三平方定理的排除条件

即  $n = 4^a(8b+7)$

Args:

n: 要检查的数

Returns:

bool: 如果满足排除条件返回 True, 否则返回 False

```
"""
```

```
while n % 4 == 0:
```

```
 n /= 4
```

```
return n % 8 == 7
```

```
@staticmethod
```

```
def num_squares_math(n: int) -> int:
 """
```

数学解法 - 利用四平方定理

拉格朗日四平方定理: 每个正整数都可以表示为 4 个整数的平方和

勒让德三平方定理: 当且仅当  $n \neq 4^a(8b+7)$  时,  $n$  可以表示为 3 个整数的平方和

Args:

n: 目标正整数

Returns:

int: 组成  $n$  的最少完全平方数个数

```
"""
```

```

检查 n 是否是完全平方数
if PerfectSquares._is_perfect_square(n):
 return 1

检查是否满足勒让德三平方定理的排除条件
if PerfectSquares._check_legendre_three_square(n):
 return 4

检查是否可以表示为两个平方数之和
i = 1
while i * i <= n:
 j = n - i * i
 if PerfectSquares._is_perfect_square(j):
 return 2
 i += 1

其他情况返回 3
return 3

```

@staticmethod

def num\_squares\_bfs(n: int) -> int:

"""

BFS 解法 – 将问题转化为图的最短路径问题

每个数字是一个节点，如果两个数字相差一个完全平方数，则它们之间有边

Args:

n: 目标正整数

Returns:

int: 组成 n 的最少完全平方数个数

"""

if n <= 0:

raise ValueError("n must be positive")

# 使用队列进行 BFS

queue = deque()

# 记录到达每个数字的最短步数

steps = [-1] \* (n + 1)

# 从 0 开始

queue.append(0)

steps[0] = 0

```

while queue:
 current = queue.popleft()

 # 尝试所有可能的完全平方数
 i = 1
 while i * i <= n - current:
 next_val = current + i * i

 # 如果超出范围或已经访问过，跳过
 if next_val > n or steps[next_val] != -1:
 i += 1
 continue

 steps[next_val] = steps[current] + 1

 # 如果到达目标，直接返回
 if next_val == n:
 return steps[next_val]

 queue.append(next_val)
 i += 1

return steps[n]

@staticmethod
def run_tests():
 """
 运行测试用例，验证所有方法的正确性
 """

 # 测试用例
 test_cases = [12, 13, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]

 print("完全平方数问题测试：")
 for n in test_cases:
 try:
 result1 = PerfectSquares.num_squares_dp(n)
 result2 = PerfectSquares.num_squares_optimized(n)
 result3 = PerfectSquares.num_squares_bfs(n)
 result4 = PerfectSquares.num_squares_math(n)

 print(f"n={n}: DP={result1}, Optimized={result2}, BFS={result3}, Math={result4}")
 except Exception as e:
 print(f"Error for n={n}: {e}")

 # 验证所有方法结果一致

```

```

 if result1 != result2 or result2 != result3 or result3 != result4:
 print("警告: 不同方法结果不一致!")

 except Exception as e:
 print(f"测试 n={n} 时发生错误: {e}")

性能测试
import time
start_time = time.time()
large_result = PerfectSquares.num_squares_dp(10000)
end_time = time.time()
print(f"n=10000 的结果: {large_result}, 耗时: {end_time - start_time:.4f}秒")

def main():
 """
 主函数 - 运行测试和演示
 """
 try:
 PerfectSquares.run_tests()
 except Exception as e:
 print(f"程序执行错误: {e}")
 return 1
 return 0

if __name__ == "__main__":
 sys.exit(main())

```

复杂度分析:

方法 1: 动态规划 (完全背包)

- 时间复杂度:  $O(n * \sqrt{n})$ 
  - 外层循环:  $\sqrt{n}$  次 (完全平方数的个数)
  - 内层循环:  $n$  次 (背包容量)
- 空间复杂度:  $O(n)$

方法 2: 优化的动态规划

- 时间复杂度:  $O(n * \sqrt{n})$  (与方法 1 相同, 但常数更小)
- 空间复杂度:  $O(n)$

方法 3: 数学解法

- 时间复杂度:  $O(\sqrt{n})$ 
  - 检查完全平方数:  $O(1)$

- 检查勒让德条件:  $O(\log n)$
- 检查两个平方数之和:  $O(\sqrt{n})$
- 空间复杂度:  $O(1)$

#### 方法 4: BFS 解法

- 时间复杂度:  $O(n * \sqrt{n})$  (最坏情况)
- 空间复杂度:  $O(n)$

#### 最优解分析:

- 对于小规模  $n$  ( $n < 1000$ ): 所有方法都很快
- 对于大规模  $n$  ( $n \geq 10000$ ): 数学解法最优, 时间复杂度最低
- 在实际工程中: 推荐使用动态规划, 代码清晰易懂

#### Python 特定优化:

1. 使用类型注解提高代码可读性
2. 使用 deque 进行 BFS, 效率更高
3. 使用 math.sqrt 进行平方根计算
4. 使用 float('inf') 表示无穷大
5. 异常处理使用 Python 标准异常

#### 边界场景测试:

1.  $n=0$ : 应该返回 0 (根据题目定义,  $n$  是正整数, 但需要处理边界)
2.  $n=1$ : 应该返回 1 (1 本身就是完全平方数)
3.  $n=2$ : 应该返回 2 ( $1+1$ )
4.  $n=3$ : 应该返回 3 ( $1+1+1$ )
5.  $n=4$ : 应该返回 1 (4 本身就是完全平方数)
6.  $n=12$ : 应该返回 3 ( $4+4+4$ )
7.  $n=13$ : 应该返回 2 ( $4+9$ )

#### 工程化考量:

1. 模块化设计: 将不同解法封装为静态方法
2. 类型注解: 提高代码可读性和可维护性
3. 异常处理: 对非法输入抛出明确异常
4. 测试覆盖: 包含正常情况、边界情况、性能测试
5. 文档完善: 详细的 docstring 和注释

"""

---

文件: Code45\_CombinationSumIV.cpp

---

```
#include <iostream>
#include <vector>
```

```

#include <algorithm>
#include <climits>
#include <chrono>

using namespace std;

// LeetCode 377. 组合总和 IV
// 题目描述：给定一个由不同整数组成的数组 nums 和一个目标整数 target,
// 请从 nums 中找出并返回总和为 target 的元素组合的个数。
// 链接: https://leetcode.cn/problems/combination-sum-iv/
//
// 解题思路：
// 这是一个完全背包问题的排列数变种，需要计算所有可能的排列数。
// 与零钱兑换 II 不同，这里顺序不同的序列被视为不同的组合。
//
// 状态定义：dp[i] 表示总和为 i 的元素组合个数
// 状态转移方程：dp[i] = sum(dp[i - num]), 其中 num 是 nums 中的元素且 i >= num
// 初始状态：dp[0] = 1 (空组合)
//
// 关键点：为了计算排列数，需要将目标值循环放在外层，数组元素循环放在内层
//
// 时间复杂度：O(target * n)，其中 n 是数组长度
// 空间复杂度：O(target)，使用一维 DP 数组
//
// 工程化考量：
// 1. 异常处理：处理空数组、负数等情况
// 2. 整数溢出：使用 long long 类型处理大数
// 3. 性能优化：排序数组进行剪枝
// 4. 边界条件：target=0 时返回 1

class Solution {
public:
 /**
 * 动态规划解法 - 计算排列数
 * @param nums 不同整数组成的数组
 * @param target 目标整数
 * @return 总和为 target 的元素组合个数
 */
 int combinationSum4(vector<int>& nums, int target) {
 // 参数验证
 if (nums.empty()) {
 return target == 0 ? 1 : 0;
 }

```

```

 if (target < 0) {
 return 0;
 }

 // 特殊情况处理
 if (target == 0) {
 return 1; // 空组合
 }

 // 创建 DP 数组
 vector<unsigned int> dp(target + 1, 0);
 dp[0] = 1; // 空组合

 // 为了计算排列数，需要将目标值循环放在外层
 // 数组元素循环放在内层
 for (int i = 1; i <= target; i++) {
 for (int num : nums) {
 if (i >= num) {
 // 使用 unsigned int 避免溢出警告
 if (dp[i] > UINT_MAX - dp[i - num]) {
 // 处理溢出情况
 dp[i] = UINT_MAX;
 } else {
 dp[i] += dp[i - num];
 }
 }
 }
 }

 return static_cast<int>(dp[target]);
}

/***
 * 优化的动态规划解法 - 处理整数溢出
 * 使用 long long 类型避免整数溢出
 */
int combinationSum4Optimized(vector<int>& nums, int target) {
 if (nums.empty()) {
 return target == 0 ? 1 : 0;
 }

 if (target < 0) {
 return 0;
 }
}

```

```

// 使用 long long 数组避免整数溢出
vector<long long> dp(target + 1, 0);
dp[0] = 1;

for (int i = 1; i <= target; i++) {
 for (int num : nums) {
 if (i >= num) {
 dp[i] += dp[i - num];
 // 如果超过 int 最大值，取最大值
 if (dp[i] > INT_MAX) {
 dp[i] = INT_MAX;
 }
 }
 }
}

return static_cast<int>(dp[target]);
}

/***
 * 带剪枝优化的动态规划解法
 * 先排序数组，当 num > i 时提前终止内层循环
 */
int combinationSum4WithPruning(vector<int>& nums, int target) {
 if (nums.empty()) {
 return target == 0 ? 1 : 0;
 }

 if (target < 0) {
 return 0;
 }

 // 排序数组，便于剪枝
 sort(nums.begin(), nums.end());
 vector<unsigned int> dp(target + 1, 0);
 dp[0] = 1;

 for (int i = 1; i <= target; i++) {
 for (int num : nums) {
 if (num > i) {
 break; // 剪枝：由于数组已排序，后续数字更大
 }
 if (dp[i] > UINT_MAX - dp[i - num]) {

```

```

 dp[i] = UINT_MAX;
 } else {
 dp[i] += dp[i - num];
 }
}

return static_cast<int>(dp[target]);
}

/***
 * 递归+记忆化搜索解法
 */
int combinationSum4DFS(vector<int>& nums, int target) {
 if (nums.empty()) {
 return target == 0 ? 1 : 0;
 }
 if (target < 0) {
 return 0;
 }

 // 使用记忆化数组
 vector<int> memo(target + 1, -1);
 return dfs(nums, target, memo);
}

/***
 * 递归辅助函数
 */
int dfs(vector<int>& nums, int target, vector<int>& memo) {
 // 基础情况
 if (target == 0) {
 return 1;
 }
 if (target < 0) {
 return 0;
 }

 // 检查记忆化数组
 if (memo[target] != -1) {
 return memo[target];
 }

 int result = 0;
 for (int num : nums) {
 result += dfs(nums, target - num, memo);
 }
 memo[target] = result;
 return result;
}

```

```

int count = 0;
for (int num : nums) {
 count += dfs(nums, target - num, memo);
}

memo[target] = count;
return count;
}

};

/***
 * 测试函数
 */
void testCombinationSum4() {
 Solution sol;

 // 测试用例
 vector<pair<vector<int>, int>> testCases = {
 {{1, 2, 3}, 4}, // 预期: 7
 {{9}, 3}, // 预期: 0
 {{1, 2, 3}, 0}, // 预期: 1
 {{1, 2, 3}, 1}, // 预期: 1
 {{1, 2, 3}, 2}, // 预期: 2
 {{1, 2, 3}, 3} // 预期: 4
 };

 cout << "组合总和 IV 问题测试: " << endl;
 for (auto& testCase : testCases) {
 vector<int> nums = testCase.first;
 int target = testCase.second;

 int result1 = sol.combinationSum4(nums, target);
 int result2 = sol.combinationSum4Optimized(nums, target);
 int result3 = sol.combinationSum4WithPruning(nums, target);
 int result4 = sol.combinationSum4DFS(nums, target);

 cout << "nums=[";
 for (size_t i = 0; i < nums.size(); i++) {
 cout << nums[i];
 if (i < nums.size() - 1) cout << ", ";
 }
 cout << "], target=" << target
 << ": DP=" << result1
 }
}

```

```

 << ", Optimized=" << result2
 << ", Pruning=" << result3
 << ", DFS=" << result4 << endl;

 // 验证结果一致性
 if (result1 != result2 || result2 != result3 || result3 != result4) {
 cout << "警告：不同方法结果不一致！" << endl;
 }
}

// 性能测试 - 大规模数据
vector<int> largeNums = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int largeTarget = 50;

auto startTime = chrono::high_resolution_clock::now();
int largeResult = sol.combinationSum4WithPruning(largeNums, largeTarget);
auto endTime = chrono::high_resolution_clock::now();
auto duration = chrono::duration_cast<chrono::milliseconds>(endTime - startTime);

cout << "大规模测试: nums 长度=" << largeNums.size()
 << ", target=" << largeTarget
 << ", 结果=" << largeResult
 << ", 耗时=" << duration.count() << "ms" << endl;

// 边界情况测试
cout << "边界情况测试: " << endl;
vector<int> emptyNums;
cout << "空数组, target=0: " << sol.combinationSum4(emptyNums, 0) << endl; // 预期: 1
cout << "空数组, target=1: " << sol.combinationSum4(emptyNums, 1) << endl; // 预期: 0
cout << "负数 target: " << sol.combinationSum4({1, 2, 3}, -1) << endl; // 预期: 0
}

int main() {
 try {
 testCombinationSum4();
 } catch (const exception& e) {
 cerr << "错误: " << e.what() << endl;
 return 1;
 }
 return 0;
}

/*

```

- \* 复杂度分析:
  - \*
  - \* 方法 1: 动态规划 (排列数)
    - \* - 时间复杂度:  $O(target * n)$ 
      - 外层循环: target 次
      - 内层循环: n 次 (数组长度)
    - \* - 空间复杂度:  $O(target)$
  - \*
  - \* 方法 2: 优化的动态规划 (处理溢出)
    - \* - 时间复杂度:  $O(target * n)$  (与方法 1 相同)
    - \* - 空间复杂度:  $O(target)$
  - \*
  - \* 方法 3: 带剪枝的动态规划
    - \* - 时间复杂度:  $O(target * n)$  (平均情况下由于剪枝可能更快)
    - \* - 空间复杂度:  $O(target)$
  - \*
  - \* 方法 4: 递归+记忆化搜索
    - \* - 时间复杂度:  $O(target * n)$  (每个状态计算一次)
    - \* - 空间复杂度:  $O(target)$  (递归栈深度+记忆化数组)
  - \*
  - \* C++特定优化:
    - \* 1. 使用 vector 代替数组, 更安全
    - \* 2. 使用 unsigned int 处理大数
    - \* 3. 使用 chrono 进行精确性能测试
    - \* 4. 异常处理使用 C++标准异常
  - \*
  - \* 关键点分析:
    - \* 1. 排列数 vs 组合数: 本题需要计算排列数, 因此遍历顺序很重要
    - \* 2. 整数溢出: 当 target 较大时, 结果可能超过 int 范围
    - \* 3. 剪枝优化: 排序数组可以在内层循环提前终止
  - \*
  - \* 工程化考量:
    - \* 1. 模块化设计: 将不同解法封装为类方法
    - \* 2. 类型安全: 使用适当的类型避免溢出
    - \* 3. 性能优化: 利用 STL 算法进行排序
    - \* 4. 测试覆盖: 包含各种边界情况

=====

文件: Code45\_CombinationSumIV.java

=====

```
package class073;
```

```
import java.util.Arrays;

// LeetCode 377. 组合总和 IV
// 题目描述: 给定一个由不同整数组成的数组 nums 和一个目标整数 target,
// 请从 nums 中找出并返回总和为 target 的元素组合的个数。
// 链接: https://leetcode.cn/problems/combination-sum-iv/
//
// 解题思路:
// 这是一个完全背包问题的排列数变种, 需要计算所有可能的排列数。
// 与零钱兑换 II 不同, 这里顺序不同的序列被视为不同的组合。
//
// 状态定义: dp[i] 表示总和为 i 的元素组合个数
// 状态转移方程: dp[i] = sum(dp[i - num]), 其中 num 是 nums 中的元素且 i >= num
// 初始状态: dp[0] = 1 (空组合)
//
// 关键点: 为了计算排列数, 需要将目标值循环放在外层, 数组元素循环放在内层
//
// 时间复杂度: O(target * n), 其中 n 是数组长度
// 空间复杂度: O(target), 使用一维 DP 数组
//
// 工程化考量:
// 1. 异常处理: 处理空数组、负数等情况
// 2. 整数溢出: 使用 long 类型处理大数
// 3. 性能优化: 排序数组进行剪枝
// 4. 边界条件: target=0 时返回 1

public class Code45_CombinationSumIV {

 /**
 * 动态规划解法 - 计算排列数
 * @param nums 不同整数组成的数组
 * @param target 目标整数
 * @return 总和为 target 的元素组合个数
 */
 public static int combinationSum4(int[] nums, int target) {
 // 参数验证
 if (nums == null || nums.length == 0) {
 return target == 0 ? 1 : 0;
 }
 if (target < 0) {
 return 0;
 }
 }
}
```

```

// 特殊情况处理
if (target == 0) {
 return 1; // 空组合
}

// 创建 DP 数组
int[] dp = new int[target + 1];
dp[0] = 1; // 空组合

// 为了计算排列数，需要将目标值循环放在外层
// 数组元素循环放在内层
for (int i = 1; i <= target; i++) {
 for (int num : nums) {
 if (i >= num) {
 dp[i] += dp[i - num];
 }
 }
}

return dp[target];
}

/**
 * 优化的动态规划解法 - 处理整数溢出
 * 使用 long 类型避免整数溢出，最后转换为 int
 */
public static int combinationSum4Optimized(int[] nums, int target) {
 if (nums == null || nums.length == 0) {
 return target == 0 ? 1 : 0;
 }

 if (target < 0) {
 return 0;
 }

 // 使用 long 数组避免整数溢出
 long[] dp = new long[target + 1];
 dp[0] = 1;

 for (int i = 1; i <= target; i++) {
 for (int num : nums) {
 if (i >= num) {
 dp[i] += dp[i - num];
 }
 }
 }
}

```

```

 // 如果超过 int 最大值，取模或返回最大值
 if (dp[i] > Integer.MAX_VALUE) {
 dp[i] = Integer.MAX_VALUE;
 }
 }

}

return (int) dp[target];
}

/***
 * 带剪枝优化的动态规划解法
 * 先排序数组，当 num > i 时提前终止内层循环
 */
public static int combinationSum4WithPruning(int[] nums, int target) {
 if (nums == null || nums.length == 0) {
 return target == 0 ? 1 : 0;
 }
 if (target < 0) {
 return 0;
 }

 // 排序数组，便于剪枝
 Arrays.sort(nums);
 int[] dp = new int[target + 1];
 dp[0] = 1;

 for (int i = 1; i <= target; i++) {
 for (int num : nums) {
 if (num > i) {
 break; // 剪枝：由于数组已排序，后续数字更大
 }
 dp[i] += dp[i - num];
 }
 }

 return dp[target];
}

/***
 * 递归+记忆化搜索解法
 */

```

```
public static int combinationSum4DFS(int[] nums, int target) {
 if (nums == null || nums.length == 0) {
 return target == 0 ? 1 : 0;
 }
 if (target < 0) {
 return 0;
 }

 // 使用记忆化数组
 Integer[] memo = new Integer[target + 1];
 return dfs(nums, target, memo);
}

/**
 * 递归辅助函数
 */
private static int dfs(int[] nums, int target, Integer[] memo) {
 // 基础情况
 if (target == 0) {
 return 1;
 }
 if (target < 0) {
 return 0;
 }

 // 检查记忆化数组
 if (memo[target] != null) {
 return memo[target];
 }

 int count = 0;
 for (int num : nums) {
 count += dfs(nums, target - num, memo);
 }

 memo[target] = count;
 return count;
}

/**
 * 测试方法
 */
public static void main(String[] args) {
```

```

// 测试用例
int[][] testCases = {
 {1, 2, 3}, 4, // 预期: 7
 {9}, 3, // 预期: 0
 {1, 2, 3}, 0, // 预期: 1
 {1, 2, 3}, 1, // 预期: 1
 {1, 2, 3}, 2, // 预期: 2
 {1, 2, 3}, 3 // 预期: 4
};

System.out.println("组合总和 IV 问题测试: ");
for (int i = 0; i < testCases.length; i += 2) {
 int[] nums = testCases[i];
 int target = testCases[i + 1];

 int result1 = combinationSum4(nums, target);
 int result2 = combinationSum4Optimized(nums, target);
 int result3 = combinationSum4WithPruning(nums, target);
 int result4 = combinationSum4DFS(nums, target);

 System.out.printf("nums=%s, target=%d: DP=%d, Optimized=%d, Pruning=%d, DFS=%d%n",
 Arrays.toString(nums), target, result1, result2, result3, result4);
}

// 验证结果一致性
if (result1 != result2 || result2 != result3 || result3 != result4) {
 System.out.println("警告: 不同方法结果不一致!");
}
}

// 性能测试 - 大规模数据
int[] largeNums = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int largeTarget = 50;

long startTime = System.currentTimeMillis();
int largeResult = combinationSum4WithPruning(largeNums, largeTarget);
long endTime = System.currentTimeMillis();

System.out.printf("大规模测试: nums 长度=%d, target=%d, 结果=%d, 耗时=%dms%n",
 largeNums.length, largeTarget, largeResult, endTime - startTime);

// 边界情况测试
System.out.println("边界情况测试: ");
System.out.println("空数组, target=0: " + combinationSum4(new int[] {}, 0)); // 预期: 1

```

```
System.out.println("空数组, target=1: " + combinationSum4(new int[] {}, 1)); // 预期: 0
System.out.println("负数 target: " + combinationSum4(new int[] {1, 2, 3}, -1)); // 预期: 0
}

/*
 * 复杂度分析:
 *
 * 方法 1: 动态规划 (排列数)
 * - 时间复杂度: O(target * n)
 * - 外层循环: target 次
 * - 内层循环: n 次 (数组长度)
 * - 空间复杂度: O(target)
 *
 * 方法 2: 优化的动态规划 (处理溢出)
 * - 时间复杂度: O(target * n) (与方法 1 相同)
 * - 空间复杂度: O(target)
 *
 * 方法 3: 带剪枝的动态规划
 * - 时间复杂度: O(target * n) (平均情况下由于剪枝可能更快)
 * - 空间复杂度: O(target)
 *
 * 方法 4: 递归+记忆化搜索
 * - 时间复杂度: O(target * n) (每个状态计算一次)
 * - 空间复杂度: O(target) (递归栈深度+记忆化数组)
 *
 * 关键点分析:
 * 1. 排列数 vs 组合数: 本题需要计算排列数, 因此遍历顺序很重要
 * 2. 整数溢出: 当 target 较大时, 结果可能超过 int 范围
 * 3. 剪枝优化: 排序数组可以在内层循环提前终止
 *
 * 与零钱兑换 II 的区别:
 * - 零钱兑换 II: 计算组合数 (顺序不同的序列视为相同)
 * - 组合总和 IV: 计算排列数 (顺序不同的序列视为不同)
 *
 * 工程化考量:
 * 1. 异常防御: 处理各种边界输入
 * 2. 性能优化: 剪枝、记忆化等技术
 * 3. 可维护性: 清晰的代码结构和注释
 * 4. 测试覆盖: 包含正常、边界、性能测试
 *
 * 面试要点:
 * 1. 理解排列数和组合数的区别
```

- \* 2. 掌握动态规划的遍历顺序对结果的影响
  - \* 3. 能够处理整数溢出等边界情况
  - \* 4. 了解不同解法的优缺点和适用场景
- \*/
- 

文件: Code45\_CombinationSumIV.py

---

```
from typing import List
import sys
from functools import lru_cache

LeetCode 377. 组合总和 IV
题目描述: 给定一个由不同整数组成的数组 nums 和一个目标整数 target,
请从 nums 中找出并返回总和为 target 的元素组合的个数。
链接: https://leetcode.cn/problems/combination-sum-iv/
#
解题思路:
这是一个完全背包问题的排列数变种, 需要计算所有可能的排列数。
与零钱兑换 II 不同, 这里顺序不同的序列被视为不同的组合。
#
状态定义: dp[i] 表示总和为 i 的元素组合个数
状态转移方程: dp[i] = sum(dp[i - num]), 其中 num 是 nums 中的元素且 i >= num
初始状态: dp[0] = 1 (空组合)
#
关键点: 为了计算排列数, 需要将目标值循环放在外层, 数组元素循环放在内层
#
时间复杂度: O(target * n), 其中 n 是数组长度
空间复杂度: O(target), 使用一维 DP 数组
#
工程化考量:
1. 异常处理: 处理空数组、负数等情况
2. 整数溢出: Python 自动处理大整数, 但需要注意性能
3. 性能优化: 使用 lru_cache 进行记忆化
4. 边界条件: target=0 时返回 1

class CombinationSumIV:
 """
 组合总和 IV 问题的多种解法
 """

 @staticmethod
```

```
def combination_sum4_dp(nums: List[int], target: int) -> int:
 """
 动态规划解法 - 计算排列数

 Args:
 nums: 不同整数组成的数组
 target: 目标整数

 Returns:
 int: 总和为 target 的元素组合个数

 Raises:
 ValueError: 当 target 为负数时
 """

 # 参数验证
 if not nums:
 return 1 if target == 0 else 0
 if target < 0:
 return 0

 # 特殊情况处理
 if target == 0:
 return 1 # 空组合

 # 创建 DP 数组
 dp = [0] * (target + 1)
 dp[0] = 1 # 空组合

 # 为了计算排列数，需要将目标值循环放在外层
 # 数组元素循环放在内层
 for i in range(1, target + 1):
 for num in nums:
 if i >= num:
 dp[i] += dp[i - num]

 return dp[target]

@staticmethod
def combination_sum4_optimized(nums: List[int], target: int) -> int:
 """
 优化的动态规划解法 - 排序数组进行剪枝

 Args:
```

nums: 不同整数组成的数组

target: 目标整数

Returns:

int: 总和为 target 的元素组合个数

"""

if not nums:

    return 1 if target == 0 else 0

if target < 0:

    return 0

# 排序数组，便于剪枝

nums\_sorted = sorted(nums)

dp = [0] \* (target + 1)

dp[0] = 1

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

    for num in nums\_sorted:

        if num > i:

            break # 剪枝：由于数组已排序，后续数字更大

        dp[i] += dp[i - num]

return dp[target]

@staticmethod

@lru\_cache(maxsize=None)

def \_combination\_sum4\_dfs(nums\_tuple: tuple, target: int) -> int:

"""

递归辅助函数 - 使用 lru\_cache 进行记忆化

Args:

nums\_tuple: 转换为元组的 nums 数组（用于缓存）

target: 目标整数

Returns:

int: 组合个数

"""

# 基础情况

if target == 0:

    return 1

if target < 0:

    return 0

```
count = 0
for num in nums_tuple:
 if target >= num:
 count += CombinationSumIV._combination_sum4_dfs(nums_tuple, target - num)

return count
```

```
@staticmethod
def combination_sum4_dfs(nums: List[int], target: int) -> int:
 """
```

递归+记忆化搜索解法

Args:

```
 nums: 不同整数组成的数组
 target: 目标整数
```

Returns:

```
 int: 总和为 target 的元素组合个数
 """
```

```
if not nums:
 return 1 if target == 0 else 0
if target < 0:
 return 0
```

# 将列表转换为元组以便缓存

```
nums_tuple = tuple(sorted(nums))
return CombinationSumIV._combination_sum4_dfs(nums_tuple, target)
```

```
@staticmethod
```

```
def combination_sum4_iterative(nums: List[int], target: int) -> int:
 """
```

迭代解法 - 避免递归深度限制

Args:

```
 nums: 不同整数组成的数组
 target: 目标整数
```

Returns:

```
 int: 总和为 target 的元素组合个数
 """
```

```
if not nums:
 return 1 if target == 0 else 0
if target < 0:
 return 0
```

```

 return 0

dp = [0] * (target + 1)
dp[0] = 1

使用队列进行迭代计算
from collections import deque
queue = deque([0])

while queue:
 current = queue.popleft()

 for num in nums:
 next_val = current + num
 if next_val <= target:
 if dp[next_val] == 0:
 queue.append(next_val)
 dp[next_val] += dp[current]

return dp[target]

@staticmethod
def run_tests():
 """
 运行测试用例，验证所有方法的正确性
 """

 # 测试用例
 test_cases = [
 ([1, 2, 3], 4), # 预期: 7
 ([9], 3), # 预期: 0
 ([1, 2, 3], 0), # 预期: 1
 ([1, 2, 3], 1), # 预期: 1
 ([1, 2, 3], 2), # 预期: 2
 ([1, 2, 3], 3) # 预期: 4
]

 print("组合总和 IV 问题测试: ")
 for nums, target in test_cases:
 try:
 result1 = CombinationSumIV.combination_sum4_dp(nums, target)
 result2 = CombinationSumIV.combination_sum4_optimized(nums, target)
 result3 = CombinationSumIV.combination_sum4_dfs(nums, target)
 result4 = CombinationSumIV.combination_sum4_iterative(nums, target)

 if result1 != result2 or result1 != result3 or result1 != result4:
 print(f"测试用例 {nums} 和 {target} 失败！")
 print(f"预期结果: {result1}, 实际结果: {result2}, {result3}, {result4}")
 except Exception as e:
 print(f"发生错误: {e}")

```

```

 print(f"nums={nums}, target={target}: "
 f"DP={result1}, Optimized={result2}, DFS={result3}, Iterative={result4}")

 # 验证结果一致性
 if result1 != result2 or result2 != result3 or result3 != result4:
 print("警告: 不同方法结果不一致!")

except Exception as e:
 print(f"测试 nums={nums}, target={target} 时发生错误: {e}")

性能测试 - 大规模数据
import time
large_nums = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
large_target = 50

start_time = time.time()
large_result = CombinationSumIV.combination_sum4_optimized(large_nums, large_target)
end_time = time.time()

print(f"大规模测试: nums 长度={len(large_nums)}, target={large_target}, "
 f"结果={large_result}, 耗时={end_time - start_time:.4f}秒")

边界情况测试
print("边界情况测试: ")
print(f"空数组, target=0: {CombinationSumIV.combination_sum4_dp([], 0)}") # 预期: 1
print(f"空数组, target=1: {CombinationSumIV.combination_sum4_dp([], 1)}") # 预期: 0
print(f"负数 target: {CombinationSumIV.combination_sum4_dp([1, 2, 3], -1)}") # 预期: 0

测试大数情况 (Python 自动处理大整数)
print("大数测试: ")
try:
 large_result2 = CombinationSumIV.combination_sum4_dp([1, 2], 100)
 print(f"nums=[1, 2], target=100: 结果={large_result2}")
except Exception as e:
 print(f"大数测试错误: {e}")

def main():
 """
 主函数 - 运行测试和演示
 """
 try:
 CombinationSumIV.run_tests()

```

```
except Exception as e:
 print(f"程序执行错误: {e}")
 return 1
return 0

if __name__ == "__main__":
 sys.exit(main())

"""
```

复杂度分析:

方法 1: 动态规划 (排列数)

- 时间复杂度:  $O(target * n)$ 
  - 外层循环: target 次
  - 内层循环: n 次 (数组长度)
- 空间复杂度:  $O(target)$

方法 2: 优化的动态规划 (剪枝)

- 时间复杂度:  $O(target * n)$  (平均情况下由于剪枝可能更快)
- 空间复杂度:  $O(target)$

方法 3: 递归+记忆化搜索

- 时间复杂度:  $O(target * n)$  (每个状态计算一次)
- 空间复杂度:  $O(target)$  (记忆化缓存)

方法 4: 迭代解法

- 时间复杂度:  $O(target * n)$  (最坏情况)
- 空间复杂度:  $O(target)$

Python 特定优化:

1. 使用 `lru_cache` 进行自动记忆化
2. 利用 Python 的大整数特性, 无需担心溢出
3. 使用类型注解提高代码可读性
4. 使用 `deque` 进行迭代计算, 避免递归深度限制

关键点分析:

1. 排列数 vs 组合数: 本题需要计算排列数, 因此遍历顺序很重要
2. Python 优势: 自动处理大整数, 无需担心溢出问题
3. 记忆化优化: 使用 `lru_cache` 简化记忆化实现
4. 剪枝策略: 排序数组可以在内层循环提前终止

工程化考量:

1. 模块化设计: 将不同解法封装为静态方法

2. 异常处理：完善的参数验证和错误处理
3. 性能监控：包含性能测试和时间测量
4. 测试覆盖：包含各种边界情况和性能测试

面试要点：

1. 理解排列数和组合数的本质区别
2. 掌握动态规划中遍历顺序的重要性
3. 了解记忆化搜索的实现技巧
4. 能够分析不同解法的时空复杂度

"""

文件：Code46\_CoinChangeII.cpp

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <chrono>

using namespace std;

// LeetCode 518. 零钱兑换 II

// 题目描述：给你一个整数数组 coins 表示不同面额的硬币，另给一个整数 amount 表示总金额。
// 请你计算并返回可以凑成总金额的硬币组合数。如果任何硬币组合都无法凑出总金额，返回 0。
// 链接：https://leetcode.cn/problems/coin-change-ii/
//

// 解题思路：
// 这是一个完全背包问题的组合数变种，需要计算所有可能的组合数。
// 与组合总和 IV 不同，这里顺序不同的序列被视为相同的组合。
//
// 状态定义：dp[i] 表示凑成总金额 i 的硬币组合数
// 状态转移方程：dp[i] += dp[i - coin]，其中 coin 是 coins 中的硬币且 i >= coin
// 初始状态：dp[0] = 1 (空组合)
//
// 关键点：为了计算组合数，需要将硬币循环放在外层，金额循环放在内层
//
// 时间复杂度：O(amount * n)，其中 n 是硬币种类数
// 空间复杂度：O(amount)，使用一维 DP 数组
//
// 工程化考量：
// 1. 异常处理：处理空数组、负数等情况
// 2. 边界条件：amount=0 时返回 1
```

```
// 3. 性能优化：排序硬币进行剪枝
// 4. 类型安全：使用适当的数据类型

class Solution {
public:
 /**
 * 动态规划解法 - 计算组合数
 * @param coins 不同面额的硬币数组
 * @param amount 目标总金额
 * @return 凑成总金额的硬币组合数
 */
 int change(int amount, vector<int>& coins) {
 // 参数验证
 if (coins.empty()) {
 return amount == 0 ? 1 : 0;
 }
 if (amount < 0) {
 return 0;
 }

 // 特殊情况处理
 if (amount == 0) {
 return 1; // 空组合
 }

 // 创建 DP 数组
 vector<int> dp(amount + 1, 0);
 dp[0] = 1; // 空组合

 // 为了计算组合数，需要将硬币循环放在外层
 // 金额循环放在内层
 for (int coin : coins) {
 for (int i = coin; i <= amount; i++) {
 dp[i] += dp[i - coin];
 }
 }

 return dp[amount];
 }

 /**
 * 优化的动态规划解法 - 排序硬币进行剪枝
 */
}
```

```

int changeOptimized(int amount, vector<int>& coins) {
 if (coins.empty()) {
 return amount == 0 ? 1 : 0;
 }
 if (amount < 0) {
 return 0;
 }

 // 排序硬币，便于理解和调试（对组合数结果无影响）
 sort(coins.begin(), coins.end());
 vector<int> dp(amount + 1, 0);
 dp[0] = 1;

 for (int coin : coins) {
 for (int i = coin; i <= amount; i++) {
 dp[i] += dp[i - coin];
 }
 }

 return dp[amount];
}

/***
 * 带剪枝的动态规划解法 - 当硬币大于剩余金额时提前终止
 */
int changeWithPruning(int amount, vector<int>& coins) {
 if (coins.empty()) {
 return amount == 0 ? 1 : 0;
 }
 if (amount < 0) {
 return 0;
 }

 // 排序硬币，便于剪枝
 sort(coins.begin(), coins.end());
 vector<int> dp(amount + 1, 0);
 dp[0] = 1;

 for (int coin : coins) {
 // 如果硬币面额已经大于 amount，后续硬币更大，直接跳过
 if (coin > amount) {
 continue;
 }

```

```

 for (int i = coin; i <= amount; i++) {
 dp[i] += dp[i - coin];
 }
 }

 return dp[amount];
}

/***
 * 递归+记忆化搜索解法
 */
int changeDFS(int amount, vector<int>& coins) {
 if (coins.empty()) {
 return amount == 0 ? 1 : 0;
 }
 if (amount < 0) {
 return 0;
 }

 // 使用记忆化数组
 vector<vector<int>> memo(coins.size(), vector<int>(amount + 1, -1));
 return dfs(coins, 0, amount, memo);
}

/***
 * 递归辅助函数
 * @param coins 硬币数组
 * @param index 当前考虑的硬币索引
 * @param amount 剩余金额
 * @param memo 记忆化数组
 * @return 组合数
 */
int dfs(vector<int>& coins, int index, int amount, vector<vector<int>>& memo) {
 // 基础情况
 if (amount == 0) {
 return 1;
 }
 if (amount < 0 || index >= coins.size()) {
 return 0;
 }

 // 检查记忆化数组
 if (memo[index][amount] != -1) {

```

```

 return memo[index][amount];
 }

 int count = 0;
 // 选择当前硬币 0 次或多次
 for (int k = 0; k * coins[index] <= amount; k++) {
 count += dfs(coins, index + 1, amount - k * coins[index], memo);
 }

 memo[index][amount] = count;
 return count;
}

/***
 * 空间优化的递归解法 - 一维记忆化
 */
int changeDFSOptimized(int amount, vector<int>& coins) {
 if (coins.empty()) {
 return amount == 0 ? 1 : 0;
 }
 if (amount < 0) {
 return 0;
 }

 // 使用一维记忆化数组
 vector<int> memo(amount + 1, -1);
 return dfsOptimized(coins, 0, amount, memo);
}

int dfsOptimized(vector<int>& coins, int index, int amount, vector<int>& memo) {
 if (amount == 0) {
 return 1;
 }
 if (amount < 0 || index >= coins.size()) {
 return 0;
 }

 if (memo[amount] != -1) {
 return memo[amount];
 }

 int count = 0;
 // 考虑当前硬币

```

```

 if (amount >= coins[index]) {
 count += dfsOptimized(coins, index, amount - coins[index], memo);
 }
 // 跳过当前硬币
 count += dfsOptimized(coins, index + 1, amount, memo);

 memo[amount] = count;
 return count;
}

};

/***
 * 测试函数
 */
void testCoinChangeII() {
 Solution sol;

 // 测试用例
 vector<pair<int, vector<int>>> testCases = {
 {5, {1, 2, 5}}, // 预期: 4
 {3, {2}}, // 预期: 0
 {10, {10}}, // 预期: 1
 {0, {1, 2, 3}}, // 预期: 1
 {5, {1, 2, 3}}, // 预期: 5
 {100, {1, 2, 5}} // 大规模测试
 };

 cout << "零钱兑换 II 问题测试: " << endl;
 for (auto& testCase : testCases) {
 int amount = testCase.first;
 vector<int> coins = testCase.second;

 int result1 = sol.change(amount, coins);
 int result2 = sol.changeOptimized(amount, coins);
 int result3 = sol.changeWithPruning(amount, coins);
 int result4 = sol.changeDFS(amount, coins);
 int result5 = sol.changeDFSOptimized(amount, coins);

 cout << "amount=" << amount << ", coins=[";
 for (size_t i = 0; i < coins.size(); i++) {
 cout << coins[i];
 if (i < coins.size() - 1) cout << ", ";
 }
 }
}

```

```

cout << "] : DP=" << result1
 << ", Optimized=" << result2
 << ", Pruning=" << result3
 << ", DFS=" << result4
 << ", DFS_Opt=" << result5 << endl;

// 验证结果一致性
if (result1 != result2 || result2 != result3 || result3 != result4 || result4 != result5)
{
 cout << "警告：不同方法结果不一致！" << endl;
}

// 性能测试 - 大规模数据
vector<int> largeCoins = {1, 2, 5, 10, 20, 50, 100};
int largeAmount = 1000;

auto startTime = chrono::high_resolution_clock::now();
int largeResult = sol.changeWithPruning(largeAmount, largeCoins);
auto endTime = chrono::high_resolution_clock::now();
auto duration = chrono::duration_cast<chrono::milliseconds>(endTime - startTime);

cout << "大规模测试: coins 长度=" << largeCoins.size()
 << ", amount=" << largeAmount
 << ", 结果=" << largeResult
 << ", 耗时=" << duration.count() << "ms" << endl;

// 边界情况测试
cout << "边界情况测试：" << endl;
vector<int> emptyCoins;
cout << "空数组, amount=0: " << sol.change(0, emptyCoins) << endl; // 预期: 1
cout << "空数组, amount=1: " << sol.change(1, emptyCoins) << endl; // 预期: 0
cout << "负数 amount: " << sol.change(-1, {1, 2, 3}) << endl; // 预期: 0

// 对比组合总和 IV, 验证遍历顺序的重要性
cout << "组合数 vs 排列数对比：" << endl;
vector<int> coins = {1, 2, 5};
int amt = 5;
cout << "零钱兑换 II (组合数) : amount=" << amt << ", coins=[";
for (size_t i = 0; i < coins.size(); i++) {
 cout << coins[i];
 if (i < coins.size() - 1) cout << ", ";
}

```

```

cout << "], 结果=" << sol.change(amt, coins) << endl;

// 模拟组合总和 IV 的排列数计算（错误用法）
vector<int> dp(amt + 1, 0);
dp[0] = 1;
for (int i = 1; i <= amt; i++) {
 for (int coin : coins) {
 if (i >= coin) {
 dp[i] += dp[i - coin];
 }
 }
}
cout << "错误用法（排列数）: amount=" << amt << ", coins=[";
for (size_t i = 0; i < coins.size(); i++) {
 cout << coins[i];
 if (i < coins.size() - 1) cout << ", ";
}
cout << "], 结果=" << dp[amt] << endl;
}

int main() {
 try {
 testCoinChangeII();
 } catch (const exception& e) {
 cerr << "错误: " << e.what() << endl;
 return 1;
 }
 return 0;
}

/*
 * 复杂度分析:
 *
 * 方法 1: 动态规划（组合数）
 * - 时间复杂度: O(amount * n)
 * - 外层循环: n 次（硬币种类数）
 * - 内层循环: amount 次（金额范围）
 * - 空间复杂度: O(amount)
 *
 * 方法 2: 优化的动态规划
 * - 时间复杂度: O(amount * n)（与方法 1 相同）
 * - 空间复杂度: O(amount)
 */

```

- \* 方法 3：带剪枝的动态规划
- \* - 时间复杂度:  $O(amount * n)$  (平均情况下可能更快)
- \* - 空间复杂度:  $O(amount)$
- \*
- \* 方法 4：递归+记忆化搜索
- \* - 时间复杂度:  $O(amount * n)$  (每个状态计算一次)
- \* - 空间复杂度:  $O(amount * n)$  (二维记忆化数组)
- \*
- \* 方法 5：空间优化的递归
- \* - 时间复杂度:  $O(amount * n)$
- \* - 空间复杂度:  $O(amount)$  (一维记忆化数组)
- \*
- \* C++特定优化:
  - \* 1. 使用 vector 代替数组，更安全
  - \* 2. 使用 STL 算法进行排序
  - \* 3. 使用 chrono 进行精确性能测试
  - \* 4. 异常处理使用 C++标准异常
- \*
- \* 关键点分析:
  - \* 1. 组合数 vs 排列数: 本题需要计算组合数，因此遍历顺序很重要
  - \* 2. 外层循环硬币: 确保计算的是组合数 (顺序无关)
  - \* 3. 内层循环金额: 完全背包的正序遍历
- \*
- \* 工程化考量:
  - \* 1. 模块化设计: 将不同解法封装为类方法
  - \* 2. 性能优化: 利用 STL 算法和数据结构
  - \* 3. 测试覆盖: 包含各种边界情况和性能测试
  - \* 4. 错误演示: 展示遍历顺序错误导致的差异
- \*/

=====

文件: Code46\_CoinChangeII.java

=====

```
package class073;

import java.util.Arrays;

// LeetCode 518. 零钱兑换 II
// 题目描述: 给你一个整数数组 coins 表示不同面额的硬币，另给一个整数 amount 表示总金额。
// 请你计算并返回可以凑成总金额的硬币组合数。如果任何硬币组合都无法凑出总金额，返回 0。
// 链接: https://leetcode.cn/problems/coin-change-ii/
//
```

```
// 解题思路:
// 这是一个完全背包问题的组合数变种，需要计算所有可能的组合数。
// 与组合总和 IV 不同，这里顺序不同的序列被视为相同的组合。

//
// 状态定义：dp[i] 表示凑成总金额 i 的硬币组合数
// 状态转移方程：dp[i] += dp[i - coin]，其中 coin 是 coins 中的硬币且 i >= coin
// 初始状态：dp[0] = 1 (空组合)

//
// 关键点：为了计算组合数，需要将硬币循环放在外层，金额循环放在内层

//
// 时间复杂度：O(amount * n)，其中 n 是硬币种类数
// 空间复杂度：O(amount)，使用一维 DP 数组

//
// 工程化考量：
// 1. 异常处理：处理空数组、负数等情况
// 2. 边界条件：amount=0 时返回 1
// 3. 性能优化：排序硬币进行剪枝
// 4. 可读性：清晰的变量命名和注释
```

```
public class Code46_CoinChangeII {

 /**
 * 动态规划解法 - 计算组合数
 * @param coins 不同面额的硬币数组
 * @param amount 目标总金额
 * @return 凑成总金额的硬币组合数
 */
 public static int change(int amount, int[] coins) {
 // 参数验证
 if (coins == null || coins.length == 0) {
 return amount == 0 ? 1 : 0;
 }
 if (amount < 0) {
 return 0;
 }

 // 特殊情况处理
 if (amount == 0) {
 return 1; // 空组合
 }

 // 创建 DP 数组
 int[] dp = new int[amount + 1];
```

```

dp[0] = 1; // 空组合

// 为了计算组合数，需要将硬币循环放在外层
// 金额循环放在内层
for (int coin : coins) {
 for (int i = coin; i <= amount; i++) {
 dp[i] += dp[i - coin];
 }
}

return dp[amount];
}

/***
 * 优化的动态规划解法 - 排序硬币进行剪枝
 */
public static int changeOptimized(int amount, int[] coins) {
 if (coins == null || coins.length == 0) {
 return amount == 0 ? 1 : 0;
 }

 if (amount < 0) {
 return 0;
 }

 // 排序硬币，便于理解和调试（对组合数结果无影响）
 Arrays.sort(coins);
 int[] dp = new int[amount + 1];
 dp[0] = 1;

 for (int coin : coins) {
 for (int i = coin; i <= amount; i++) {
 dp[i] += dp[i - coin];
 }
 }

 return dp[amount];
}

/***
 * 带剪枝的动态规划解法 - 当硬币大于剩余金额时提前终止
 */
public static int changeWithPruning(int amount, int[] coins) {
 if (coins == null || coins.length == 0) {

```

```

 return amount == 0 ? 1 : 0;
 }

 if (amount < 0) {
 return 0;
 }

// 排序硬币，便于剪枝
Arrays.sort(coins);
int[] dp = new int[amount + 1];
dp[0] = 1;

for (int coin : coins) {
 // 如果硬币面额已经大于 amount，后续硬币更大，直接跳过
 if (coin > amount) {
 continue;
 }
 for (int i = coin; i <= amount; i++) {
 dp[i] += dp[i - coin];
 }
}

return dp[amount];
}

/***
 * 递归+记忆化搜索解法
 */
public static int changeDFS(int amount, int[] coins) {
 if (coins == null || coins.length == 0) {
 return amount == 0 ? 1 : 0;
 }
 if (amount < 0) {
 return 0;
 }

// 使用记忆化数组
Integer[][] memo = new Integer[coins.length][amount + 1];
return dfs(coins, 0, amount, memo);
}

/***
 * 递归辅助函数
 * @param coins 硬币数组

```

```

* @param index 当前考虑的硬币索引
* @param amount 剩余金额
* @param memo 记忆化数组
* @return 组合数
*/
private static int dfs(int[] coins, int index, int amount, Integer[][] memo) {
 // 基础情况
 if (amount == 0) {
 return 1;
 }
 if (amount < 0 || index >= coins.length) {
 return 0;
 }

 // 检查记忆化数组
 if (memo[index][amount] != null) {
 return memo[index][amount];
 }

 int count = 0;
 // 选择当前硬币 0 次或多次
 for (int k = 0; k * coins[index] <= amount; k++) {
 count += dfs(coins, index + 1, amount - k * coins[index], memo);
 }

 memo[index][amount] = count;
 return count;
}

/**
* 空间优化的递归解法 - 一维记忆化
*/
public static int changeDFSOptimized(int amount, int[] coins) {
 if (coins == null || coins.length == 0) {
 return amount == 0 ? 1 : 0;
 }
 if (amount < 0) {
 return 0;
 }

 // 使用一维记忆化数组
 Integer[] memo = new Integer[amount + 1];
 return dfsOptimized(coins, 0, amount, memo);
}

```

```

}

private static int dfsOptimized(int[] coins, int index, int amount, Integer[] memo) {
 if (amount == 0) {
 return 1;
 }
 if (amount < 0 || index >= coins.length) {
 return 0;
 }

 if (memo[amount] != null) {
 return memo[amount];
 }

 int count = 0;
 // 考虑当前硬币
 if (amount >= coins[index]) {
 count += dfsOptimized(coins, index, amount - coins[index], memo);
 }
 // 跳过当前硬币
 count += dfsOptimized(coins, index + 1, amount, memo);

 memo[amount] = count;
 return count;
}

/**
 * 测试方法
 */
public static void main(String[] args) {
 // 测试用例
 int[][] testCases = {
 {5}, {1, 2, 5}, // 预期: 4
 {3}, {2}, // 预期: 0
 {10}, {10}, // 预期: 1
 {0}, {1, 2, 3}, // 预期: 1
 {5}, {1, 2, 3}, // 预期: 5
 {100}, {1, 2, 5} // 大规模测试
 };

 System.out.println("零钱兑换 II 问题测试: ");
 for (int i = 0; i < testCases.length; i += 2) {
 int amount = testCases[i][0];

```

```

int[] coins = testCases[i + 1];

int result1 = change(amount, coins);
int result2 = changeOptimized(amount, coins);
int result3 = changeWithPruning(amount, coins);
int result4 = changeDFS(amount, coins);
int result5 = changeDFSOptimized(amount, coins);

System.out.printf("amount=%d, coins=%s: DP=%d, Optimized=%d, Pruning=%d, DFS=%d,
DFS_Opt=%d%n",
amount, Arrays.toString(coins), result1, result2, result3, result4,
result5);

// 验证结果一致性
if (result1 != result2 || result2 != result3 || result3 != result4 || result4 != result5) {
 System.out.println("警告：不同方法结果不一致！");
}

// 性能测试 - 大规模数据
int[] largeCoins = {1, 2, 5, 10, 20, 50, 100};
int largeAmount = 1000;

long startTime = System.currentTimeMillis();
int largeResult = changeWithPruning(largeAmount, largeCoins);
long endTime = System.currentTimeMillis();

System.out.printf("大规模测试: coins 长度=%d, amount=%d, 结果=%d, 耗时=%dms%n",
largeCoins.length, largeAmount, largeResult, endTime - startTime);

// 边界情况测试
System.out.println("边界情况测试: ");
System.out.println("空数组, amount=0: " + change(new int[] {}, 0)); // 预期: 1
System.out.println("空数组, amount=1: " + change(new int[] {}, 1)); // 预期: 0
System.out.println("负数 amount: " + change(new int[] {1, 2, 3}, -1)); // 预期: 0

// 对比组合总和 IV, 验证遍历顺序的重要性
System.out.println("组合数 vs 排列数对比: ");
int[] coins = {1, 2, 5};
int amt = 5;
System.out.printf("零钱兑换 II (组合数) : amount=%d, coins=%s, 结果=%d%n",
amt, Arrays.toString(coins), change(amt, coins));

```

```

// 模拟组合总和 IV 的排列数计算（错误用法）
int[] dp = new int[amt + 1];
dp[0] = 1;
for (int i = 1; i <= amt; i++) {
 for (int coin : coins) {
 if (i >= coin) {
 dp[i] += dp[i - coin];
 }
 }
}
System.out.printf("错误用法（排列数）：amount=%d, coins=%s, 结果=%d%n",
 amt, Arrays.toString(coins), dp[amt]);
}

/*
* 复杂度分析：
*
* 方法 1：动态规划（组合数）
* - 时间复杂度：O(amount * n)
* - 外层循环：n 次（硬币种类数）
* - 内层循环：amount 次（金额范围）
* - 空间复杂度：O(amount)
*
* 方法 2：优化的动态规划
* - 时间复杂度：O(amount * n)（与方法 1 相同）
* - 空间复杂度：O(amount)
*
* 方法 3：带剪枝的动态规划
* - 时间复杂度：O(amount * n)（平均情况下可能更快）
* - 空间复杂度：O(amount)
*
* 方法 4：递归+记忆化搜索
* - 时间复杂度：O(amount * n)（每个状态计算一次）
* - 空间复杂度：O(amount * n)（二维记忆化数组）
*
* 方法 5：空间优化的递归
* - 时间复杂度：O(amount * n)
* - 空间复杂度：O(amount)（一维记忆化数组）
*
* 关键点分析：
* 1. 组合数 vs 排列数：本题需要计算组合数，因此遍历顺序很重要

```

- \* 2. 外层循环硬币：确保计算的是组合数（顺序无关）
- \* 3. 内层循环金额：完全背包的正序遍历
- \*
- \* 与组合总和 IV 的区别：
  - \* - 零钱兑换 II：计算组合数（顺序不同的序列视为相同）
  - \* - 组合总和 IV：计算排列数（顺序不同的序列视为不同）
- \*
- \* 工程化考量：
  - \* 1. 多解法对比：提供不同实现便于理解和选择
  - \* 2. 性能测试：包含大规模数据测试
  - \* 3. 边界测试：验证各种边界情况
  - \* 4. 错误演示：展示遍历顺序错误导致的差异
- \*
- \* 面试要点：
  - \* 1. 理解组合数和排列数的本质区别
  - \* 2. 掌握动态规划中遍历顺序的重要性
  - \* 3. 能够分析不同解法的时空复杂度
  - \* 4. 了解记忆化搜索的实现技巧

\*/

=====

文件：Code46\_CoinChangeII.py

=====

```
from typing import List
import sys
from functools import lru_cache

LeetCode 518. 零钱兑换 II
题目描述：给你一个整数数组 coins 表示不同面额的硬币，另给一个整数 amount 表示总金额。
请你计算并返回可以凑成总金额的硬币组合数。如果任何硬币组合都无法凑出总金额，返回 0。
链接：https://leetcode.cn/problems/coin-change-ii/
#
解题思路：
这是一个完全背包问题的组合数变种，需要计算所有可能的组合数。
与组合总和 IV 不同，这里顺序不同的序列被视为相同的组合。
#
状态定义：dp[i] 表示凑成总金额 i 的硬币组合数
状态转移方程：dp[i] += dp[i - coin]，其中 coin 是 coins 中的硬币且 i >= coin
初始状态：dp[0] = 1 (空组合)
#
关键点：为了计算组合数，需要将硬币循环放在外层，金额循环放在内层
#
```

```
时间复杂度: O(amount * n), 其中 n 是硬币种类数
空间复杂度: O(amount), 使用一维 DP 数组
#
工程化考量:
1. 异常处理: 处理空数组、负数等情况
2. 边界条件: amount=0 时返回 1
3. 性能优化: 利用 Python 特性进行优化
4. 可读性: 清晰的变量命名和类型注解

class CoinChangeII:
 """
 零钱兑换 II 问题的多种解法
 """

 @staticmethod
 def change_dp(amount: int, coins: List[int]) -> int:
 """
 动态规划解法 - 计算组合数

 Args:
 amount: 目标总金额
 coins: 不同面额的硬币数组

 Returns:
 int: 凑成总金额的硬币组合数
 """

 # 参数验证
 if not coins:
 return 1 if amount == 0 else 0
 if amount < 0:
 return 0

 # 特殊情况处理
 if amount == 0:
 return 1 # 空组合

 # 创建 DP 数组
 dp = [0] * (amount + 1)
 dp[0] = 1 # 空组合

 # 为了计算组合数, 需要将硬币循环放在外层
 # 金额循环放在内层
 for coin in coins:
```

```
 for i in range(coin, amount + 1):
 dp[i] += dp[i - coin]

 return dp[amount]
```

```
@staticmethod
def change_optimized(amount: int, coins: List[int]) -> int:
 """
```

优化的动态规划解法 – 排序硬币

Args:

```
 amount: 目标总金额
 coins: 不同面额的硬币数组
```

Returns:

```
 int: 凑成总金额的硬币组合数
 """
```

```
if not coins:
 return 1 if amount == 0 else 0
if amount < 0:
 return 0
```

# 排序硬币, 便于理解和调试 (对组合数结果无影响)

```
coins_sorted = sorted(coins)
dp = [0] * (amount + 1)
dp[0] = 1
```

```
for coin in coins_sorted:
 for i in range(coin, amount + 1):
 dp[i] += dp[i - coin]
```

```
return dp[amount]
```

```
@staticmethod
def change_with_pruning(amount: int, coins: List[int]) -> int:
 """
```

带剪枝的动态规划解法

Args:

```
 amount: 目标总金额
 coins: 不同面额的硬币数组
```

Returns:

```

int: 凑成总金额的硬币组合数
"""

if not coins:
 return 1 if amount == 0 else 0
if amount < 0:
 return 0

排序硬币，便于剪枝
coins_sorted = sorted(coins)
dp = [0] * (amount + 1)
dp[0] = 1

for coin in coins_sorted:
 # 如果硬币面额已经大于 amount，后续硬币更大，直接跳过
 if coin > amount:
 continue
 for i in range(coin, amount + 1):
 dp[i] += dp[i - coin]

return dp[amount]

```

```

@staticmethod
@lru_cache(maxsize=None)
def _change_dfs(coins_tuple: tuple, index: int, amount: int) -> int:
"""
递归辅助函数 - 使用 lru_cache 进行记忆化

```

Args:

- coins\_tuple: 转换为元组的硬币数组
- index: 当前考虑的硬币索引
- amount: 剩余金额

Returns:

- int: 组合数

```

"""

基础情况
if amount == 0:
 return 1
if amount < 0 or index >= len(coins_tuple):
 return 0

count = 0
coin = coins_tuple[index]
```

```
选择当前硬币 0 次或多次
k = 0
while k * coin <= amount:
 count += CoinChangeII._change_dfs(coins_tuple, index + 1, amount - k * coin)
 k += 1

return count
```

```
@staticmethod
def change_dfs(amount: int, coins: List[int]) -> int:
 """
```

递归+记忆化搜索解法

Args:

amount: 目标总金额  
coins: 不同面额的硬币数组

Returns:

int: 凑成总金额的硬币组合数

"""

```
if not coins:
 return 1 if amount == 0 else 0
if amount < 0:
 return 0
```

# 将列表转换为元组以便缓存

```
coins_tuple = tuple(sorted(coins))
return CoinChangeII._change_dfs(coins_tuple, 0, amount)
```

```
@staticmethod
```

```
@lru_cache(maxsize=None)
def _change_dfs_optimized(coins_tuple: tuple, amount: int) -> int:
 """
```

空间优化的递归辅助函数

Args:

coins\_tuple: 转换为元组的硬币数组  
amount: 剩余金额

Returns:

int: 组合数

"""

```
if amount == 0:
 return 1
if amount < 0:
 return 0

count = 0
for i, coin in enumerate(coins_tuple):
 if amount >= coin:
 # 考虑当前硬币（可以重复使用）
 count += CoinChangeII._change_dfs_optimized(coins_tuple[i:], amount - coin)

return count
```

```
@staticmethod
def change_dfs_optimized(amount: int, coins: List[int]) -> int:
 """
```

空间优化的递归解法

Args:

amount: 目标总金额  
coins: 不同面额的硬币数组

Returns:

int: 凑成总金额的硬币组合数

```
"""
if not coins:
 return 1 if amount == 0 else 0
if amount < 0:
 return 0
```

# 将列表转换为元组以便缓存

```
coins_tuple = tuple(sorted(coins))
return CoinChangeII._change_dfs_optimized(coins_tuple, amount)
```

```
@staticmethod
```

```
def run_tests():
```

"""

运行测试用例，验证所有方法的正确性

"""

# 测试用例

```
test_cases = [
 (5, [1, 2, 5]), # 预期: 4
 (3, [2]), # 预期: 0
```

```

(10, [10]), # 预期: 1
(0, [1, 2, 3]), # 预期: 1
(5, [1, 2, 3]), # 预期: 5
(100, [1, 2, 5]) # 大规模测试
]

print("零钱兑换 II 问题测试: ")
for amount, coins in test_cases:
 try:
 result1 = CoinChangeII.change_dp(amount, coins)
 result2 = CoinChangeII.change_optimized(amount, coins)
 result3 = CoinChangeII.change_with_pruning(amount, coins)
 result4 = CoinChangeII.change_dfs(amount, coins)
 result5 = CoinChangeII.change_dfs_optimized(amount, coins)

 print(f"amount={amount}, coins={coins}: "
 f"DP={result1}, Optimized={result2}, Pruning={result3}, "
 f"DFS={result4}, DFS_Opt={result5}")

 # 验证结果一致性
 if result1 != result2 or result2 != result3 or result3 != result4 or result4 != result5:
 print("警告: 不同方法结果不一致!")
 except Exception as e:
 print(f"测试 amount={amount}, coins={coins} 时发生错误: {e}")

性能测试 - 大规模数据
import time
large_coins = [1, 2, 5, 10, 20, 50, 100]
large_amount = 1000

start_time = time.time()
large_result = CoinChangeII.change_with_pruning(large_amount, large_coins)
end_time = time.time()

print(f"大规模测试: coins 长度={len(large_coins)}, amount={large_amount}, "
 f"结果={large_result}, 耗时={end_time - start_time:.4f}秒")

边界情况测试
print("边界情况测试: ")
print(f"空数组, amount=0: {CoinChangeII.change_dp(0, [])}") # 预期: 1
print(f"空数组, amount=1: {CoinChangeII.change_dp(1, [])}") # 预期: 0

```

```

print(f"负数 amount: {CoinChangeII.change_dp(-1, [1, 2, 3])}") # 预期: 0

对比组合总和 IV，验证遍历顺序的重要性
print("组合数 vs 排列数对比: ")
coins = [1, 2, 5]
amt = 5
print(f"零钱兑换 II (组合数) : amount={amt}, coins={coins}, 结果
={CoinChangeII.change_dp(amt, coins)}")

模拟组合总和 IV 的排列数计算 (错误用法)
dp = [0] * (amt + 1)
dp[0] = 1
for i in range(1, amt + 1):
 for coin in coins:
 if i >= coin:
 dp[i] += dp[i - coin]
print(f"错误用法 (排列数) : amount={amt}, coins={coins}, 结果={dp[amt]}")

def main():
 """
 主函数 - 运行测试和演示
 """
 try:
 CoinChangeII.run_tests()
 except Exception as e:
 print(f"程序执行错误: {e}")
 return 1
 return 0

if __name__ == "__main__":
 sys.exit(main())

```

"""

复杂度分析:

方法 1: 动态规划 (组合数)

- 时间复杂度:  $O(amount * n)$ 
  - 外层循环:  $n$  次 (硬币种类数)
  - 内层循环:  $amount$  次 (金额范围)
- 空间复杂度:  $O(amount)$

方法 2: 优化的动态规划

- 时间复杂度:  $O(amount * n)$  (与方法 1 相同)

- 空间复杂度:  $O(amount)$

#### 方法 3: 带剪枝的动态规划

- 时间复杂度:  $O(amount * n)$  (平均情况下可能更快)
- 空间复杂度:  $O(amount)$

#### 方法 4: 递归+记忆化搜索

- 时间复杂度:  $O(amount * n)$  (每个状态计算一次)
- 空间复杂度:  $O(amount * n)$  (记忆化缓存)

#### 方法 5: 空间优化的递归

- 时间复杂度:  $O(amount * n)$
- 空间复杂度:  $O(amount)$  (一维记忆化缓存)

#### Python 特定优化:

1. 使用 `lru_cache` 进行自动记忆化
2. 利用 Python 的大整数特性, 无需担心溢出
3. 使用类型注解提高代码可读性
4. 利用元组进行缓存, 避免列表的可变性问题

#### 关键点分析:

1. 组合数 vs 排列数: 本题需要计算组合数, 因此遍历顺序很重要
2. 外层循环硬币: 确保计算的是组合数 (顺序无关)
3. 内层循环金额: 完全背包的正序遍历
4. Python 优势: 自动处理大整数, 简化实现

#### 工程化考量:

1. 模块化设计: 将不同解法封装为静态方法
2. 异常处理: 完善的参数验证和错误处理
3. 性能监控: 包含性能测试和时间测量
4. 测试覆盖: 包含各种边界情况和对比测试

#### 面试要点:

1. 理解组合数和排列数的本质区别
2. 掌握动态规划中遍历顺序的重要性
3. 了解记忆化搜索的实现技巧
4. 能够分析不同解法的时空复杂度
5. 了解 Python 特性在算法实现中的应用

"""\p

```
=====
#include <iostream>
#include <vector>
#include <algorithm>
#include <climits>
#include <random>

using namespace std;

// POJ 2184 Cow Exhibition
// 题目描述：奶牛们想证明它们是聪明而风趣的。为此，贝西筹备了一个奶牛博览会，
// 她已经对 N 头奶牛进行了面试，确定了每头奶牛的聪明度和幽默度。
// 贝西可以选择任意数量的奶牛参加展览，但希望总聪明度和总幽默度都非负。
// 在满足条件的情况下，使得总聪明度与总幽默度的和最大。
// 链接: http://poj.org/problem?id=2184
//
// 解题思路：
// 这是一个二维费用的 01 背包问题，需要同时考虑两个维度（聪明度和幽默度）。
// 由于两个维度都可能为负数，需要进行坐标平移。
//
// 状态定义：dp[i][j] 表示前 i 头奶牛，聪明度总和为 j 时的最大幽默度总和
// 状态转移方程：
// dp[i][j] = max(dp[i-1][j], dp[i-1][j-smart[i]] + funny[i])
//
// 关键点：
// 1. 坐标平移：由于聪明度可能为负数，需要将坐标平移到非负数范围
// 2. 二维费用：同时考虑聪明度和幽默度两个维度
// 3. 状态优化：使用滚动数组优化空间复杂度
//
// 时间复杂度：O(N * range)，其中 range 是聪明度的可能范围
// 空间复杂度：O(range)，使用滚动数组优化
//
// 工程化考量：
// 1. 异常处理：处理空输入、边界值等情况
// 2. 性能优化：坐标平移和滚动数组优化
// 3. 边界条件：处理负数范围和结果有效性检查

class CowExhibition {
private:
 static const int OFFSET = 100000; // 坐标偏移量，处理负数
 static const int MAX_RANGE = 200000; // 总范围大小
 static const int INF = INT_MIN / 2; // 表示不可达状态
```

```
public:
 /**
 * 动态规划解法 - 二维费用 01 背包
 * @param cows 奶牛数组，每个奶牛包含聪明度和幽默度
 * @return 最大总聪明度与总幽默度的和
 */

 static int maxCowExhibition(vector<vector<int>>& cows) {
 // 参数验证
 if (cows.empty()) {
 return 0;
 }

 int n = cows.size();

 // 创建 DP 数组，使用滚动数组优化
 vector<int> dp(MAX_RANGE + 1, INF);
 dp[OFFSET] = 0; // 初始状态：聪明度总和为 0， 幽默度总和为 0

 // 遍历每头奶牛
 for (int i = 0; i < n; i++) {
 int smart = cows[i][0];
 int funny = cows[i][1];

 // 根据 smart 的正负决定遍历方向
 if (smart >= 0) {
 // 正数：倒序遍历，避免重复选择
 for (int j = MAX_RANGE; j >= smart; j--) {
 if (dp[j - smart] != INF) {
 dp[j] = max(dp[j], dp[j - smart] + funny);
 }
 }
 } else {
 // 负数：正序遍历
 for (int j = 0; j <= MAX_RANGE + smart; j++) {
 if (dp[j - smart] != INF) {
 dp[j] = max(dp[j], dp[j - smart] + funny);
 }
 }
 }
 }

 // 寻找最大和（聪明度+幽默度）
 int maxSum = 0;
```

```

for (int j = OFFSET; j <= MAX_RANGE; j++) {
 if (dp[j] >= 0) { // 幽默度总和需要非负
 maxSum = max(maxSum, j - OFFSET + dp[j]);
 }
}

return maxSum;
}

/***
 * 优化的动态规划解法 - 使用二维数组便于理解
 */
static int maxCowExhibition2D(vector<vector<int>>& cows) {
 if (cows.empty()) {
 return 0;
 }

 int n = cows.size();

 // 计算聪明度的可能范围
 int minSmart = 0, maxSmart = 0;
 for (auto& cow : cows) {
 if (cow[0] < 0) minSmart += cow[0];
 else maxSmart += cow[0];
 }

 int range = maxSmart - minSmart;
 int offset = -minSmart;

 // 创建二维 DP 数组
 vector<vector<int>> dp(n + 1, vector<int>(range + 1, INF));
 dp[0][offset] = 0;

 // 动态规划
 for (int i = 1; i <= n; i++) {
 int smart = cows[i - 1][0];
 int funny = cows[i - 1][1];

 for (int j = 0; j <= range; j++) {
 // 不选当前奶牛
 dp[i][j] = dp[i - 1][j];
 if (j >= smart) {
 dp[i][j] = max(dp[i][j], dp[i - 1][j - smart] + funny);
 }
 }
 }
}

```

```

 int prev = j - smart;
 if (prev >= 0 && prev <= range && dp[i - 1][prev] != INF) {
 dp[i][j] = max(dp[i][j], dp[i - 1][prev] + funny);
 }
 }
}

// 寻找最大和
int maxSum = 0;
for (int j = offset; j <= range; j++) {
 if (dp[n][j] >= 0) {
 maxSum = max(maxSum, j - offset + dp[n][j]);
 }
}

return maxSum;
}

/***
 * 空间优化的解法 - 只记录有效状态
 */
static int maxCowExhibitionOptimized(vector<vector<int>>& cows) {
 if (cows.empty()) {
 return 0;
 }

 // 分离正负聪明度的奶牛
 vector<vector<int>> positiveCows;
 vector<vector<int>> negativeCows;

 for (auto& cow : cows) {
 if (cow[0] >= 0) {
 positiveCows.push_back(cow);
 } else {
 negativeCows.push_back(cow);
 }
 }

 // 处理正数聪明度的奶牛
 vector<int> dp(MAX_RANGE + 1, INF);
 dp[OFFSET] = 0;

 for (auto& cow : positiveCows) {

```

```

int smart = cow[0];
int funny = cow[1];
for (int j = MAX_RANGE; j >= smart; j--) {
 if (dp[j - smart] != INF) {
 dp[j] = max(dp[j], dp[j - smart] + funny);
 }
}
}

// 处理负数聪明度的奶牛
for (auto& cow : negativeCows) {
 int smart = cow[0];
 int funny = cow[1];
 for (int j = 0; j <= MAX_RANGE + smart; j++) {
 if (dp[j - smart] != INF) {
 dp[j] = max(dp[j], dp[j - smart] + funny);
 }
 }
}

// 寻找最大和
int maxSum = 0;
for (int j = OFFSET; j <= MAX_RANGE; j++) {
 if (dp[j] >= 0) {
 maxSum = max(maxSum, j - OFFSET + dp[j]);
 }
}

return maxSum;
}

};

/***
 * 测试函数
 */
void testCowExhibition() {
 // 测试用例
 vector<vector<vector<int>>> testCases = {
 // 示例测试用例
 {
 {5, 1},
 {1, 5},
 {-5, 5},

```

```

{5, -1}
},
// 边界测试用例
{
{10, 20},
{15, 15}
},
// 包含负数的测试用例
{
{-1, 100},
{2, 50},
{-3, 200}
},
// 空测试用例
{}
};

cout << "奶牛展览问题测试: " << endl;
for (size_t i = 0; i < testCases.size(); i++) {
 auto& cows = testCases[i];

 int result1 = CowExhibition::maxCowExhibition(cows);
 int result2 = CowExhibition::maxCowExhibition2D(cows);
 int result3 = CowExhibition::maxCowExhibitionOptimized(cows);

 cout << "测试用例" << i + 1 << ": 奶牛数量=" << cows.size()
 << ", 方法 1=" << result1 << ", 方法 2=" << result2
 << ", 方法 3=" << result3 << endl;

 // 验证结果一致性
 if (result1 != result2 || result2 != result3) {
 cout << "警告: 不同方法结果不一致!" << endl;
 }
}

// 性能测试 - 大规模数据
int n = 100;
vector<vector<int>> largeCows;
// 生成随机测试数据
random_device rd;
mt19937 gen(rd());
uniform_int_distribution<> smartDist(-100, 100);
uniform_int_distribution<> funnyDist(0, 100);

```

```

for (int i = 0; i < n; i++) {
 largeCows.push_back({smartDist(gen), funnyDist(gen)});
}

auto startTime = chrono::high_resolution_clock::now();
int largeResult = CowExhibition::maxCowExhibitionOptimized(largeCows);
auto endTime = chrono::high_resolution_clock::now();
auto duration = chrono::duration_cast<chrono::milliseconds>(endTime - startTime);

cout << "大规模测试: 奶牛数量=" << n << ", 结果=" << largeResult
 << ", 耗时=" << duration.count() << "ms" << endl;

// 边界情况测试
cout << "边界情况测试: " << endl;
vector<vector<int>> emptyCows;
cout << "空数组: " << CowExhibition::maxCowExhibition(emptyCows) << endl;

vector<vector<int>> singleCow = {{10, 20}};
cout << "单头奶牛: " << CowExhibition::maxCowExhibition(singleCow) << endl;

vector<vector<int>> negativeCows = {{-1, 5}, {-2, 10}};
cout << "全负数聪明度: " << CowExhibition::maxCowExhibition(negativeCows) << endl;
}

int main() {
 try {
 testCowExhibition();
 } catch (const exception& e) {
 cerr << "错误: " << e.what() << endl;
 return 1;
 }
 return 0;
}

/*
 * 复杂度分析:
 *
 * 方法 1: 动态规划 (滚动数组)
 * - 时间复杂度: O(N * range)
 * - N: 奶牛数量
 * - range: 聪明度的可能范围 (经过坐标平移)
 * - 空间复杂度: O(range)

```

```
*
* 方法 2: 二维动态规划
* - 时间复杂度: O(N * range)
* - 空间复杂度: O(N * range)

*
* 方法 3: 优化的动态规划
* - 时间复杂度: O(N * range) (但常数更小)
* - 空间复杂度: O(range)

*
* C++特定优化:
* 1. 使用 vector 代替数组, 更安全
* 2. 使用 STL 算法进行最大值计算
* 3. 使用随机数生成器进行性能测试
* 4. 使用 chrono 进行精确性能测量

*
* 关键点分析:
* 1. 坐标平移: 处理负数聪明度, 将坐标平移到非负数范围
* 2. 遍历方向: 根据 smart 的正负决定遍历方向 (01 背包特性)
* 3. 状态有效性: 只考虑幽默度非负的状态
* 4. 结果计算: 聪明度+幽默度的最大和

*
* 工程化考量:
* 1. 模块化设计: 将不同解法封装为静态方法
* 2. 异常处理: 使用 try-catch 处理异常
* 3. 性能优化: 利用 STL 容器和算法
* 4. 测试覆盖: 包含各种边界情况和性能测试

*
* 面试要点:
* 1. 理解二维费用背包问题的特点
* 2. 掌握坐标平移处理负数的方法
* 3. 了解不同遍历方向的原因
* 4. 能够分析算法的时空复杂度
*/
```

=====

文件: Code47\_CowExhibition.java

=====

```
package class073;

import java.util.Arrays;

// POJ 2184 Cow Exhibition
```

```
// 题目描述：奶牛们想证明它们是聪明而风趣的。为此，贝西筹备了一个奶牛博览会，
// 她已经对 N 头奶牛进行了面试，确定了每头奶牛的聪明度和幽默度。
// 贝西可以选择任意数量的奶牛参加展览，但希望总聪明度和总幽默度都非负。
// 在满足条件的情况下，使得总聪明度与总幽默度的和最大。
// 链接: http://poj.org/problem?id=2184

//
// 解题思路：
// 这是一个二维费用的 01 背包问题，需要同时考虑两个维度（聪明度和幽默度）。
// 由于两个维度都可能为负数，需要进行坐标平移。

//
// 状态定义: dp[i][j] 表示前 i 头奶牛，聪明度总和为 j 时的最大幽默度总和
// 状态转移方程：
// dp[i][j] = max(dp[i-1][j], dp[i-1][j-smart[i]] + funny[i])

//
// 关键点：
// 1. 坐标平移：由于聪明度可能为负数，需要将坐标平移到非负数范围
// 2. 二维费用：同时考虑聪明度和幽默度两个维度
// 3. 状态优化：使用滚动数组优化空间复杂度

//
// 时间复杂度: O(N * range)，其中 range 是聪明度的可能范围
// 空间复杂度: O(range)，使用滚动数组优化

//
// 工程化考量：
// 1. 异常处理：处理空输入、边界值等情况
// 2. 性能优化：坐标平移和滚动数组优化
// 3. 边界条件：处理负数范围和结果有效性检查
```

```
public class Code47_CowExhibition {

 private static final int OFFSET = 100000; // 坐标偏移量，处理负数
 private static final int MAX_RANGE = 200000; // 总范围大小
 private static final int INF = Integer.MIN_VALUE / 2; // 表示不可达状态

 /**
 * 动态规划解法 - 二维费用 01 背包
 * @param cows 奶牛数组，每个奶牛包含聪明度和幽默度
 * @return 最大总聪明度与总幽默度的和
 */
 public static int maxCowExhibition(int[][] cows) {
 // 参数验证
 if (cows == null || cows.length == 0) {
 return 0;
 }
 }
```

```

int n = cows.length;

// 创建 DP 数组，使用滚动数组优化
int[] dp = new int[MAX_RANGE + 1];
Arrays.fill(dp, INF);
dp[OFFSET] = 0; // 初始状态：聪明度总和为 0，幽默度总和为 0

// 遍历每头奶牛
for (int i = 0; i < n; i++) {
 int smart = cows[i][0];
 int funny = cows[i][1];

 // 根据 smart 的正负决定遍历方向
 if (smart >= 0) {
 // 正数：倒序遍历，避免重复选择
 for (int j = MAX_RANGE; j >= smart; j--) {
 if (dp[j - smart] != INF) {
 dp[j] = Math.max(dp[j], dp[j - smart] + funny);
 }
 }
 } else {
 // 负数：正序遍历
 for (int j = 0; j <= MAX_RANGE + smart; j++) {
 if (dp[j - smart] != INF) {
 dp[j] = Math.max(dp[j], dp[j - smart] + funny);
 }
 }
 }
}

// 寻找最大和（聪明度+幽默度）
int maxSum = 0;
for (int j = OFFSET; j <= MAX_RANGE; j++) {
 if (dp[j] >= 0) { // 幽默度总和需要非负
 maxSum = Math.max(maxSum, j - OFFSET + dp[j]);
 }
}

return maxSum;
}

/***

```

\* 优化的动态规划解法 - 使用二维数组便于理解

\*/

```
public static int maxCowExhibition2D(int[][] cows) {
 if (cows == null || cows.length == 0) {
 return 0;
 }

 int n = cows.length;

 // 计算聪明度的可能范围
 int minSmart = 0, maxSmart = 0;
 for (int[] cow : cows) {
 if (cow[0] < 0) minSmart += cow[0];
 else maxSmart += cow[0];
 }

 int range = maxSmart - minSmart;
 int offset = -minSmart;

 // 创建二维 DP 数组
 int[][] dp = new int[n + 1][range + 1];
 for (int i = 0; i <= n; i++) {
 Arrays.fill(dp[i], INF);
 }
 dp[0][offset] = 0;

 // 动态规划
 for (int i = 1; i <= n; i++) {
 int smart = cows[i - 1][0];
 int funny = cows[i - 1][1];

 for (int j = 0; j <= range; j++) {
 // 不选当前奶牛
 dp[i][j] = dp[i - 1][j];

 // 选当前奶牛
 int prev = j - smart;
 if (prev >= 0 && prev <= range && dp[i - 1][prev] != INF) {
 dp[i][j] = Math.max(dp[i][j], dp[i - 1][prev] + funny);
 }
 }
 }
}
```

```

// 寻找最大和
int maxSum = 0;
for (int j = offset; j <= range; j++) {
 if (dp[n][j] >= 0) {
 maxSum = Math.max(maxSum, j - offset + dp[n][j]);
 }
}

return maxSum;
}

/***
 * 空间优化的解法 - 只记录有效状态
 */
public static int maxCowExhibitionOptimized(int[][] cows) {
 if (cows == null || cows.length == 0) {
 return 0;
 }

 // 分离正负聪明度的奶牛
 java.util.List<int[]> positiveCows = new java.util.ArrayList<>();
 java.util.List<int[]> negativeCows = new java.util.ArrayList<>();

 for (int[] cow : cows) {
 if (cow[0] >= 0) {
 positiveCows.add(cow);
 } else {
 negativeCows.add(cow);
 }
 }

 // 处理正数聪明度的奶牛
 int[] dp = new int[MAX_RANGE + 1];
 Arrays.fill(dp, INF);
 dp[OFFSET] = 0;

 for (int[] cow : positiveCows) {
 int smart = cow[0];
 int funny = cow[1];
 for (int j = MAX_RANGE; j >= smart; j--) {
 if (dp[j - smart] != INF) {
 dp[j] = Math.max(dp[j], dp[j - smart] + funny);
 }
 }
 }
}

```

```
 }

 }

// 处理负数聪明度的奶牛
for (int[] cow : negativeCows) {
 int smart = cow[0];
 int funny = cow[1];
 for (int j = 0; j <= MAX_RANGE + smart; j++) {
 if (dp[j - smart] != INF) {
 dp[j] = Math.max(dp[j], dp[j - smart] + funny);
 }
 }
}

// 寻找最大和
int maxSum = 0;
for (int j = OFFSET; j <= MAX_RANGE; j++) {
 if (dp[j] >= 0) {
 maxSum = Math.max(maxSum, j - OFFSET + dp[j]);
 }
}

return maxSum;
}

/***
 * 测试方法
 */
public static void main(String[] args) {
 // 测试用例
 int[][][] testCases = {
 // 示例测试用例
 {
 {5, 1},
 {1, 5},
 {-5, 5},
 {5, -1}
 },
 // 边界测试用例
 {
 {10, 20},
 {15, 15}
 },
 };
}
```

```

// 包含负数的测试用例
{
 {-1, 100},
 {2, 50},
 {-3, 200}
},
// 空测试用例
{}
};

System.out.println("奶牛展览问题测试：");
for (int i = 0; i < testCases.length; i++) {
 int[][] cows = testCases[i];

 int result1 = maxCowExhibition(cows);
 int result2 = maxCowExhibition2D(cows);
 int result3 = maxCowExhibitionOptimized(cows);

 System.out.printf("测试用例%d: 奶牛数量=%d, 方法 1=%d, 方法 2=%d, 方法 3=%d%n",
 i + 1, cows.length, result1, result2, result3);

 // 验证结果一致性
 if (result1 != result2 || result2 != result3) {
 System.out.println("警告：不同方法结果不一致！");
 }
}

// 性能测试 - 大规模数据
int n = 100;
int[][] largeCows = generateLargeTestData(n);

long startTime = System.currentTimeMillis();
int largeResult = maxCowExhibitionOptimized(largeCows);
long endTime = System.currentTimeMillis();

System.out.printf("大规模测试: 奶牛数量=%d, 结果=%d, 耗时=%dms%n",
 n, largeResult, endTime - startTime);

// 边界情况测试
System.out.println("边界情况测试：");
System.out.println("空数组: " + maxCowExhibition(new int[][]{}));
System.out.println("单头奶牛: " + maxCowExhibition(new int[][]{{10, 20}}));
System.out.println("全负数聪明度: " + maxCowExhibition(new int[][]{{-1, 5}, {-2, 10}}));

```

```

 }

 /**
 * 生成大规模测试数据
 */
 private static int[][] generateLargeTestData(int n) {
 int[][] cows = new int[n][2];
 java.util.Random random = new java.util.Random();

 for (int i = 0; i < n; i++) {
 // 生成-100 到 100 之间的聪明度
 cows[i][0] = random.nextInt(201) - 100;
 // 生成 0 到 100 之间的幽默度
 cows[i][1] = random.nextInt(101);
 }

 return cows;
 }
}

```

```

/*
 * 复杂度分析:
 *
 * 方法 1: 动态规划 (滚动数组)
 * - 时间复杂度: O(N * range)
 * - N: 奶牛数量
 * - range: 聪明度的可能范围 (经过坐标平移)
 * - 空间复杂度: O(range)
 *
 * 方法 2: 二维动态规划
 * - 时间复杂度: O(N * range)
 * - 空间复杂度: O(N * range)
 *
 * 方法 3: 优化的动态规划
 * - 时间复杂度: O(N * range) (但常数更小)
 * - 空间复杂度: O(range)
 *
 * 关键点分析:
 * 1. 坐标平移: 处理负数聪明度, 将坐标平移到非负数范围
 * 2. 遍历方向: 根据 smart 的正负决定遍历方向 (01 背包特性)
 * 3. 状态有效性: 只考虑幽默度非负的状态
 * 4. 结果计算: 聪明度+幽默度的最大和
 */

```

- \* 工程化考量:
  - \* 1. 参数验证: 处理各种边界输入
  - \* 2. 性能优化: 使用滚动数组和分离正负策略
  - \* 3. 测试覆盖: 包含正常、边界、性能测试
  - \* 4. 代码可读性: 清晰的变量命名和注释
- \*
- \* 面试要点:
  - \* 1. 理解二维费用背包问题的特点
  - \* 2. 掌握坐标平移处理负数的方法
  - \* 3. 了解不同遍历方向的原因
  - \* 4. 能够分析算法的时空复杂度
- \*
- \* 扩展应用:
  - \* 1. 多维度优化问题
  - \* 2. 资源分配问题
  - \* 3. 投资组合优化
  - \* 4. 特征选择问题
- \*/

=====

文件: Code47\_CowExhibition.py

=====

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

POJ 2184 Cow Exhibition
题目描述: 奶牛们想证明它们是聪明而风趣的。为此, 贝西筹备了一个奶牛博览会,
她已经对 N 头奶牛进行了面试, 确定了每头奶牛的聪明度和幽默度。
贝西可以选择任意数量的奶牛参加展览, 但希望总聪明度和总幽默度都非负。
在满足条件的情况下, 使得总聪明度与总幽默度的和最大。
链接: http://poj.org/problem?id=2184
#
解题思路:
这是一个二维费用的 01 背包问题, 需要同时考虑两个维度(聪明度和幽默度)。
由于两个维度都可能为负数, 需要进行坐标平移。
#
状态定义: dp[i][j] 表示前 i 头奶牛, 聪明度总和为 j 时的最大幽默度总和
状态转移方程:
dp[i][j] = max(dp[i-1][j], dp[i-1][j-smart[i]] + funny[i])
#
```

```

关键点:
1. 坐标平移: 由于聪明度可能为负数, 需要将坐标平移到非负数范围
2. 二维费用: 同时考虑聪明度和幽默度两个维度
3. 状态优化: 使用滚动数组优化空间复杂度
#
时间复杂度: O(N * range), 其中 range 是聪明度的可能范围
空间复杂度: O(range), 使用滚动数组优化
#
工程化考量:
1. 异常处理: 处理空输入、边界值等情况
2. 性能优化: 坐标平移和滚动数组优化
3. 边界条件: 处理负数范围和结果有效性检查

class CowExhibition:
 """
 奶牛展览问题的多种解法
 """

 OFFSET = 100000 # 坐标偏移量, 处理负数
 MAX_RANGE = 200000 # 总范围大小
 INF = -10**9 # 表示不可达状态

 @staticmethod
 def max_cow_exhibition(cows: List[Tuple[int, int]]) -> int:
 """
 动态规划解法 - 二维费用 01 背包

 Args:
 cows: 奶牛列表, 每个奶牛包含聪明度和幽默度

 Returns:
 int: 最大总聪明度与总幽默度的和
 """

 # 参数验证
 if not cows:
 return 0

 n = len(cows)

 # 创建 DP 数组, 使用滚动数组优化
 dp = [CowExhibition.INF] * (CowExhibition.MAX_RANGE + 1)
 dp[CowExhibition.OFFSET] = 0 # 初始状态: 聪明度总和为 0, 幽默度总和为 0

```

```

遍历每头奶牛
for smart, funny in cows:
 # 根据 smart 的正负决定遍历方向
 if smart >= 0:
 # 正数: 倒序遍历, 避免重复选择
 for j in range(CowExhibition.MAX_RANGE, smart - 1, -1):
 if dp[j - smart] != CowExhibition.INF:
 dp[j] = max(dp[j], dp[j - smart] + funny)
 else:
 # 负数: 正序遍历
 for j in range(0, CowExhibition.MAX_RANGE + smart + 1):
 if dp[j - smart] != CowExhibition.INF:
 dp[j] = max(dp[j], dp[j - smart] + funny)

寻找最大和 (聪明度+幽默度)
max_sum = 0
for j in range(CowExhibition.OFFSET, CowExhibition.MAX_RANGE + 1):
 if dp[j] >= 0: # 幽默度总和需要非负
 max_sum = max(max_sum, j - CowExhibition.OFFSET + dp[j])

return max_sum

```

```

@staticmethod
def max_cow_exhibition_2d(cows: List[Tuple[int, int]]) -> int:
 """

```

优化的动态规划解法 - 使用二维数组便于理解

Args:

cows: 奶牛列表, 每个奶牛包含聪明度和幽默度

Returns:

int: 最大总聪明度与总幽默度的和

"""

if not cows:

return 0

n = len(cows)

# 计算聪明度的可能范围

min\_smart = 0

max\_smart = 0

for smart, \_ in cows:

if smart < 0:

```

 min_smart += smart
 else:
 max_smart += smart

range_size = max_smart - min_smart
offset = -min_smart

创建二维 DP 数组
dp = [[CowExhibition.INF] * (range_size + 1) for _ in range(n + 1)]
dp[0][offset] = 0

动态规划
for i in range(1, n + 1):
 smart, funny = cows[i - 1]

 for j in range(range_size + 1):
 # 不选当前奶牛
 dp[i][j] = dp[i - 1][j]

 # 选当前奶牛
 prev = j - smart
 if 0 <= prev <= range_size and dp[i - 1][prev] != CowExhibition.INF:
 dp[i][j] = max(dp[i][j], dp[i - 1][prev] + funny)

寻找最大和
max_sum = 0
for j in range(offset, range_size + 1):
 if dp[n][j] >= 0:
 max_sum = max(max_sum, j - offset + dp[n][j])

return max_sum

```

```

@staticmethod
def max_cow_exhibition_optimized(cows: List[Tuple[int, int]]) -> int:
 """

```

空间优化的解法 - 只记录有效状态

Args:

cows: 奶牛列表，每个奶牛包含聪明度和幽默度

Returns:

int: 最大总聪明度与总幽默度的和

"""

```

if not cows:
 return 0

分离正负聪明度的奶牛
positive_cows = []
negative_cows = []

for smart, funny in cows:
 if smart >= 0:
 positive_cows.append((smart, funny))
 else:
 negative_cows.append((smart, funny))

处理正数聪明度的奶牛
dp = [CowExhibition.INF] * (CowExhibition.MAX_RANGE + 1)
dp[CowExhibition.OFFSET] = 0

for smart, funny in positive_cows:
 for j in range(CowExhibition.MAX_RANGE, smart - 1, -1):
 if dp[j - smart] != CowExhibition.INF:
 dp[j] = max(dp[j], dp[j - smart] + funny)

处理负数聪明度的奶牛
for smart, funny in negative_cows:
 for j in range(0, CowExhibition.MAX_RANGE + smart + 1):
 if dp[j - smart] != CowExhibition.INF:
 dp[j] = max(dp[j], dp[j - smart] + funny)

寻找最大和
max_sum = 0
for j in range(CowExhibition.OFFSET, CowExhibition.MAX_RANGE + 1):
 if dp[j] >= 0:
 max_sum = max(max_sum, j - CowExhibition.OFFSET + dp[j])

return max_sum

@staticmethod
def run_tests():
 """
 运行测试用例，验证所有方法的正确性
 """
 # 测试用例
 test_cases = [

```

```

示例测试用例
[(5, 1), (1, 5), (-5, 5), (5, -1)],
边界测试用例
[(10, 20), (15, 15)],
包含负数的测试用例
[(-1, 100), (2, 50), (-3, 200)],
空测试用例
[]

]

print("奶牛展览问题测试: ")
for i, cows in enumerate(test_cases, 1):
 try:
 result1 = CowExhibition.max_cow_exhibition(cows)
 result2 = CowExhibition.max_cow_exhibition_2d(cows)
 result3 = CowExhibition.max_cow_exhibition_optimized(cows)

 print(f"测试用例{i}: 奶牛数量={len(cows)}, "
 f"方法 1={result1}, 方法 2={result2}, 方法 3={result3}")

 # 验证结果一致性
 if result1 != result2 or result2 != result3:
 print("警告: 不同方法结果不一致!")

 except Exception as e:
 print(f"测试用例{i}时发生错误: {e}")

性能测试 - 大规模数据
n = 100
large_cows = CowExhibition.generate_large_test_data(n)

start_time = time.time()
large_result = CowExhibition.max_cow_exhibition_optimized(large_cows)
end_time = time.time()

print(f"大规模测试: 奶牛数量={n}, 结果={large_result}, "
 f"耗时={end_time - start_time:.4f}秒")

边界情况测试
print("边界情况测试: ")
print(f"空数组: {CowExhibition.max_cow_exhibition([])}")
print(f"单头奶牛: {CowExhibition.max_cow_exhibition([(10, 20)])}")
print(f"全负数聪明度: {CowExhibition.max_cow_exhibition([(-1, 5), (-2, 10)])}")

```

```

@staticmethod
def generate_large_test_data(n: int) -> List[Tuple[int, int]]:
 """
 生成大规模测试数据

 Args:
 n: 奶牛数量

 Returns:
 List[Tuple[int, int]]: 生成的奶牛数据
 """
 cows = []
 for _ in range(n):
 # 生成-100 到 100 之间的聪明度
 smart = random.randint(-100, 100)
 # 生成 0 到 100 之间的幽默度
 funny = random.randint(0, 100)
 cows.append((smart, funny))
 return cows

def main():
 """
 主函数 - 运行测试和演示
 """
 try:
 CowExhibition.run_tests()
 except Exception as e:
 print(f"程序执行错误: {e}")
 return 1
 return 0

if __name__ == "__main__":
 sys.exit(main())

```

复杂度分析:

方法 1: 动态规划 (滚动数组)

- 时间复杂度:  $O(N * \text{range})$ 
  - $N$ : 奶牛数量
  - $\text{range}$ : 聪明度的可能范围 (经过坐标平移)
- 空间复杂度:  $O(\text{range})$

## 方法 2: 二维动态规划

- 时间复杂度:  $O(N * \text{range})$
- 空间复杂度:  $O(N * \text{range})$

## 方法 3: 优化的动态规划

- 时间复杂度:  $O(N * \text{range})$  (但常数更小)
- 空间复杂度:  $O(\text{range})$

Python 特定优化:

1. 使用列表推导式和生成器表达式
2. 利用 Python 的动态类型特性
3. 使用类型注解提高代码可读性
4. 使用 `random` 模块进行性能测试

关键点分析:

1. 坐标平移: 处理负数聪明度, 将坐标平移到非负数范围
2. 遍历方向: 根据 smart 的正负决定遍历方向 (01 背包特性)
3. 状态有效性: 只考虑幽默度非负的状态
4. 结果计算: 聪明度+幽默度的最大和

工程化考量:

1. 模块化设计: 将不同解法封装为静态方法
2. 异常处理: 完善的参数验证和错误处理
3. 性能监控: 包含性能测试和时间测量
4. 测试覆盖: 包含各种边界情况和性能测试

面试要点:

1. 理解二维费用背包问题的特点
2. 掌握坐标平移处理负数的方法
3. 了解不同遍历方向的原因
4. 能够分析算法的时空复杂度
5. 了解 Python 在算法实现中的优势

扩展应用:

1. 多目标优化问题
2. 资源约束下的最优选择
3. 投资组合优化
4. 特征选择和权重分配

"""

=====

文件: Code48\_Robberies.cpp

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <random>
#include <chrono>

using namespace std;

// HDU 2955 Robberies
// 题目描述: 抢劫犯想抢劫银行, 每个银行有一定的金额和被抓的概率。
// 抢劫犯希望在被抓概率不超过某个值的情况下, 抢劫到最多的钱。
// 链接: https://acm.hdu.edu.cn/showproblem.php?pid=2955
//
// 解题思路:
// 这是一个概率背包问题, 需要将问题转化为标准的 01 背包问题。
// 关键点: 将金额作为背包容量, 将安全概率 (1-被抓概率) 作为价值。
//
// 状态定义: dp[i] 表示抢劫到金额 i 时的最大安全概率
// 状态转移方程: dp[i] = max(dp[i], dp[i-money[j]] * (1-p[j]))
// 初始状态: dp[0] = 1 (抢劫 0 元时安全概率为 1)
//
// 时间复杂度: O(N * totalMoney), 其中 N 是银行数量, totalMoney 是总金额
// 空间复杂度: O(totalMoney), 使用一维 DP 数组
//
// 工程化考量:
// 1. 精度处理: 使用 double 类型处理概率
// 2. 边界条件: 处理概率为 0 或 1 的情况
// 3. 性能优化: 计算总金额作为背包容量上限
// 4. 异常处理: 处理非法输入

class Robberies {
public:
 /**
 * 动态规划解法 - 概率背包问题
 * @param P 最大允许被抓概率
 * @param money 每个银行的金额数组
 * @param prob 每个银行的被抓概率数组
 * @return 在安全概率范围内的最大抢劫金额
 */
 static double rob(double P, vector<int>& money, vector<double>& prob) {
 // 参数验证
```

```

if (money.size() != prob.size()) {
 throw invalid_argument("Money and probability arrays must have same size");
}

if (P < 0 || P > 1) {
 throw invalid_argument("Probability P must be between 0 and 1");
}

int n = money.size();
if (n == 0) {
 return 0;
}

// 计算总金额作为背包容量上限
int totalMoney = 0;
for (int m : money) {
 totalMoney += m;
}

// 创建 DP 数组, dp[i] 表示抢劫到金额 i 时的最大安全概率
vector<double> dp(totalMoney + 1, 0.0);
dp[0] = 1.0; // 抢劫 0 元时安全概率为 1

// 01 背包: 遍历每个银行
for (int i = 0; i < n; i++) {
 int m = money[i];
 double p = prob[i];
 double safeProb = 1 - p; // 安全概率

 // 倒序遍历金额, 避免重复选择
 for (int j = totalMoney; j >= m; j--) {
 if (dp[j - m] > 0) {
 dp[j] = max(dp[j], dp[j - m] * safeProb);
 }
 }
}

// 寻找最大的金额, 使得安全概率 >= 1-P
double minSafeProb = 1 - P;
int maxMoney = 0;
for (int j = totalMoney; j >= 0; j--) {
 if (dp[j] >= minSafeProb) {
 maxMoney = j;
 break;
 }
}

```

```

 }

 }

 return maxMoney;
}

/***
 * 优化的动态规划解法 - 提前终止遍历
 */
static double robOptimized(double P, vector<int>& money, vector<double>& prob) {
 if (money.size() != prob.size()) {
 throw invalid_argument("Money and probability arrays must have same size");
 }
 if (P < 0 || P > 1) {
 throw invalid_argument("Probability P must be between 0 and 1");
 }

 int n = money.size();
 if (n == 0) {
 return 0;
 }

 // 计算总金额
 int totalMoney = 0;
 for (int m : money) {
 totalMoney += m;
 }

 vector<double> dp(totalMoney + 1, 0.0);
 dp[0] = 1.0;

 for (int i = 0; i < n; i++) {
 int m = money[i];
 double safeProb = 1 - prob[i];

 for (int j = totalMoney; j >= m; j--) {
 if (dp[j - m] > 0) {
 double newProb = dp[j - m] * safeProb;
 if (newProb > dp[j]) {
 dp[j] = newProb;
 }
 }
 }
 }
}

```

```

}

double minSafeProb = 1 - P;
// 从大到小遍历，找到第一个满足条件的金额
for (int j = totalMoney; j >= 0; j--) {
 if (dp[j] >= minSafeProb) {
 return j;
 }
}

return 0;
}

/***
 * 另一种思路：将金额作为价值，概率作为约束
 */
static double robAlternative(double P, vector<int>& money, vector<double>& prob) {
 if (money.size() != prob.size()) {
 throw invalid_argument("Money and probability arrays must have same size");
 }
 if (P < 0 || P > 1) {
 throw invalid_argument("Probability P must be between 0 and 1");
 }

 int n = money.size();
 if (n == 0) {
 return 0;
 }

 // 计算总金额
 int totalMoney = 0;
 for (int m : money) {
 totalMoney += m;
 }

 // dp[i]表示达到金额 i 所需的最小被抓概率
 vector<double> dp(totalMoney + 1, 1.0); // 初始化为最大概率 1
 dp[0] = 0.0; // 抢劫 0 元时被抓概率为 0

 for (int i = 0; i < n; i++) {
 int m = money[i];
 double p = prob[i];

```

```

 for (int j = totalMoney; j >= m; j--) {
 // 计算选择当前银行的被抓概率
 double newProb = 1 - (1 - dp[j - m]) * (1 - p);
 if (newProb < dp[j]) {
 dp[j] = newProb;
 }
 }

 }

 // 寻找最大的金额，使得被抓概率 <= P
 for (int j = totalMoney; j >= 0; j--) {
 if (dp[j] <= P) {
 return j;
 }
 }

 return 0;
}

};

/***
 * 测试函数
 */
void testRobberies() {
 // 测试用例
 vector<double> P = {0.1, 0.05, 0.5};
 vector<vector<int>> moneyCases = {
 {10, 20, 30},
 {1, 2, 3, 4},
 {100, 200, 300}
 };
 vector<vector<double>> probCases = {
 {0.05, 0.1, 0.2},
 {0.01, 0.02, 0.03, 0.04},
 {0.3, 0.2, 0.1}
 };

 cout << "抢劫银行问题测试: " << endl;
 for (size_t i = 0; i < P.size(); i++) {
 double p = P[i];
 vector<int> money = moneyCases[i];
 vector<double> prob = probCases[i];

```

```

double result1 = Robberies::rob(p, money, prob);
double result2 = Robberies::robOptimized(p, money, prob);
double result3 = Robberies::robAlternative(p, money, prob);

cout << "P=" << p << ", money=[";
for (size_t j = 0; j < money.size(); j++) {
 cout << money[j];
 if (j < money.size() - 1) cout << ",";
}
cout << "], prob=[";
for (size_t j = 0; j < prob.size(); j++) {
 cout << prob[j];
 if (j < prob.size() - 1) cout << ",";
}
cout << "]: 方法 1=" << result1
 << ", 方法 2=" << result2
 << ", 方法 3=" << result3 << endl;

// 验证结果一致性（允许小的浮点数误差）
if (abs(result1 - result2) > 1e-6 || abs(result2 - result3) > 1e-6) {
 cout << "警告：不同方法结果不一致！" << endl;
}

// 性能测试 - 大规模数据
int n = 50;
vector<int> largeMoney(n);
vector<double> largeProb(n);

// 生成随机测试数据
random_device rd;
mt19937 gen(rd());
uniform_int_distribution<> moneyDist(1, 1000);
uniform_real_distribution<> probDist(0.0, 0.1);

for (int i = 0; i < n; i++) {
 largeMoney[i] = moneyDist(gen);
 largeProb[i] = probDist(gen);
}

auto startTime = chrono::high_resolution_clock::now();
double largeResult = Robberies::robOptimized(0.1, largeMoney, largeProb);
auto endTime = chrono::high_resolution_clock::now();

```

```

auto duration = chrono::duration_cast<chrono::milliseconds>(endTime - startTime);

cout << "大规模测试: 银行数量=" << n << ", 结果=" << largeResult
 << ", 耗时=" << duration.count() << "ms" << endl;

// 边界情况测试
cout << "边界情况测试: " << endl;
vector<int> emptyMoney;
vector<double> emptyProb;
cout << "空数组: " << Robberies::rob(0.1, emptyMoney, emptyProb) << endl;

vector<int> singleMoney = {10};
vector<double> singleProb = {0.1};
cout << "P=0: " << Robberies::rob(0.0, singleMoney, singleProb) << endl;

vector<int> doubleMoney = {10, 20};
vector<double> doubleProb = {0.1, 0.2};
cout << "P=1: " << Robberies::rob(1.0, doubleMoney, doubleProb) << endl;

// 特殊测试: 概率为 0 的银行
cout << "概率为 0 的银行测试: " << endl;
vector<int> specialMoney = {100, 200};
vector<double> specialProb = {0.0, 0.0};
double specialResult = Robberies::rob(0.01, specialMoney, specialProb);
cout << "特殊测试结果: " << specialResult << endl;
}

int main() {
 try {
 testRobberies();
 } catch (const exception& e) {
 cerr << "错误: " << e.what() << endl;
 return 1;
 }
 return 0;
}

/*
 * 复杂度分析:
 *
 * 方法 1: 动态规划 (概率背包)
 * - 时间复杂度: O(N * totalMoney)
 * - N: 银行数量

```

```
* - totalMoney: 总金额
* - 空间复杂度: O(totalMoney)
*
* 方法 2: 优化的动态规划
* - 时间复杂度: O(N * totalMoney) (与方法 1 相同)
* - 空间复杂度: O(totalMoney)
*
* 方法 3: 替代思路的动态规划
* - 时间复杂度: O(N * totalMoney)
* - 空间复杂度: O(totalMoney)
*
* C++特定优化:
* 1. 使用 vector 代替数组, 更安全
* 2. 使用 STL 算法进行最大值计算
* 3. 使用随机数生成器进行性能测试
* 4. 使用 chrono 进行精确性能测量
*
* 关键点分析:
* 1. 问题转化: 将概率问题转化为标准的背包问题
* 2. 精度处理: 使用 double 类型处理概率计算
* 3. 状态定义: dp[i] 表示金额 i 对应的最大安全概率
* 4. 结果提取: 从后向前遍历找到第一个满足条件的金额
*
* 工程化考量:
* 1. 模块化设计: 将不同解法封装为静态方法
* 2. 异常处理: 使用 try-catch 处理异常
* 3. 性能优化: 利用 STL 容器和算法
* 4. 测试覆盖: 包含各种边界情况和性能测试
*
* 面试要点:
* 1. 理解概率背包问题的转化思路
* 2. 掌握浮点数精度处理技巧
* 3. 了解不同状态定义对算法的影响
* 4. 能够分析算法的时空复杂度
*/
=====
```

文件: Code48\_Robberies.java

```
=====
package class073;
```

```
import java.util.Arrays;
```

```

// HDU 2955 Robberies
// 题目描述：抢劫犯想抢劫银行，每个银行有一定的金额和被抓的概率。
// 抢劫犯希望在被抓概率不超过某个值的情况下，抢劫到最多的钱。
// 链接: https://acm.hdu.edu.cn/showproblem.php?pid=2955
//
// 解题思路：
// 这是一个概率背包问题，需要将问题转化为标准的 01 背包问题。
// 关键点：将金额作为背包容量，将安全概率（1-被抓概率）作为价值。
//
// 状态定义：dp[i] 表示抢劫到金额 i 时的最大安全概率
// 状态转移方程：dp[i] = max(dp[i], dp[i-money[j]] * (1-p[j]))
// 初始状态：dp[0] = 1 (抢劫 0 元时安全概率为 1)
//
// 时间复杂度：O(N * totalMoney)，其中 N 是银行数量，totalMoney 是总金额
// 空间复杂度：O(totalMoney)，使用一维 DP 数组
//
// 工程化考量：
// 1. 精度处理：使用 double 类型处理概率
// 2. 边界条件：处理概率为 0 或 1 的情况
// 3. 性能优化：计算总金额作为背包容量上限
// 4. 异常处理：处理非法输入

```

```

public class Code48_Robberies {

 /**
 * 动态规划解法 - 概率背包问题
 * @param P 最大允许被抓概率
 * @param money 每个银行的金额数组
 * @param prob 每个银行的被抓概率数组
 * @return 在安全概率范围内的最大抢劫金额
 */
 public static double rob(double P, int[] money, double[] prob) {
 // 参数验证
 if (money == null || prob == null || money.length != prob.length) {
 throw new IllegalArgumentException("Invalid input parameters");
 }
 if (P < 0 || P > 1) {
 throw new IllegalArgumentException("Probability P must be between 0 and 1");
 }

 int n = money.length;
 if (n == 0) {

```

```

 return 0;
 }

 // 计算总金额作为背包容量上限
 int totalMoney = 0;
 for (int m : money) {
 totalMoney += m;
 }

 // 创建 DP 数组, dp[i] 表示抢劫到金额 i 时的最大安全概率
 double[] dp = new double[totalMoney + 1];
 Arrays.fill(dp, 0);
 dp[0] = 1.0; // 抢劫 0 元时安全概率为 1

 // 01 背包: 遍历每个银行
 for (int i = 0; i < n; i++) {
 int m = money[i];
 double p = prob[i];
 double safeProb = 1 - p; // 安全概率

 // 倒序遍历金额, 避免重复选择
 for (int j = totalMoney; j >= m; j--) {
 if (dp[j - m] > 0) {
 dp[j] = Math.max(dp[j], dp[j - m] * safeProb);
 }
 }
 }

 // 寻找最大的金额, 使得安全概率 >= 1-P
 double minSafeProb = 1 - P;
 int maxMoney = 0;
 for (int j = totalMoney; j >= 0; j--) {
 if (dp[j] >= minSafeProb) {
 maxMoney = j;
 break;
 }
 }

 return maxMoney;
}

/**
 * 优化的动态规划解法 - 提前终止遍历

```

```

*/
public static double robOptimized(double P, int[] money, double[] prob) {
 if (money == null || prob == null || money.length != prob.length) {
 throw new IllegalArgumentException("Invalid input parameters");
 }
 if (P < 0 || P > 1) {
 throw new IllegalArgumentException("Probability P must be between 0 and 1");
 }

 int n = money.length;
 if (n == 0) {
 return 0;
 }

 // 计算总金额
 int totalMoney = 0;
 for (int m : money) {
 totalMoney += m;
 }

 double[] dp = new double[totalMoney + 1];
 Arrays.fill(dp, 0);
 dp[0] = 1.0;

 for (int i = 0; i < n; i++) {
 int m = money[i];
 double safeProb = 1 - prob[i];

 for (int j = totalMoney; j >= m; j--) {
 if (dp[j - m] > 0) {
 double newProb = dp[j - m] * safeProb;
 if (newProb > dp[j]) {
 dp[j] = newProb;
 }
 }
 }
 }

 double minSafeProb = 1 - P;
 // 从大到小遍历，找到第一个满足条件的金额
 for (int j = totalMoney; j >= 0; j--) {
 if (dp[j] >= minSafeProb) {
 return j;
 }
 }
}

```

```

 }

 }

 return 0;
}

/***
 * 另一种思路：将金额作为价值，概率作为约束
 */
public static double robAlternative(double P, int[] money, double[] prob) {
 if (money == null || prob == null || money.length != prob.length) {
 throw new IllegalArgumentException("Invalid input parameters");
 }
 if (P < 0 || P > 1) {
 throw new IllegalArgumentException("Probability P must be between 0 and 1");
 }

 int n = money.length;
 if (n == 0) {
 return 0;
 }

 // 计算总金额
 int totalMoney = 0;
 for (int m : money) {
 totalMoney += m;
 }

 // dp[i]表示达到金额 i 所需的最小被抓概率
 double[] dp = new double[totalMoney + 1];
 Arrays.fill(dp, 1.0); // 初始化为最大概率 1
 dp[0] = 0.0; // 抢劫 0 元时被抓概率为 0

 for (int i = 0; i < n; i++) {
 int m = money[i];
 double p = prob[i];

 for (int j = totalMoney; j >= m; j--) {
 // 计算选择当前银行的被抓概率
 double newProb = 1 - (1 - dp[j - m]) * (1 - p);
 if (newProb < dp[j]) {
 dp[j] = newProb;
 }
 }
 }
}

```



```

// 基础情况：遍历完所有银行
if (index == money.length) {
 return currentMoney;
}

// 检查记忆化数组
if (memo[index][currentMoney] != null) {
 return memo[index][currentMoney];
}

double maxMoney = 0;

// 不抢劫当前银行
maxMoney = Math.max(maxMoney,
 dfs(money, prob, index + 1, currentMoney, currentSafeProb,
 maxCaughtProb, totalMoney, memo));

// 抢劫当前银行
int newMoney = currentMoney + money[index];
double newSafeProb = currentSafeProb * (1 - prob[index]);
double newCaughtProb = 1 - newSafeProb;

// 如果抢劫后被抓概率不超过限制，则可以选择抢劫
if (newCaughtProb <= maxCaughtProb) {
 maxMoney = Math.max(maxMoney,
 dfs(money, prob, index + 1, newMoney, newSafeProb,
 maxCaughtProb, totalMoney, memo));
}

memo[index][currentMoney] = maxMoney;
return maxMoney;
}

/***
 * 测试方法
 */
public static void main(String[] args) {
 // 测试用例
 double[] P = {0.1, 0.05, 0.5};
 int[][] moneyCases = {
 {10, 20, 30},
 {1, 2, 3, 4},
 {100, 200, 300}
}

```

```

} ;

double[][] probCases = {
 {0.05, 0.1, 0.2},
 {0.01, 0.02, 0.03, 0.04},
 {0.3, 0.2, 0.1}
};

System.out.println("抢劫银行问题测试: ");
for (int i = 0; i < P.length; i++) {
 double p = P[i];
 int[] money = moneyCases[i];
 double[] prob = probCases[i];

 double result1 = rob(p, money, prob);
 double result2 = robOptimized(p, money, prob);
 double result3 = robAlternative(p, money, prob);
 double result4 = robDFS(p, money, prob);

 System.out.printf("P=%.2f, money=%s, prob=%s: 方法 1=%.0f, 方法 2=%.0f, 方法 3=%.0f, 方\n法 4=%.0f%n",
 p, Arrays.toString(money), Arrays.toString(prob),
 result1, result2, result3, result4);

 // 验证结果一致性 (允许小的浮点数误差)
 if (Math.abs(result1 - result2) > 1e-6 ||
 Math.abs(result2 - result3) > 1e-6 ||
 Math.abs(result3 - result4) > 1e-6) {
 System.out.println("警告: 不同方法结果不一致!");
 }
}

// 性能测试 - 大规模数据
int n = 50;
int[] largeMoney = new int[n];
double[] largeProb = new double[n];

// 生成随机测试数据
java.util.Random random = new java.util.Random();
for (int i = 0; i < n; i++) {
 largeMoney[i] = random.nextInt(1000) + 1;
 largeProb[i] = random.nextDouble() * 0.1; // 概率在 0-0.1 之间
}

```

```

long startTime = System.currentTimeMillis();
double largeResult = robOptimized(0.1, largeMoney, largeProb);
long endTime = System.currentTimeMillis();

System.out.printf("大规模测试: 银行数量=%d, 结果=%.0f, 耗时=%dms%n",
n, largeResult, endTime - startTime);

// 边界情况测试
System.out.println("边界情况测试: ");
System.out.println("空数组: " + rob(0.1, new int[] {}, new double[] {}));
System.out.println("P=0: " + rob(0.0, new int[] {10}, new double[] {0.1}));
System.out.println("P=1: " + rob(1.0, new int[] {10, 20}, new double[] {0.1, 0.2}));

// 特殊测试: 概率为 0 的银行
System.out.println("概率为 0 的银行测试: ");
double specialResult = rob(0.01, new int[] {100, 200}, new double[] {0.0, 0.0});
System.out.println("特殊测试结果: " + specialResult);
}

}

/*
* 复杂度分析:
*
* 方法 1: 动态规划 (概率背包)
* - 时间复杂度: O(N * totalMoney)
* - N: 银行数量
* - totalMoney: 总金额
* - 空间复杂度: O(totalMoney)
*
* 方法 2: 优化的动态规划
* - 时间复杂度: O(N * totalMoney) (与方法 1 相同)
* - 空间复杂度: O(totalMoney)
*
* 方法 3: 替代思路的动态规划
* - 时间复杂度: O(N * totalMoney)
* - 空间复杂度: O(totalMoney)
*
* 方法 4: 递归+记忆化搜索
* - 时间复杂度: O(N * totalMoney) (每个状态计算一次)
* - 空间复杂度: O(N * totalMoney) (记忆化数组)
*
* 关键点分析:
* 1. 问题转化: 将概率问题转化为标准的背包问题

```

- \* 2. 精度处理：使用 double 类型处理概率计算
- \* 3. 状态定义： $dp[i]$  表示金额  $i$  对应的最大安全概率
- \* 4. 结果提取：从后向前遍历找到第一个满足条件的金额
- \*
- \* 工程化考量：
  - \* 1. 精度控制：处理浮点数比较和计算
  - \* 2. 异常处理：验证输入参数的合法性
  - \* 3. 性能优化：计算总金额作为背包容量上限
  - \* 4. 测试覆盖：包含正常、边界、性能测试
- \*
- \* 面试要点：
  - \* 1. 理解概率背包问题的转化思路
  - \* 2. 掌握浮点数精度处理技巧
  - \* 3. 了解不同状态定义对算法的影响
  - \* 4. 能够分析算法的时空复杂度
- \*
- \* 扩展应用：
  - \* 1. 风险管理问题
  - \* 2. 投资组合优化
  - \* 3. 资源分配问题
  - \* 4. 多约束优化问题
- \*/

=====

文件：Code48\_Robberies.py

=====

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

HDU 2955 Robberies
题目描述：抢劫犯想抢劫银行，每个银行有一定的金额和被抓的概率。
抢劫犯希望在被抓概率不超过某个值的情况下，抢劫到最多的钱。
链接：https://acm.hdu.edu.cn/showproblem.php?pid=2955
#
解题思路：
这是一个概率背包问题，需要将问题转化为标准的 01 背包问题。
关键点：将金额作为背包容量，将安全概率（1-被抓概率）作为价值。
#
状态定义： $dp[i]$ 表示抢劫到金额 i 时的最大安全概率
状态转移方程： $dp[i] = \max(dp[i], dp[i-money[j]] * (1-p[j]))$
```

```
初始状态: dp[0] = 1 (抢劫 0 元时安全概率为 1)
#
时间复杂度: O(N * totalMoney), 其中 N 是银行数量, totalMoney 是总金额
空间复杂度: O(totalMoney), 使用一维 DP 数组
#
工程化考量:
1. 精度处理: 使用 float 类型处理概率
2. 边界条件: 处理概率为 0 或 1 的情况
3. 性能优化: 计算总金额作为背包容量上限
4. 异常处理: 处理非法输入
```

```
class Robberies:
```

```
 """
 抢劫银行问题的多种解法
 """
```

```
@staticmethod
```

```
def rob(P: float, money: List[int], prob: List[float]) -> float:
 """
 动态规划解法 - 概率背包问题

```

```
Args:
```

```
P: 最大允许被抓概率
```

```
money: 每个银行的金额数组
```

```
prob: 每个银行的被抓概率数组
```

```
Returns:
```

```
float: 在安全概率范围内的最大抢劫金额
 """
```

```
参数验证
```

```
if len(money) != len(prob):
```

```
 raise ValueError("Money and probability arrays must have same size")
```

```
if P < 0 or P > 1:
```

```
 raise ValueError("Probability P must be between 0 and 1")
```

```
n = len(money)
```

```
if n == 0:
```

```
 return 0
```

```
计算总金额作为背包容量上限
```

```
total_money = sum(money)
```

```
创建 DP 数组, dp[i] 表示抢劫到金额 i 时的最大安全概率
```

```

dp = [0.0] * (total_money + 1)
dp[0] = 1.0 # 抢劫 0 元时安全概率为 1

01 背包：遍历每个银行
for i in range(n):
 m = money[i]
 p = prob[i]
 safe_prob = 1 - p # 安全概率

 # 倒序遍历金额，避免重复选择
 for j in range(total_money, m - 1, -1):
 if dp[j - m] > 0:
 dp[j] = max(dp[j], dp[j - m] * safe_prob)

寻找最大的金额，使得安全概率 >= 1-P
min_safe_prob = 1 - P
max_money = 0
for j in range(total_money, -1, -1):
 if dp[j] >= min_safe_prob:
 max_money = j
 break

return max_money

```

```

@staticmethod
def rob_optimized(P: float, money: List[int], prob: List[float]) -> float:
 """
 优化的动态规划解法 - 提前终止遍历
 """

```

Args:

- P: 最大允许被抓概率
- money: 每个银行的金额数组
- prob: 每个银行的被抓概率数组

Returns:

- float: 在安全概率范围内的最大抢劫金额

```

 """
 if len(money) != len(prob):
 raise ValueError("Money and probability arrays must have same size")
 if P < 0 or P > 1:
 raise ValueError("Probability P must be between 0 and 1")

n = len(money)

```

```

if n == 0:
 return 0

计算总金额
total_money = sum(money)

dp = [0.0] * (total_money + 1)
dp[0] = 1.0

for i in range(n):
 m = money[i]
 safe_prob = 1 - prob[i]

 for j in range(total_money, m - 1, -1):
 if dp[j - m] > 0:
 new_prob = dp[j - m] * safe_prob
 if new_prob > dp[j]:
 dp[j] = new_prob

min_safe_prob = 1 - P
从大到小遍历，找到第一个满足条件的金额
for j in range(total_money, -1, -1):
 if dp[j] >= min_safe_prob:
 return j

return 0

```

```

@staticmethod
def rob_alternative(P: float, money: List[int], prob: List[float]) -> float:
 """

```

另一种思路：将金额作为价值，概率作为约束

Args:

- P: 最大允许被抓概率
- money: 每个银行的金额数组
- prob: 每个银行的被抓概率数组

Returns:

- float: 在安全概率范围内的最大抢劫金额

"""

```

if len(money) != len(prob):
 raise ValueError("Money and probability arrays must have same size")
if P < 0 or P > 1:

```

```

raise ValueError("Probability P must be between 0 and 1")

n = len(money)
if n == 0:
 return 0

计算总金额
total_money = sum(money)

dp[i]表示达到金额 i 所需的最小被抓概率
dp = [1.0] * (total_money + 1) # 初始化为最大概率 1
dp[0] = 0.0 # 抢劫 0 元时被抓概率为 0

for i in range(n):
 m = money[i]
 p = prob[i]

 for j in range(total_money, m - 1, -1):
 # 计算选择当前银行的被抓概率
 new_prob = 1 - (1 - dp[j - m]) * (1 - p)
 if new_prob < dp[j]:
 dp[j] = new_prob

寻找最大的金额，使得被抓概率 <= P
for j in range(total_money, -1, -1):
 if dp[j] <= P:
 return j

return 0

@staticmethod
def run_tests():
 """
 运行测试用例，验证所有方法的正确性
 """

 # 测试用例
 P_list = [0.1, 0.05, 0.5]
 money_cases = [
 [10, 20, 30],
 [1, 2, 3, 4],
 [100, 200, 300]
]
 prob_cases = [

```

```

[0.05, 0.1, 0.2],
[0.01, 0.02, 0.03, 0.04],
[0.3, 0.2, 0.1]
]

print("抢劫银行问题测试: ")
for i, (P, money, prob) in enumerate(zip(P_list, money_cases, prob_cases)):
 try:
 result1 = Robberies.rob(P, money, prob)
 result2 = Robberies.rob_optimized(P, money, prob)
 result3 = Robberies.rob_alternative(P, money, prob)

 print(f"P={P}, money={money}, prob={prob}: "
 f"方法 1={result1}, 方法 2={result2}, 方法 3={result3}")

 # 验证结果一致性 (允许小的浮点数误差)
 if abs(result1 - result2) > 1e-6 or abs(result2 - result3) > 1e-6:
 print("警告: 不同方法结果不一致!")

 except Exception as e:
 print(f"测试用例{i+1}时发生错误: {e}")

性能测试 - 大规模数据
n = 50
large_money = [random.randint(1, 1000) for _ in range(n)]
large_prob = [random.uniform(0, 0.1) for _ in range(n)]

start_time = time.time()
large_result = Robberies.rob_optimized(0.1, large_money, large_prob)
end_time = time.time()

print(f"大规模测试: 银行数量={n}, 结果={large_result}, "
 f"耗时={end_time - start_time:.4f}秒")

边界情况测试
print("边界情况测试: ")
print(f"空数组: {Robberies.rob(0.1, [], [])}")
print(f"P=0: {Robberies.rob(0.0, [10], [0.1])}")
print(f"P=1: {Robberies.rob(1.0, [10, 20], [0.1, 0.2])}")

特殊测试: 概率为 0 的银行
print("概率为 0 的银行测试: ")
special_result = Robberies.rob(0.01, [100, 200], [0.0, 0.0])

```

```
print(f"特殊测试结果: {special_result}")

def main():
 """
 主函数 - 运行测试和演示
 """
 try:
 Robberies.run_tests()
 except Exception as e:
 print(f"程序执行错误: {e}")
 return 1
 return 0

if __name__ == "__main__":
 sys.exit(main())

"""


```

复杂度分析:

方法 1: 动态规划 (概率背包)

- 时间复杂度:  $O(N * \text{totalMoney})$ 
  - N: 银行数量
  - totalMoney: 总金额
- 空间复杂度:  $O(\text{totalMoney})$

方法 2: 优化的动态规划

- 时间复杂度:  $O(N * \text{totalMoney})$  (与方法 1 相同)
- 空间复杂度:  $O(\text{totalMoney})$

方法 3: 替代思路的动态规划

- 时间复杂度:  $O(N * \text{totalMoney})$
- 空间复杂度:  $O(\text{totalMoney})$

Python 特定优化:

1. 使用列表推导式和生成器表达式
2. 利用 Python 的动态类型特性
3. 使用类型注解提高代码可读性
4. 使用 random 模块进行性能测试

关键点分析:

1. 问题转化: 将概率问题转化为标准的背包问题
2. 精度处理: 使用 float 类型处理概率计算
3. 状态定义:  $dp[i]$  表示金额  $i$  对应的最大安全概率

4. 结果提取：从后向前遍历找到第一个满足条件的金额

工程化考量：

1. 模块化设计：将不同解法封装为静态方法
2. 异常处理：完善的参数验证和错误处理
3. 性能监控：包含性能测试和时间测量
4. 测试覆盖：包含各种边界情况和性能测试

面试要点：

1. 理解概率背包问题的转化思路
2. 掌握浮点数精度处理技巧
3. 了解不同状态定义对算法的影响
4. 能够分析算法的时空复杂度

扩展应用：

1. 风险管理问题
2. 投资组合优化
3. 资源分配问题
4. 多约束优化问题

"""

文件：Code49\_Knapsack2.cpp

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <climits>
#include <random>
#include <chrono>

using namespace std;

// AtCoder DP Contest E - Knapsack 2
// 题目描述：经典的01背包问题，但是背包容量非常大(10^9)，而物品价值比较小(10^3)。
// 链接：https://atcoder.jp/contests/dp/tasks/dp_e
//
// 解题思路：
// 当背包容量非常大时，传统的DP方法会超时或超内存。
// 需要转换思路：将价值作为背包容量，求达到某个价值所需的最小重量。
//
// 状态定义：dp[i] 表示达到价值 i 所需的最小重量
```

```

// 状态转移方程: dp[i] = min(dp[i], dp[i-value[j]] + weight[j])
// 初始状态: dp[0] = 0, 其他为无穷大
//
// 时间复杂度: O(N * totalValue), 其中 N 是物品数量, totalValue 是总价值
// 空间复杂度: O(totalValue)
//
// 工程化考量:
// 1. 问题转化: 从重量维度转为价值维度
// 2. 边界处理: 处理无穷大值和结果提取
// 3. 性能优化: 计算总价值作为新背包容量
// 4. 异常处理: 处理空输入和边界值

class Knapsack2 {
private:
 static const long long INF = LLONG_MAX / 2; // 表示不可达状态

public:
 /**
 * 动态规划解法 - 价值维度 DP
 * @param W 背包容量
 * @param weights 物品重量数组
 * @param values 物品价值数组
 * @return 能装入背包的最大价值
 */
 static long long knapsack2(long long W, vector<int>& weights, vector<int>& values) {
 // 参数验证
 if (weights.size() != values.size()) {
 throw invalid_argument("Weights and values arrays must have same size");
 }
 if (W < 0) {
 throw invalid_argument("Capacity W must be non-negative");
 }

 int n = weights.size();
 if (n == 0) {
 return 0;
 }

 // 计算总价值
 int totalValue = 0;
 for (int value : values) {
 totalValue += value;
 }
 }
}

```

```

// 创建 DP 数组，dp[i] 表示达到价值 i 所需的最小重量
vector<long long> dp(totalValue + 1, INF);
dp[0] = 0; // 价值为 0 时重量为 0

// 遍历每个物品
for (int i = 0; i < n; i++) {
 int weight = weights[i];
 int value = values[i];

 // 倒序遍历价值，避免重复选择
 for (int j = totalValue; j >= value; j--) {
 if (dp[j - value] != INF) {
 dp[j] = min(dp[j], dp[j - value] + weight);
 }
 }
}

// 寻找最大的价值，使得所需重量 <= W
long long maxValue = 0;
for (int j = totalValue; j >= 0; j--) {
 if (dp[j] <= W) {
 maxValue = j;
 break;
 }
}

return maxValue;
}

/**
 * 优化的动态规划解法 - 提前终止
 */
static long long knapsack2Optimized(long long W, vector<int>& weights, vector<int>& values) {
 if (weights.size() != values.size()) {
 throw invalid_argument("Weights and values arrays must have same size");
 }

 if (W < 0) {
 throw invalid_argument("Capacity W must be non-negative");
 }

 int n = weights.size();
 if (n == 0) {

```

```

 return 0;
}

// 计算总价值
int totalValue = 0;
for (int value : values) {
 totalValue += value;
}

vector<long long> dp(totalValue + 1, INF);
dp[0] = 0;

for (int i = 0; i < n; i++) {
 int weight = weights[i];
 int value = values[i];

 for (int j = totalValue; j >= value; j--) {
 if (dp[j - value] != INF) {
 long long newWeight = dp[j - value] + weight;
 if (newWeight < dp[j]) {
 dp[j] = newWeight;
 }
 }
 }
}

// 从大到小遍历，找到第一个满足条件的价值
for (int j = totalValue; j >= 0; j--) {
 if (dp[j] <= W) {
 return j;
 }
}

return 0;
}

/***
 * 传统 DP 解法（用于对比） - 仅适用于小容量
 */
static long long knapsackTraditional(long long W, vector<int>& weights, vector<int>& values)
{
 if (weights.size() != values.size()) {
 throw invalid_argument("Weights and values arrays must have same size");
 }
}

```

```

}

if (W < 0) {
 throw invalid_argument("Capacity W must be non-negative");
}

int n = weights.size();
if (n == 0) {
 return 0;
}

// 传统DP: dp[i]表示容量为 i 时的最大价值
int maxW = min(W, (long long)INT_MAX);
vector<long long> dp(maxW + 1, 0);

for (int i = 0; i < n; i++) {
 int weight = weights[i];
 int value = values[i];

 for (int j = maxW; j >= weight; j--) {
 dp[j] = max(dp[j], dp[j - weight] + value);
 }
}

return dp[maxW];
}

};

/***
 * 测试函数
 */
void testKnapsack2() {
 // 测试用例
 vector<long long> W = {100, 1000, 1000000000LL};
 vector<vector<int>> weightsCases = {
 {10, 20, 30},
 {50, 100, 150},
 {1, 2, 3}
 };
 vector<vector<int>> valuesCases = {
 {60, 100, 120},
 {60, 100, 120},
 {10, 15, 40}
 };
}

```

```

cout << "大容量背包问题测试: " << endl;
for (size_t i = 0; i < W.size(); i++) {
 long long w = W[i];
 vector<int> weights = weightsCases[i];
 vector<int> values = valuesCases[i];

 long long result1 = Knapsack2::knapsack2(w, weights, values);
 long long result2 = Knapsack2::knapsack2Optimized(w, weights, values);

 // 对于小容量，可以用传统方法验证
 long long traditionalResult = 0;
 if (w <= 10000) {
 traditionalResult = Knapsack2::knapsackTraditional(w, weights, values);
 }

 cout << "W=" << w << ", weights=[";
 for (size_t j = 0; j < weights.size(); j++) {
 cout << weights[j];
 if (j < weights.size() - 1) cout << ", ";
 }
 cout << "], values=[";
 for (size_t j = 0; j < values.size(); j++) {
 cout << values[j];
 if (j < values.size() - 1) cout << ", ";
 }
 cout << "]: 方法1=" << result1 << ", 方法2=" << result2;

 if (w <= 10000) {
 cout << ", 传统方法=" << traditionalResult;
 // 验证结果一致性
 if (result1 != traditionalResult || result2 != traditionalResult) {
 cout << "- 警告: 结果不一致!" << endl;
 } else {
 cout << "- 验证通过" << endl;
 }
 } else {
 cout << "- 大容量测试" << endl;
 }
}

// 性能测试 - 大规模数据
int n = 100;

```

```

vector<int> largeWeights(n);
vector<int> largeValues(n);

// 生成随机测试数据
random_device rd;
mt19937 gen(rd());
uniform_int_distribution<> weightDist(1, 1000);
uniform_int_distribution<> valueDist(1, 100);

for (int i = 0; i < n; i++) {
 largeWeights[i] = weightDist(gen);
 largeValues[i] = valueDist(gen);
}

long long largeW = 1000000000LL; // 10^9

auto startTime = chrono::high_resolution_clock::now();
long long largeResult = Knapsack2::knapsack2Optimized(largeW, largeWeights, largeValues);
auto endTime = chrono::high_resolution_clock::now();
auto duration = chrono::duration_cast<chrono::milliseconds>(endTime - startTime);

cout << "大规模测试: 物品数量=" << n << ", 容量=" << largeW
 << ", 结果=" << largeResult << ", 耗时=" << duration.count() << "ms" << endl;

// 边界情况测试
cout << "边界情况测试: " << endl;
vector<int> emptyWeights;
vector<int> emptyValues;
cout << "空数组: " << Knapsack2::knapsack2(100, emptyWeights, emptyValues) << endl;

vector<int> singleWeight = {10};
vector<int> singleValue = {5};
cout << "W=0: " << Knapsack2::knapsack2(0, singleWeight, singleValue) << endl;

vector<int> heavyWeights = {10, 20};
vector<int> heavyValues = {5, 10};
cout << "所有物品超重: " << Knapsack2::knapsack2(5, heavyWeights, heavyValues) << endl;

// 特殊测试: 价值为 0 的物品
cout << "价值为 0 的物品测试: " << endl;
vector<int> zeroValueWeights = {10, 20};
vector<int> zeroValueValues = {0, 0};
long long specialResult = Knapsack2::knapsack2(100, zeroValueWeights, zeroValueValues);

```

```
cout << "特殊测试结果: " << specialResult << endl;
}

int main() {
 try {
 testKnapsack2();
 } catch (const exception& e) {
 cerr << "错误: " << e.what() << endl;
 return 1;
 }
 return 0;
}

/*
 * 复杂度分析:
 *
 * 方法 1: 价值维度 DP
 * - 时间复杂度: O(N * totalValue)
 * - N: 物品数量
 * - totalValue: 总价值
 * - 空间复杂度: O(totalValue)
 *
 * 方法 2: 优化的价值维度 DP
 * - 时间复杂度: O(N * totalValue) (与方法 1 相同)
 * - 空间复杂度: O(totalValue)
 *
 * 方法 3: 传统重量维度 DP
 * - 时间复杂度: O(N * W)
 * - 空间复杂度: O(W)
 *
 * C++特定优化:
 * 1. 使用 vector 代替数组, 更安全
 * 2. 使用 STL 算法进行最小值和最大值计算
 * 3. 使用随机数生成器进行性能测试
 * 4. 使用 chrono 进行精确性能测量
 *
 * 关键点分析:
 * 1. 问题转化: 当 W 很大时, 从重量维度转为价值维度
 * 2. 状态定义: dp[i] 表示达到价值 i 所需的最小重量
 * 3. 结果提取: 从后向前遍历找到第一个满足重量约束的价值
 * 4. 适用场景: W 很大但总价值不大的情况
 *
 * 工程化考量:
```

- \* 1. 方法选择：根据  $W$  的大小选择合适的算法
- \* 2. 内存优化：使用 vector 的动态分配
- \* 3. 边界处理：处理各种极端情况
- \* 4. 性能测试：包含大规模数据测试

\*

- \* 面试要点：
- \* 1. 理解传统 DP 的局限性
- \* 2. 掌握问题转化的思路
- \* 3. 了解不同维度 DP 的适用场景
- \* 4. 能够分析算法的时空复杂度

\*/

=====

文件：Code49\_Knapsack2.java

=====

```
package class073;

import java.util.Arrays;

// AtCoder DP Contest E - Knapsack 2
// 题目描述：经典的 01 背包问题，但是背包容量非常大 (10^9)，而物品价值比较小 (10^3)。
// 链接：https://atcoder.jp/contests/dp/tasks/dp_e
//
// 解题思路：
// 当背包容量非常大时，传统的 DP 方法会超时或超内存。
// 需要转换思路：将价值作为背包容量，求达到某个价值所需的最小重量。
//
// 状态定义：dp[i] 表示达到价值 i 所需的最小重量
// 状态转移方程：dp[i] = min(dp[i], dp[i-value[j]] + weight[j])
// 初始状态：dp[0] = 0，其他为无穷大
//
// 时间复杂度：O(N * totalValue)，其中 N 是物品数量，totalValue 是总价值
// 空间复杂度：O(totalValue)
//
// 工程化考量：
// 1. 问题转化：从重量维度转为价值维度
// 2. 边界处理：处理无穷大值和结果提取
// 3. 性能优化：计算总价值作为新背包容量
// 4. 异常处理：处理空输入和边界值
```

```
public class Code49_Knapsack2 {
```

```

private static final long INF = Long.MAX_VALUE / 2; // 表示不可达状态

/**
 * 动态规划解法 - 价值维度 DP
 * @param W 背包容量
 * @param weights 物品重量数组
 * @param values 物品价值数组
 * @return 能装入背包的最大价值
*/
public static long knapsack2(long W, int[] weights, int[] values) {
 // 参数验证
 if (weights == null || values == null || weights.length != values.length) {
 throw new IllegalArgumentException("Invalid input parameters");
 }
 if (W < 0) {
 throw new IllegalArgumentException("Capacity W must be non-negative");
 }

 int n = weights.length;
 if (n == 0) {
 return 0;
 }

 // 计算总价值
 int totalValue = 0;
 for (int value : values) {
 totalValue += value;
 }

 // 创建 DP 数组, dp[i] 表示达到价值 i 所需的最小重量
 long[] dp = new long[totalValue + 1];
 Arrays.fill(dp, INF);
 dp[0] = 0; // 价值为 0 时重量为 0

 // 遍历每个物品
 for (int i = 0; i < n; i++) {
 int weight = weights[i];
 int value = values[i];

 // 倒序遍历价值, 避免重复选择
 for (int j = totalValue; j >= value; j--) {
 if (dp[j - value] != INF) {
 dp[j] = Math.min(dp[j], dp[j - value] + weight);
 }
 }
 }
}

```

```

 }
 }
}

// 寻找最大的价值，使得所需重量 <= W
long maxValue = 0;
for (int j = totalValue; j >= 0; j--) {
 if (dp[j] <= W) {
 maxValue = j;
 break;
 }
}

return maxValue;
}

/**
 * 优化的动态规划解法 - 提前终止
 */
public static long knapsack2Optimized(long W, int[] weights, int[] values) {
 if (weights == null || values == null || weights.length != values.length) {
 throw new IllegalArgumentException("Invalid input parameters");
 }

 if (W < 0) {
 throw new IllegalArgumentException("Capacity W must be non-negative");
 }

 int n = weights.length;
 if (n == 0) {
 return 0;
 }

 // 计算总价值
 int totalValue = 0;
 for (int value : values) {
 totalValue += value;
 }

 long[] dp = new long[totalValue + 1];
 Arrays.fill(dp, INF);
 dp[0] = 0;

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

```

```

int weight = weights[i];
int value = values[i];

for (int j = totalValue; j >= value; j--) {
 if (dp[j - value] != INF) {
 long newWeight = dp[j - value] + weight;
 if (newWeight < dp[j]) {
 dp[j] = newWeight;
 }
 }
}

// 从大到小遍历，找到第一个满足条件的价值
for (int j = totalValue; j >= 0; j--) {
 if (dp[j] <= W) {
 return j;
 }
}

return 0;
}

/**
 * 空间优化的解法 - 使用滚动数组思想
 */
public static long knapsack2SpaceOptimized(long W, int[] weights, int[] values) {
 if (weights == null || values == null || weights.length != values.length) {
 throw new IllegalArgumentException("Invalid input parameters");
 }

 if (W < 0) {
 throw new IllegalArgumentException("Capacity W must be non-negative");
 }

 int n = weights.length;
 if (n == 0) {
 return 0;
 }

 // 计算总价值
 int totalValue = 0;
 for (int value : values) {
 totalValue += value;
 }
}

```

```

}

// 使用两个数组交替，优化空间
long[] dp = new long[totalValue + 1];
Arrays.fill(dp, INF);
dp[0] = 0;

for (int i = 0; i < n; i++) {
 int weight = weights[i];
 int value = values[i];

 // 创建临时数组用于更新
 long[] temp = dp.clone();

 for (int j = value; j <= totalValue; j++) {
 if (temp[j - value] != INF) {
 dp[j] = Math.min(dp[j], temp[j - value] + weight);
 }
 }
}

// 寻找最大价值
for (int j = totalValue; j >= 0; j--) {
 if (dp[j] <= W) {
 return j;
 }
}

return 0;
}

/**
 * 传统DP解法（用于对比）- 仅适用于小容量
 */
public static long knapsackTraditional(long W, int[] weights, int[] values) {
 if (weights == null || values == null || weights.length != values.length) {
 throw new IllegalArgumentException("Invalid input parameters");
 }

 if (W < 0) {
 throw new IllegalArgumentException("Capacity W must be non-negative");
 }

 int n = weights.length;

```

```
if (n == 0) {
 return 0;
}

// 传统DP: dp[i]表示容量为 i 时的最大价值
int maxW = (int) Math.min(W, Integer.MAX_VALUE);
long[] dp = new long[maxW + 1];

for (int i = 0; i < n; i++) {
 int weight = weights[i];
 int value = values[i];

 for (int j = maxW; j >= weight; j--) {
 dp[j] = Math.max(dp[j], dp[j - weight] + value);
 }
}

return dp[maxW];
}

/***
 * 测试方法
 */
public static void main(String[] args) {
 // 测试用例
 long[] W = {100, 1000, 1000000000L};
 int[][] weightsCases = {
 {10, 20, 30},
 {50, 100, 150},
 {1, 2, 3}
 };
 int[][] valuesCases = {
 {60, 100, 120},
 {60, 100, 120},
 {10, 15, 40}
 };

 System.out.println("大容量背包问题测试: ");
 for (int i = 0; i < W.length; i++) {
 long w = W[i];
 int[] weights = weightsCases[i];
 int[] values = valuesCases[i];
 }
}
```

```

long result1 = knapsack2(w, weights, values);
long result2 = knapsack2Optimized(w, weights, values);
long result3 = knapsack2SpaceOptimized(w, weights, values);

// 对于小容量，可以用传统方法验证
long traditionalResult = 0;
if (w <= 10000) {
 traditionalResult = knapsackTraditional(w, weights, values);
}

System.out.printf("W=%d, weights=%s, values=%s: 方法 1=%d, 方法 2=%d, 方法 3=%d",
 w, Arrays.toString(weights), Arrays.toString(values),
 result1, result2, result3);

if (w <= 10000) {
 System.out.printf(", 传统方法=%d", traditionalResult);
 // 验证结果一致性
 if (result1 != traditionalResult || result2 != traditionalResult ||
 result3 != traditionalResult) {
 System.out.println(" - 警告: 结果不一致!");
 } else {
 System.out.println(" - 验证通过");
 }
} else {
 System.out.println(" - 大容量测试");
}
}

// 性能测试 - 大规模数据
int n = 100;
int[] largeWeights = new int[n];
int[] largeValues = new int[n];

// 生成随机测试数据
java.util.Random random = new java.util.Random();
for (int i = 0; i < n; i++) {
 largeWeights[i] = random.nextInt(1000) + 1;
 largeValues[i] = random.nextInt(100) + 1;
}

long largeW = 1000000000L; // 10^9

long startTime = System.currentTimeMillis();

```

```

 long largeResult = knapsack2Optimized(largeW, largeWeights, largeValues);
 long endTime = System.currentTimeMillis();

 System.out.printf("大规模测试：物品数量=%d, 容量=%d, 结果=%d, 耗时=%dms%n",
 n, largeW, largeResult, endTime - startTime);

 // 边界情况测试
 System.out.println("边界情况测试：");
 System.out.println("空数组：" + knapsack2(100, new int[] {}, new int[] {}));
 System.out.println("W=0：" + knapsack2(0, new int[] {10}, new int[] {5}));
 System.out.println("所有物品超重：" + knapsack2(5, new int[] {10, 20}, new int[] {5, 10}));

 // 特殊测试：价值为 0 的物品
 System.out.println("价值为 0 的物品测试：");
 long specialResult = knapsack2(100, new int[] {10, 20}, new int[] {0, 0});
 System.out.println("特殊测试结果：" + specialResult);
 }

}

/*
 * 复杂度分析：
 *
 * 方法 1：价值维度 DP
 * - 时间复杂度：O(N * totalValue)
 * - N：物品数量
 * - totalValue：总价值
 * - 空间复杂度：O(totalValue)
 *
 * 方法 2：优化的价值维度 DP
 * - 时间复杂度：O(N * totalValue)（与方法 1 相同）
 * - 空间复杂度：O(totalValue)
 *
 * 方法 3：空间优化的 DP
 * - 时间复杂度：O(N * totalValue)
 * - 空间复杂度：O(totalValue)（但使用临时数组）
 *
 * 方法 4：传统重量维度 DP
 * - 时间复杂度：O(N * W)
 * - 空间复杂度：O(W)
 *
 * 关键点分析：
 * 1. 问题转化：当 W 很大时，从重量维度转为价值维度
 * 2. 状态定义：dp[i] 表示达到价值 i 所需的最小重量

```

- \* 3. 结果提取：从后向前遍历找到第一个满足重量约束的价值
- \* 4. 适用场景：W 很大但总价值不大的情况
- \*
- \* 工程化考量：
  - \* 1. 方法选择：根据 W 的大小选择合适的算法
  - \* 2. 内存优化：使用滚动数组思想
  - \* 3. 边界处理：处理各种极端情况
  - \* 4. 性能测试：包含大规模数据测试
- \*
- \* 面试要点：
  - \* 1. 理解传统 DP 的局限性
  - \* 2. 掌握问题转化的思路
  - \* 3. 了解不同维度 DP 的适用场景
  - \* 4. 能够分析算法的时空复杂度
- \*
- \* 扩展应用：
  - \* 1. 资源分配问题
  - \* 2. 投资组合优化
  - \* 3. 多约束优化问题
  - \* 4. 大规模数据处理
- \*/

---

文件：Code49\_Knapsack2.py

---

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

AtCoder DP Contest E - Knapsack 2
题目描述：经典的 01 背包问题，但是背包容量非常大 (10^9)，而物品价值比较小 (10^3)。
链接：https://atcoder.jp/contests/dp/tasks/dp_e
#
解题思路：
当背包容量非常大时，传统的 DP 方法会超时或超内存。
需要转换思路：将价值作为背包容量，求达到某个价值所需的最小重量。
#
状态定义：dp[i] 表示达到价值 i 所需的最小重量
状态转移方程：dp[i] = min(dp[i], dp[i-value[j]] + weight[j])
初始状态：dp[0] = 0，其他为无穷大
#
```

```

时间复杂度: O(N * totalValue), 其中 N 是物品数量, totalValue 是总价值
空间复杂度: O(totalValue)

#
工程化考量:
1. 问题转化: 从重量维度转为价值维度
2. 边界处理: 处理无穷大值和结果提取
3. 性能优化: 计算总价值作为新背包容量
4. 异常处理: 处理空输入和边界值

class Knapsack2:

 """
 大容量背包问题的多种解法
 """

 INF = 10**18 # 表示不可达状态

 @staticmethod
 def knapsack2(W: int, weights: List[int], values: List[int]) -> int:
 """
 动态规划解法 - 价值维度 DP

 Args:
 W: 背包容量
 weights: 物品重量数组
 values: 物品价值数组

 Returns:
 int: 能装入背包的最大价值
 """

 # 参数验证
 if len(weights) != len(values):
 raise ValueError("Weights and values arrays must have same size")
 if W < 0:
 raise ValueError("Capacity W must be non-negative")

 n = len(weights)
 if n == 0:
 return 0

 # 计算总价值
 total_value = sum(values)

 # 创建 DP 数组, dp[i] 表示达到价值 i 所需的最小重量

```

```

dp = [Knapsack2.INF] * (total_value + 1)
dp[0] = 0 # 价值为0时重量为0

遍历每个物品
for i in range(n):
 weight = weights[i]
 value = values[i]

 # 倒序遍历价值，避免重复选择
 for j in range(total_value, value - 1, -1):
 if dp[j - value] != Knapsack2.INF:
 dp[j] = min(dp[j], dp[j - value] + weight)

寻找最大的价值，使得所需重量 <= W
max_value = 0
for j in range(total_value, -1, -1):
 if dp[j] <= W:
 max_value = j
 break

return max_value

```

```

@staticmethod
def knapsack2_optimized(W: int, weights: List[int], values: List[int]) -> int:
 """
 优化的动态规划解法 - 提前终止

```

Args:

- W: 背包容量
- weights: 物品重量数组
- values: 物品价值数组

Returns:

- int: 能装入背包的最大价值

"""

```

if len(weights) != len(values):
 raise ValueError("Weights and values arrays must have same size")
if W < 0:
 raise ValueError("Capacity W must be non-negative")

n = len(weights)
if n == 0:
 return 0

```

```

计算总价值
total_value = sum(values)

dp = [Knapsack2.INF] * (total_value + 1)
dp[0] = 0

for i in range(n):
 weight = weights[i]
 value = values[i]

 for j in range(total_value, value - 1, -1):
 if dp[j - value] != Knapsack2.INF:
 new_weight = dp[j - value] + weight
 if new_weight < dp[j]:
 dp[j] = new_weight

从大到小遍历，找到第一个满足条件的价值
for j in range(total_value, -1, -1):
 if dp[j] <= W:
 return j

return 0

```

```

@staticmethod
def knapsack_traditional(W: int, weights: List[int], values: List[int]) -> int:
 """
 传统 DP 解法（用于对比） - 仅适用于小容量
 """

```

Args:

- W: 背包容量
- weights: 物品重量数组
- values: 物品价值数组

Returns:

- int: 能装入背包的最大价值

"""

```

if len(weights) != len(values):
 raise ValueError("Weights and values arrays must have same size")
if W < 0:
 raise ValueError("Capacity W must be non-negative")

```

n = len(weights)

```

if n == 0:
 return 0

传统DP: dp[i]表示容量为 i 时的最大价值
max_w = min(W, 10**6) # 限制最大容量, 避免内存溢出
dp = [0] * (max_w + 1)

for i in range(n):
 weight = weights[i]
 value = values[i]

 for j in range(max_w, weight - 1, -1):
 dp[j] = max(dp[j], dp[j - weight] + value)

return dp[max_w]

@staticmethod
def run_tests():
 """
 运行测试用例, 验证所有方法的正确性
 """

 # 测试用例
 W_list = [100, 1000, 10**9]
 weights_cases = [
 [10, 20, 30],
 [50, 100, 150],
 [1, 2, 3]
]
 values_cases = [
 [60, 100, 120],
 [60, 100, 120],
 [10, 15, 40]
]

 print("大容量背包问题测试: ")
 for i, (W, weights, values) in enumerate(zip(W_list, weights_cases, values_cases)):
 try:
 result1 = Knapsack2.knapsack2(W, weights, values)
 result2 = Knapsack2.knapsack2_optimized(W, weights, values)

 # 对于小容量, 可以用传统方法验证
 traditional_result = 0
 if W <= 10000:

```

```

 traditional_result = Knapsack2.knapsack_traditional(W, weights, values)

 print(f"W={W}, weights={weights}, values={values}: "
 f"方法 1={result1}, 方法 2={result2}", end="")

 if W <= 10000:
 print(f"，传统方法={traditional_result}", end="")
 # 验证结果一致性
 if result1 != traditional_result or result2 != traditional_result:
 print(" - 警告：结果不一致！")
 else:
 print(" - 验证通过")
 else:
 print(" - 大容量测试")

except Exception as e:
 print(f"测试用例{i+1}时发生错误: {e}")

性能测试 - 大规模数据
n = 100
large_weights = [random.randint(1, 1000) for _ in range(n)]
large_values = [random.randint(1, 100) for _ in range(n)]
large_W = 10**9 # 10^9

start_time = time.time()
large_result = Knapsack2.knapsack2_optimized(large_W, large_weights, large_values)
end_time = time.time()

print(f"大规模测试: 物品数量={n}, 容量={large_W}, "
 f"结果={large_result}, 耗时={end_time - start_time:.4f}秒")

边界情况测试
print("边界情况测试: ")
print(f"空数组: {Knapsack2.knapsack2(100, [], [])}")
print(f"W=0: {Knapsack2.knapsack2(0, [10], [5])}")
print(f"所有物品超重: {Knapsack2.knapsack2(5, [10, 20], [5, 10])}")

特殊测试: 价值为 0 的物品
print("价值为 0 的物品测试: ")
special_result = Knapsack2.knapsack2(100, [10, 20], [0, 0])
print(f"特殊测试结果: {special_result}")

def main():

```

```
"""
主函数 - 运行测试和演示
"""

try:
 Knapsack2.run_tests()
except Exception as e:
 print(f"程序执行错误: {e}")
 return 1
return 0

if __name__ == "__main__":
 sys.exit(main())

"""


```

复杂度分析:

方法 1: 价值维度 DP

- 时间复杂度:  $O(N * \text{totalValue})$ 
  - N: 物品数量
  - totalValue: 总价值
- 空间复杂度:  $O(\text{totalValue})$

方法 2: 优化的价值维度 DP

- 时间复杂度:  $O(N * \text{totalValue})$  (与方法 1 相同)
- 空间复杂度:  $O(\text{totalValue})$

方法 3: 传统重量维度 DP

- 时间复杂度:  $O(N * W)$
- 空间复杂度:  $O(W)$

Python 特定优化:

1. 使用列表推导式和生成器表达式
2. 利用 Python 的动态类型特性
3. 使用类型注解提高代码可读性
4. 使用 random 模块进行性能测试

关键点分析:

1. 问题转化: 当  $W$  很大时, 从重量维度转为价值维度
2. 状态定义:  $dp[i]$  表示达到价值  $i$  所需的最小重量
3. 结果提取: 从后向前遍历找到第一个满足重量约束的价值
4. 适用场景:  $W$  很大但总价值不大的情况

工程化考量:

1. 方法选择：根据 W 的大小选择合适的算法
2. 内存优化：限制传统方法的最大容量
3. 边界处理：处理各种极端情况
4. 性能测试：包含大规模数据测试

面试要点：

1. 理解传统 DP 的局限性
2. 掌握问题转化的思路
3. 了解不同维度 DP 的适用场景
4. 能够分析算法的时空复杂度

扩展应用：

1. 资源分配问题
2. 投资组合优化
3. 多约束优化问题
4. 大规模数据处理

=====

文件：Code50\_DivisibleGroupSums.cpp

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <random>
#include <chrono>

using namespace std;

// UVA 10616 Divisible Group Sums
// 题目描述：给定 N 个整数，选择 M 个数字使得它们的和能被 D 整除，求方案数。
// 链接：
https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=1557
//
// 解题思路：
// 这是一个分组背包+模数运算的问题，需要计算选择 M 个数字的和能被 D 整除的方案数。
//
// 状态定义：dp[i][j][k] 表示前 i 个数字，选择 j 个数字，和对 D 取模为 k 的方案数
// 状态转移方程：
// dp[i][j][k] = dp[i-1][j][k] + dp[i-1][j-1][(k - num[i] % D + D) % D]
//
// 关键点：
```

```

// 1. 模数运算: 处理负数取模, 使用(k - num[i] % D + D) % D
// 2. 分组背包: 每个数字只能选择一次
// 3. 空间优化: 使用滚动数组优化空间复杂度
//
// 时间复杂度: O(N * M * D), 其中 N 是数字数量, M 是选择数量, D 是除数
// 空间复杂度: O(M * D), 使用滚动数组优化
//
// 工程化考量:
// 1. 模数处理: 正确处理负数取模
// 2. 边界条件: 处理 M=0、D=0 等特殊情况
// 3. 性能优化: 使用滚动数组和模数运算优化
// 4. 异常处理: 处理除数为 0 的情况

class DivisibleGroupSums {
public:
 /**
 * 动态规划解法 - 分组背包+模数运算
 * @param nums 整数数组
 * @param M 需要选择的数字个数
 * @param D 除数
 * @return 方案数
 */
 static long long divisibleGroupSums(vector<int>& nums, int M, int D) {
 // 参数验证
 if (nums.empty()) {
 return M == 0 ? 1 : 0;
 }
 if (D == 0) {
 throw invalid_argument("Divisor D cannot be zero");
 }
 if (M < 0 || M > nums.size()) {
 return 0;
 }

 int n = nums.size();

 // 创建 DP 数组, 使用滚动数组优化
 vector<vector<long long>> dp(M + 1, vector<long long>(D, 0));
 dp[0][0] = 1; // 选择 0 个数字, 和为 0 (模 D 为 0) 的方案数为 1

 // 遍历每个数字
 for (int i = 0; i < n; i++) {
 int num = nums[i];

```

```

int mod = (num % D + D) % D; // 处理负数取模

// 倒序遍历选择数量，避免重复选择
for (int j = M; j >= 1; j--) {
 // 创建临时数组保存当前状态
 vector<long long> temp = dp[j];
 for (int k = 0; k < D; k++) {
 int prevMod = (k - mod + D) % D;
 dp[j][k] += dp[j - 1][prevMod];
 }
}

return dp[M][0];
}

/***
 * 优化的动态规划解法 - 二维数组
 */
static long long divisibleGroupSums2D(vector<int>& nums, int M, int D) {
 if (nums.empty()) {
 return M == 0 ? 1 : 0;
 }

 if (D == 0) {
 throw invalid_argument("Divisor D cannot be zero");
 }

 if (M < 0 || M > nums.size()) {
 return 0;
 }

 int n = nums.size();

 // 创建三维 DP 数组
 vector<vector<vector<long long>>> dp(n + 1,
 vector<vector<long long>>(M + 1,
 vector<long long>(D, 0)));
 dp[0][0][0] = 1;

 for (int i = 1; i <= n; i++) {
 int num = nums[i - 1];
 int mod = (num % D + D) % D;

 for (int j = 0; j <= M; j++) {

```

```

 for (int k = 0; k < D; k++) {
 // 不选当前数字
 dp[i][j][k] += dp[i - 1][j][k];

 // 选当前数字
 if (j >= 1) {
 int prevMod = (k - mod + D) % D;
 dp[i][j][k] += dp[i - 1][j - 1][prevMod];
 }
 }
 }

 return dp[n][M][0];
}

/***
 * 空间优化的解法 - 使用两个二维数组交替
 */
static long long divisibleGroupSumsOptimized(vector<int>& nums, int M, int D) {
 if (nums.empty()) {
 return M == 0 ? 1 : 0;
 }

 if (D == 0) {
 throw invalid_argument("Divisor D cannot be zero");
 }

 if (M < 0 || M > nums.size()) {
 return 0;
 }

 int n = nums.size();

 // 使用两个二维数组交替
 vector<vector<long long>> dp(M + 1, vector<long long>(D, 0));
 vector<vector<long long>> next(M + 1, vector<long long>(D, 0));
 dp[0][0] = 1;

 for (int i = 0; i < n; i++) {
 int num = nums[i];
 int mod = (num % D + D) % D;

 // 复制当前状态到 next
 for (int j = 0; j <= M; j++) {

```

```

 next[j] = dp[j];
 }

 // 更新 next 数组
 for (int j = 1; j <= M; j++) {
 for (int k = 0; k < D; k++) {
 int prevMod = (k - mod + D) % D;
 next[j][k] += dp[j - 1][prevMod];
 }
 }

 // 交换数组
 swap(dp, next);

 // 清空 next 数组用于下一次迭代
 for (int j = 0; j <= M; j++) {
 fill(next[j].begin(), next[j].end(), 0);
 }

 return dp[M][0];
}

};

/***
 * 计算组合数 C(n, k)
 */
long long combination(int n, int k) {
 if (k < 0 || k > n) return 0;
 if (k == 0 || k == n) return 1;

 long long result = 1;
 for (int i = 1; i <= k; i++) {
 result = result * (n - i + 1) / i;
 }
 return result;
}

/***
 * 测试函数
 */
void testDivisibleGroupSums() {
 // 测试用例
}

```

```

vector<vector<int>> numsCases = {
 {1, 2, 3, 4, 5},
 {2, 4, 6, 8, 10},
 {-1, 1, -2, 2, -3, 3},
 {10, 20, 30, 40, 50}
};

vector<int> MList = {2, 3, 2, 3};
vector<int> DList = {3, 2, 3, 10};

cout << "可整除组和问题测试: " << endl;
for (size_t i = 0; i < numsCases.size(); i++) {
 vector<int> nums = numsCases[i];
 int M = MList[i];
 int D = DList[i];

 long long result1 = DivisibleGroupSums::divisibleGroupSums(nums, M, D);
 long long result2 = DivisibleGroupSums::divisibleGroupSums2D(nums, M, D);
 long long result3 = DivisibleGroupSums::divisibleGroupSumsOptimized(nums, M, D);

 cout << "nums=[";
 for (size_t j = 0; j < nums.size(); j++) {
 cout << nums[j];
 if (j < nums.size() - 1) cout << ", ";
 }
 cout << "], M=" << M << ", D=" << D
 << ": 方法 1=" << result1 << ", 方法 2=" << result2
 << ", 方法 3=" << result3 << endl;

 // 验证结果一致性
 if (result1 != result2 || result2 != result3) {
 cout << "警告: 不同方法结果不一致! " << endl;
 }
}

// 性能测试 - 大规模数据
int n = 50;
vector<int> largeNums(n);
random_device rd;
mt19937 gen(rd());
uniform_int_distribution<> dist(-500, 500);

for (int i = 0; i < n; i++) {
 largeNums[i] = dist(gen); // 包含负数
}

```

```

}

int largeM = 10;
int largeD = 7;

auto startTime = chrono::high_resolution_clock::now();
long long largeResult = DivisibleGroupSums::divisibleGroupSumsOptimized(largeNums, largeM,
largeD);

auto endTime = chrono::high_resolution_clock::now();
auto duration = chrono::duration_cast<chrono::milliseconds>(endTime - startTime);

cout << "大规模测试: 数字数量=" << n << ", M=" << largeM
 << ", D=" << largeD << ", 结果=" << largeResult
 << ", 耗时=" << duration.count() << "ms" << endl;

// 边界情况测试
cout << "边界情况测试: " << endl;
vector<int> emptyNums;
cout << "空数组, M=0: " << DivisibleGroupSums::divisibleGroupSums(emptyNums, 0, 5) << endl;
cout << "空数组, M=1: " << DivisibleGroupSums::divisibleGroupSums(emptyNums, 1, 5) << endl;
cout << "M=0: " << DivisibleGroupSums::divisibleGroupSums({1, 2, 3}, 0, 5) << endl;
cout << "M>n: " << DivisibleGroupSums::divisibleGroupSums({1, 2}, 3, 5) << endl;

// 特殊测试: D=1 (所有组合都满足)
cout << "D=1 特殊测试: " << endl;
vector<int> testNums = {1, 2, 3};
long long specialResult = DivisibleGroupSums::divisibleGroupSums(testNums, 2, 1);
long long expected = combination(3, 2);
cout << "D=1, 预期=C(3,2)=" << expected << ", 实际=" << specialResult << endl;

if (specialResult == expected) {
 cout << "D=1 测试验证通过" << endl;
} else {
 cout << "D=1 测试验证失败" << endl;
}
}

int main() {
 try {
 testDivisibleGroupSums();
 } catch (const exception& e) {
 cerr << "错误: " << e.what() << endl;
 return 1;
 }
}

```

```
 return 0;
}

/*
 * 复杂度分析:
 *
 * 方法 1: 动态规划 (滚动数组)
 * - 时间复杂度: O(N * M * D)
 * - N: 数字数量
 * - M: 选择数量
 * - D: 除数
 * - 空间复杂度: O(M * D)
 *
 * 方法 2: 三维动态规划
 * - 时间复杂度: O(N * M * D)
 * - 空间复杂度: O(N * M * D)
 *
 * 方法 3: 空间优化的动态规划
 * - 时间复杂度: O(N * M * D)
 * - 空间复杂度: O(M * D) (使用两个二维数组)
 *
 * C++特定优化:
 * 1. 使用 vector 代替数组, 更安全
 * 2. 使用 STL 算法进行填充和交换
 * 3. 使用随机数生成器进行性能测试
 * 4. 使用 chrono 进行精确性能测量
 *
 * 关键点分析:
 * 1. 模数运算: 正确处理负数取模, 使用(k - mod + D) % D
 * 2. 状态定义: dp[i][j][k] 表示前 i 个数字选 j 个模 D 为 k 的方案数
 * 3. 空间优化: 使用滚动数组减少空间复杂度
 * 4. 边界处理: M=0 时方案数为 1 (空选择)
 *
 * 工程化考量:
 * 1. 模块化设计: 将不同解法封装为静态方法
 * 2. 异常处理: 使用 try-catch 处理异常
 * 3. 性能优化: 利用 STL 容器和算法
 * 4. 测试覆盖: 包含各种边界情况和性能测试
 *
 * 面试要点:
 * 1. 理解分组背包+模数运算的组合
 * 2. 掌握模数运算的处理技巧
 * 3. 了解不同 DP 实现的空间优化
```

\* 4. 能够分析算法的时空复杂度

\*/

=====

文件: Code50\_DivisibleGroupSums. java

=====

```
package class073;

import java.util.Arrays;

// UVA 10616 Divisible Group Sums
// 题目描述: 给定 N 个整数, 选择 M 个数字使得它们的和能被 D 整除, 求方案数。
// 链接:
https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=1557
//
// 解题思路:
// 这是一个分组背包+模数运算的问题, 需要计算选择 M 个数字的和能被 D 整除的方案数。
//
// 状态定义: dp[i][j][k] 表示前 i 个数字, 选择 j 个数字, 和对 D 取模为 k 的方案数
// 状态转移方程:
// dp[i][j][k] = dp[i-1][j][k] + dp[i-1][j-1][(k - num[i] % D + D) % D]
//
// 关键点:
// 1. 模数运算: 处理负数取模, 使用(k - num[i] % D + D) % D
// 2. 分组背包: 每个数字只能选择一次
// 3. 空间优化: 使用滚动数组优化空间复杂度
//
// 时间复杂度: O(N * M * D), 其中 N 是数字数量, M 是选择数量, D 是除数
// 空间复杂度: O(M * D), 使用滚动数组优化
//
// 工程化考量:
// 1. 模数处理: 正确处理负数取模
// 2. 边界条件: 处理 M=0、D=0 等特殊情况
// 3. 性能优化: 使用滚动数组和模数运算优化
// 4. 异常处理: 处理除数为 0 的情况

public class Code50_DivisibleGroupSums {

 /**
 * 动态规划解法 - 分组背包+模数运算
 * @param nums 整数数组
 * @param M 需要选择的数字个数

```

```

* @param D 除数
* @return 方案数
*/
public static long divisibleGroupSums(int[] nums, int M, int D) {
 // 参数验证
 if (nums == null || nums.length == 0) {
 return M == 0 ? 1 : 0;
 }
 if (D == 0) {
 throw new IllegalArgumentException("Divisor D cannot be zero");
 }
 if (M < 0 || M > nums.length) {
 return 0;
 }

 int n = nums.length;

 // 创建 DP 数组，使用滚动数组优化
 long[][] dp = new long[M + 1][D];
 dp[0][0] = 1; // 选择 0 个数字，和为 0 (模 D 为 0) 的方案数为 1

 // 遍历每个数字
 for (int i = 0; i < n; i++) {
 int num = nums[i];
 int mod = (num % D + D) % D; // 处理负数取模

 // 倒序遍历选择数量，避免重复选择
 for (int j = M; j >= 1; j--) {
 // 倒序遍历模数状态
 long[] temp = Arrays.copyOf(dp[j], D);
 for (int k = 0; k < D; k++) {
 int prevMod = (k - mod + D) % D;
 dp[j][k] += dp[j - 1][prevMod];
 }
 }
 }

 return dp[M][0];
}

/**
 * 优化的动态规划解法 - 二维数组
 */

```

```

public static long divisibleGroupSums2D(int[] nums, int M, int D) {
 if (nums == null || nums.length == 0) {
 return M == 0 ? 1 : 0;
 }
 if (D == 0) {
 throw new IllegalArgumentException("Divisor D cannot be zero");
 }
 if (M < 0 || M > nums.length) {
 return 0;
 }

 int n = nums.length;

 // 创建三维 DP 数组
 long[][][] dp = new long[n + 1][M + 1][D];
 dp[0][0][0] = 1;

 for (int i = 1; i <= n; i++) {
 int num = nums[i - 1];
 int mod = (num % D + D) % D;

 for (int j = 0; j <= M; j++) {
 for (int k = 0; k < D; k++) {
 // 不选当前数字
 dp[i][j][k] += dp[i - 1][j][k];

 // 选当前数字
 if (j >= 1) {
 int prevMod = (k - mod + D) % D;
 dp[i][j][k] += dp[i - 1][j - 1][prevMod];
 }
 }
 }
 }

 return dp[n][M][0];
}

/**
 * 空间优化的解法 - 使用两个二维数组交替
 */
public static long divisibleGroupSumsOptimized(int[] nums, int M, int D) {
 if (nums == null || nums.length == 0) {

```

```

 return M == 0 ? 1 : 0;
 }

 if (D == 0) {
 throw new IllegalArgumentException("Divisor D cannot be zero");
 }

 if (M < 0 || M > nums.length) {
 return 0;
 }

 int n = nums.length;

 // 使用两个二维数组交替
 long[][] dp = new long[M + 1][D];
 long[][] next = new long[M + 1][D];
 dp[0][0] = 1;

 for (int i = 0; i < n; i++) {
 int num = nums[i];
 int mod = (num % D + D) % D;

 // 复制当前状态到 next
 for (int j = 0; j <= M; j++) {
 System.arraycopy(dp[j], 0, next[j], 0, D);
 }

 // 更新 next 数组
 for (int j = 1; j <= M; j++) {
 for (int k = 0; k < D; k++) {
 int prevMod = (k - mod + D) % D;
 next[j][k] += dp[j - 1][prevMod];
 }
 }
 }

 // 交换数组
 long[][] temp = dp;
 dp = next;
 next = temp;

 // 清空 next 数组用于下一次迭代
 for (int j = 0; j <= M; j++) {
 Arrays.fill(next[j], 0);
 }
}

```

```

 return dp[M][0];
 }

/***
 * 递归+记忆化搜索解法
 */
public static long divisibleGroupSumsDFS(int[] nums, int M, int D) {
 if (nums == null || nums.length == 0) {
 return M == 0 ? 1 : 0;
 }
 if (D == 0) {
 throw new IllegalArgumentException("Divisor D cannot be zero");
 }
 if (M < 0 || M > nums.length) {
 return 0;
 }

 int n = nums.length;
 Long[][][] memo = new Long[n][M + 1][D];
 return dfs(nums, 0, M, 0, D, memo);
}

private static long dfs(int[] nums, int index, int remaining, int currentMod, int D,
Long[][][] memo) {
 // 基础情况
 if (index == nums.length) {
 return (remaining == 0 && currentMod == 0) ? 1 : 0;
 }

 // 检查记忆化数组
 if (memo[index][remaining][currentMod] != null) {
 return memo[index][remaining][currentMod];
 }

 long count = 0;

 // 不选当前数字
 count += dfs(nums, index + 1, remaining, currentMod, D, memo);

 // 选当前数字
 if (remaining > 0) {
 int num = nums[index];

```

```

 int mod = (num % D + D) % D;
 int newMod = (currentMod + mod) % D;
 count += dfs(nums, index + 1, remaining - 1, newMod, D, memo);
 }

 memo[index][remaining][currentMod] = count;
 return count;
}

/***
 * 测试方法
 */
public static void main(String[] args) {
 // 测试用例
 int[][] numsCases = {
 {1, 2, 3, 4, 5},
 {2, 4, 6, 8, 10},
 {-1, 1, -2, 2, -3, 3},
 {10, 20, 30, 40, 50}
 };
 int[] MList = {2, 3, 2, 3};
 int[] DList = {3, 2, 3, 10};

 System.out.println("可整除组和问题测试: ");
 for (int i = 0; i < numsCases.length; i++) {
 int[] nums = numsCases[i];
 int M = MList[i];
 int D = DList[i];

 long result1 = divisibleGroupSums(nums, M, D);
 long result2 = divisibleGroupSums2D(nums, M, D);
 long result3 = divisibleGroupSumsOptimized(nums, M, D);
 long result4 = divisibleGroupSumsDFS(nums, M, D);

 System.out.printf("nums=%s, M=%d, D=%d: 方法 1=%d, 方法 2=%d, 方法 3=%d, 方法 4=%d%n",
 Arrays.toString(nums), M, D, result1, result2, result3, result4);

 // 验证结果一致性
 if (result1 != result2 || result2 != result3 || result3 != result4) {
 System.out.println("警告: 不同方法结果不一致!");
 }
 }
}

```

```

// 性能测试 - 大规模数据
int n = 50;
int[] largeNums = new int[n];
java.util.Random random = new java.util.Random();
for (int i = 0; i < n; i++) {
 largeNums[i] = random.nextInt(1000) - 500; // 包含负数
}
int largeM = 10;
int largeD = 7;

long startTime = System.currentTimeMillis();
long largeResult = divisibleGroupSumsOptimized(largeNums, largeM, largeD);
long endTime = System.currentTimeMillis();

System.out.printf("大规模测试: 数字数量=%d, M=%d, D=%d, 结果=%d, 耗时=%dms%n",
 n, largeM, largeD, largeResult, endTime - startTime);

// 边界情况测试
System.out.println("边界情况测试: ");
System.out.println("空数组, M=0: " + divisibleGroupSums(new int[] {}, 0, 5));
System.out.println("空数组, M=1: " + divisibleGroupSums(new int[] {}, 1, 5));
System.out.println("M=0: " + divisibleGroupSums(new int[] {1, 2, 3}, 0, 5));
System.out.println("M>n: " + divisibleGroupSums(new int[] {1, 2}, 3, 5));

// 特殊测试: D=1 (所有组合都满足)
System.out.println("D=1 特殊测试: ");
long specialResult = divisibleGroupSums(new int[] {1, 2, 3}, 2, 1);
System.out.println("D=1, 预期=C(3,2)=3, 实际=" + specialResult);

// 验证组合数公式
long expected = combination(3, 2);
if (specialResult == expected) {
 System.out.println("D=1 测试验证通过");
} else {
 System.out.println("D=1 测试验证失败");
}

/**
 * 计算组合数 C(n, k)
 */
private static long combination(int n, int k) {
 if (k < 0 || k > n) return 0;

```

```

 if (k == 0 || k == n) return 1;

 long result = 1;
 for (int i = 1; i <= k; i++) {
 result = result * (n - i + 1) / i;
 }
 return result;
}

/*
* 复杂度分析:
*
* 方法 1: 动态规划 (滚动数组)
* - 时间复杂度: O(N * M * D)
* - N: 数字数量
* - M: 选择数量
* - D: 除数
* - 空间复杂度: O(M * D)
*
* 方法 2: 三维动态规划
* - 时间复杂度: O(N * M * D)
* - 空间复杂度: O(N * M * D)
*
* 方法 3: 空间优化的动态规划
* - 时间复杂度: O(N * M * D)
* - 空间复杂度: O(M * D) (使用两个二维数组)
*
* 方法 4: 递归+记忆化搜索
* - 时间复杂度: O(N * M * D) (每个状态计算一次)
* - 空间复杂度: O(N * M * D) (记忆化数组)
*
* 关键点分析:
* 1. 模数运算: 正确处理负数取模, 使用(k - mod + D) % D
* 2. 状态定义: dp[i][j][k]表示前 i 个数字选 j 个模 D 为 k 的方案数
* 3. 空间优化: 使用滚动数组减少空间复杂度
* 4. 边界处理: M=0 时方案数为 1 (空选择)
*
* 工程化考量:
* 1. 参数验证: 检查 D=0、M 越界等情况
* 2. 性能优化: 使用滚动数组和模数运算优化
* 3. 测试覆盖: 包含正常、边界、性能测试
* 4. 特殊验证: D=1 时验证组合数公式

```

```
*
* 面试要点:
* 1. 理解分组背包+模数运算的组合
* 2. 掌握模数运算的处理技巧
* 3. 了解不同 DP 实现的空间优化
* 4. 能够分析算法的时空复杂度

*
* 扩展应用:
* 1. 数论问题中的模数运算
* 2. 组合数学中的计数问题
* 3. 密码学中的模数运算
* 4. 分布式系统中的一致性哈希
*/
```

=====

文件: Code50\_DivisibleGroupSums.py

=====

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

UVA 10616 Divisible Group Sums
题目描述: 给定 N 个整数, 选择 M 个数字使得它们的和能被 D 整除, 求方案数。
链接:
https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=1557

解题思路:
这是一个分组背包+模数运算的问题, 需要计算选择 M 个数字的和能被 D 整除的方案数。

状态定义: dp[i][j][k] 表示前 i 个数字, 选择 j 个数字, 和对 D 取模为 k 的方案数
状态转移方程:
dp[i][j][k] = dp[i-1][j][k] + dp[i-1][j-1][(k - num[i] % D + D) % D]

关键点:
1. 模数运算: 处理负数取模, 使用(k - num[i] % D + D) % D
2. 分组背包: 每个数字只能选择一次
3. 空间优化: 使用滚动数组优化空间复杂度

时间复杂度: O(N * M * D), 其中 N 是数字数量, M 是选择数量, D 是除数
空间复杂度: O(M * D), 使用滚动数组优化
#
```

```
工程化考量:
1. 模数处理: 正确处理负数取模
2. 边界条件: 处理 M=0、D=0 等特殊情况
3. 性能优化: 使用滚动数组和模数运算优化
4. 异常处理: 处理除数为 0 的情况
```

```
class DivisibleGroupSums:
```

```
 """
```

```
 可整除组和问题的多种解法
```

```
 """
```

```
@staticmethod
```

```
def divisible_group_sums(nums: List[int], M: int, D: int) -> int:
```

```
 """
```

```
 动态规划解法 - 分组背包+模数运算
```

```
Args:
```

```
 nums: 整数数组
```

```
 M: 需要选择的数字个数
```

```
 D: 除数
```

```
Returns:
```

```
 int: 方案数
```

```
 """
```

```
参数验证
```

```
if not nums:
```

```
 return 1 if M == 0 else 0
```

```
if D == 0:
```

```
 raise ValueError("Divisor D cannot be zero")
```

```
if M < 0 or M > len(nums):
```

```
 return 0
```

```
n = len(nums)
```

```
创建 DP 数组, 使用滚动数组优化
```

```
dp = [[0] * D for _ in range(M + 1)]
```

```
dp[0][0] = 1 # 选择 0 个数字, 和为 0 (模 D 为 0) 的方案数为 1
```

```
遍历每个数字
```

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

```
 num = nums[i]
```

```
 mod = (num % D + D) % D # 处理负数取模
```

```

倒序遍历选择数量，避免重复选择
for j in range(M, 0, -1):
 # 创建临时数组保存当前状态
 temp = dp[j][:]
 for k in range(D):
 prev_mod = (k - mod + D) % D
 dp[j][k] += dp[j - 1][prev_mod]

return dp[M][0]

```

```

@staticmethod
def divisible_group_sums_2d(nums: List[int], M: int, D: int) -> int:
 """
 优化的动态规划解法 - 二维数组

```

Args:

nums: 整数数组  
 M: 需要选择的数字个数  
 D: 除数

Returns:

int: 方案数

```

"""
if not nums:
 return 1 if M == 0 else 0
if D == 0:
 raise ValueError("Divisor D cannot be zero")
if M < 0 or M > len(nums):
 return 0

n = len(nums)

```

# 创建三维 DP 数组

```

dp = [[[0] * D for _ in range(M + 1)] for _ in range(n + 1)]
dp[0][0][0] = 1

```

```

for i in range(1, n + 1):
 num = nums[i - 1]
 mod = (num % D + D) % D

```

```

 for j in range(M + 1):
 for k in range(D):
 # 不选当前数字

```

```

 dp[i][j][k] += dp[i - 1][j][k]

 # 选当前数字
 if j >= 1:
 prev_mod = (k - mod + D) % D
 dp[i][j][k] += dp[i - 1][j - 1][prev_mod]

 return dp[n][M][0]

```

```

@staticmethod
def divisible_group_sums_optimized(nums: List[int], M: int, D: int) -> int:
 """

```

空间优化的解法 - 使用两个二维数组交替

Args:

nums: 整数数组  
 M: 需要选择的数字个数  
 D: 除数

Returns:

int: 方案数

```

if not nums:
 return 1 if M == 0 else 0
if D == 0:
 raise ValueError("Divisor D cannot be zero")
if M < 0 or M > len(nums):
 return 0

```

n = len(nums)

# 使用两个二维数组交替

```

dp = [[0] * D for _ in range(M + 1)]
next_dp = [[0] * D for _ in range(M + 1)]
dp[0][0] = 1

```

```

for i in range(n):
 num = nums[i]
 mod = (num % D + D) % D

```

```

 # 复制当前状态到 next_dp
 for j in range(M + 1):
 next_dp[j] = dp[j][:]

```

```

更新 next_dp 数组
for j in range(1, M + 1):
 for k in range(D):
 prev_mod = (k - mod + D) % D
 next_dp[j][k] += dp[j - 1][prev_mod]

交换数组
dp, next_dp = next_dp, dp

清空 next_dp 数组用于下一次迭代
for j in range(M + 1):
 next_dp[j] = [0] * D

return dp[M][0]

@staticmethod
def combination(n: int, k: int) -> int:
 """
 计算组合数 C(n, k)
 """

 if k < 0 or k > n:
 return 0
 if k == 0 or k == n:
 return 1

 result = 1
 for i in range(1, k + 1):
 result = result * (n - i + 1) // i
 return result

@staticmethod
def run_tests():
 """
 运行测试用例，验证所有方法的正确性
 """

 # 测试用例
 nums_cases = [
 [1, 2, 3, 4, 5],
 [2, 4, 6, 8, 10],
 [-1, 1, -2, 2, -3, 3],
 [10, 20, 30, 40, 50]
]

```

```

M_list = [2, 3, 2, 3]
D_list = [3, 2, 3, 10]

print("可整除组和问题测试: ")
for i, (nums, M, D) in enumerate(zip(nums_cases, M_list, D_list)):
 try:
 result1 = DivisibleGroupSums.divisible_group_sums(nums, M, D)
 result2 = DivisibleGroupSums.divisible_group_sums_2d(nums, M, D)
 result3 = DivisibleGroupSums.divisible_group_sums_optimized(nums, M, D)

 print(f"nums={nums}, M={M}, D={D}: "
 f"方法 1={result1}, 方法 2={result2}, 方法 3={result3}")

 # 验证结果一致性
 if result1 != result2 or result2 != result3:
 print("警告: 不同方法结果不一致!")

 except Exception as e:
 print(f"测试用例{i+1}时发生错误: {e}")

性能测试 - 大规模数据
n = 50
large_nums = [random.randint(-500, 500) for _ in range(n)]
large_M = 10
large_D = 7

start_time = time.time()
large_result = DivisibleGroupSums.divisible_group_sums_optimized(large_nums, large_M,
large_D)
end_time = time.time()

print(f"大规模测试: 数字数量={n}, M={large_M}, D={large_D}, "
 f"结果={large_result}, 耗时={end_time - start_time:.4f}秒")

边界情况测试
print("边界情况测试: ")
print(f"空数组, M=0: {DivisibleGroupSums.divisible_group_sums([], 0, 5)}")
print(f"空数组, M=1: {DivisibleGroupSums.divisible_group_sums([], 1, 5)}")
print(f"M=0: {DivisibleGroupSums.divisible_group_sums([1, 2, 3], 0, 5)}")
print(f"M>n: {DivisibleGroupSums.divisible_group_sums([1, 2], 3, 5)}")

特殊测试: D=1 (所有组合都满足)
print("D=1 特殊测试: ")

```

```

test_nums = [1, 2, 3]
special_result = DivisibleGroupSums.divisible_group_sums(test_nums, 2, 1)
expected = DivisibleGroupSums.combination(3, 2)
print(f"D=1, 预期=C(3, 2)={expected}, 实际={special_result}")

if special_result == expected:
 print("D=1 测试验证通过")
else:
 print("D=1 测试验证失败")

def main():
 """
 主函数 - 运行测试和演示
 """
 try:
 DivisibleGroupSums.run_tests()
 except Exception as e:
 print(f"程序执行错误: {e}")
 return 1
return 0

if __name__ == "__main__":
 sys.exit(main())

```

"""

复杂度分析:

方法 1: 动态规划 (滚动数组)

- 时间复杂度:  $O(N * M * D)$ 
  - N: 数字数量
  - M: 选择数量
  - D: 除数
- 空间复杂度:  $O(M * D)$

方法 2: 三维动态规划

- 时间复杂度:  $O(N * M * D)$
- 空间复杂度:  $O(N * M * D)$

方法 3: 空间优化的动态规划

- 时间复杂度:  $O(N * M * D)$
- 空间复杂度:  $O(M * D)$  (使用两个二维数组)

Python 特定优化:

1. 使用列表推导式和切片操作
2. 利用 Python 的动态类型特性
3. 使用类型注解提高代码可读性
4. 使用 random 模块进行性能测试

关键点分析:

1. 模数运算: 正确处理负数取模, 使用  $(k - \text{mod} + D) \% D$
2. 状态定义:  $dp[i][j][k]$  表示前  $i$  个数字选  $j$  个模  $D$  为  $k$  的方案数
3. 空间优化: 使用滚动数组减少空间复杂度
4. 边界处理:  $M=0$  时方案数为 1 (空选择)

工程化考量:

1. 模块化设计: 将不同解法封装为静态方法
2. 异常处理: 完善的参数验证和错误处理
3. 性能监控: 包含性能测试和时间测量
4. 测试覆盖: 包含各种边界情况和性能测试

面试要点:

1. 理解分组背包+模数运算的组合
2. 掌握模数运算的处理技巧
3. 了解不同 DP 实现的空间优化
4. 能够分析算法的时空复杂度

扩展应用:

1. 数论问题中的模数运算
2. 组合数学中的计数问题
3. 密码学中的模数运算
4. 分布式系统中的一致性哈希

"""

=====