硬币找零问题的讨论

原创 王泽霖 LOA算法学习笔记 2021-02-01 18:56

01 问题描述

给定不同面额的硬币和一个总金额。设计一个算法，计算可以凑成总金额的硬币组合数。假设每一种面额的硬币有无限个。

02 问题分析

根据题意，我们定义两个输入变量：
1. 给定的不同面额的零钱数组 $C=[C_1, C_2, \cdots, C_n]$
2. 给定的总金额量 $Amount$

类似地，我们定义一个函数表示输出值：$TotalNumbers(C, Amount)$，表示在给定零钱序列和给定总金额的情况下，所有的找零方案数。

首先，我们思考最简单的情况：显而易见的，应该从零金额和零硬币数入手分析。

1. 给定金额量为0， $Amount=0$：此时，如果我们没有硬币，亦即我们可以用0个硬币凑成金额0，因此总的找零方案数为1。
2. 给定金额量为0，但我们有至少一个硬币，显然无法用至少一个硬币凑成金额0，因此总的找零方案数为0。
3. 给定零钱数量为0， C 为空序列，此时，总的找零方案数为0。
4. 给定零钱数量为1， $C=[C_1]$，显然，如果此时 C_1 能够整除总金额 $Amount$，则总的找零方案数为1，否则为0。

其次，我们思考一些更复杂的情况，看看能不能把复杂的情况转化为最简单的四种情况?

1. 第一步分析给定零钱数量为2， $C=[C_1, C_2]$，为了转化为仅有一个零钱的情况，我们将这个问题转化为 k_1+1 个子问题（想想看， k_1 是什么？），每个问题的总金额量为 $Amount-i\times C_2, 0\le i\le k_1$，零钱数量仅有一个，为 $C=[C_1]$。此情况下的总找零数如公式1所示。（Answer: $k_1=\lfloor\frac{Amount}{C_2}\rfloor$， k_1 仅用来计数有多少个子问题。）。不难发现，我们已经成功地将一个较大的问题化成了最简单的那四个Case!

$$TotalNumbers([C_1,C_2],Amount)=\sum\begin{cases}TotalNumbers([C_1],Amount)\\TotalNumbers([C_1],Amount-C_2)\\TotalNumbers([C_1],Amount-2\times C_2)\\\cdots\\TotalNumbers([C_1],Amount-k_1\times C_2),\\\qquad when\quad k_1\times C_2\le Amount\end{cases}$$

2. 我们接下来加大分析的难度，现在我们分析三个硬币的情况， $C=[C_1, C_2, C_3]$。同样的，我们将这个问题转化为 k_2 个子问题，每个问题的总金额量为 $Amount-k_2\times C_3$，零钱数量有两个，为 $C=[C_1, C_2]$。此情况下的总找零数如公式2所示。看起来，三个硬币的情况也被转化为我们已知的问题的解了!

$$TotalNumbers([C_1,C_2,C_3],Amount)=\sum_{k=0}^{\lfloor Amount/C_3\rfloor}TotalNumbers([C_1,C_2],Amount-k\times C_3),$$

3. 现在我们分析完了吗？并没有，我们要将特殊情况的解推广到一般情况：因此，我们可以将子问题定义为：在给定总金额为 $Amount$ 的前提下，在前 i 个零钱中累计所有的找零方案数，并将这个所有的方案数记为 $OPT(i, Amount)$ 。

4. 根据对复杂情况的分析，我们可以得到公式3所示的递归关系：

$$OPT(i, Amount) = \begin{cases} 1 & Amount = 0, i = 0 \\ 0 & i = 1, 0 \\ \sum_{k=0}^{\lfloor \frac{Amount}{C_i} \rfloor} OPT(i-1, Amount - k \times C_i) & other \end{cases}$$

5. 现在我们终于松了一口气！这个问题可算分析完了，我们终于得到的一般情况的解。然后借助上面的公式，递归算法应该不难写出了。接下来我们面临一个新的问题：能不能把递归算法转化为迭代算法呢？

03 基础迭代算法

显然是可以的！我们这里借助卜老师上课讲过的一种思想，叫做“以存代算”，也就是将计算好的简单情况存下来，存成一张表，计算复杂情况的时候直接索引包含所有简单情况的表即可。

我们想把计算过的情况存起来，表示“已知的、简单的情况”，所以我们建立一张二维表 $dp[i][j]$ ，第i行表示我们目前分析到了前i枚硬币，第j列表示目前分析到了金额数为j。我们在第二节分析最简单Case的时候，发现金额0=和硬币数0的情况需要单独考虑，所以如果有n枚硬币，就需要n+1行；有总金额为 $Amount$ ，就需要 $Amount + 1$ 列。

根据这样的分析，我们可以轻松地写出如下的迭代算法：

```
int[][] dp = new int[coins.length + 1][amount + 1];
dp[0][0] = 1;
for (int i = 1; i <= n; i++) {
    for (int j = 0; j <= amount; j++) {
        for (int k = 0; k * coins[i - 1] <= j; k++) {
            dp[i][j] = dp[i][j] +  dp[i - 1][j - k * coins[i - 1]];
        }
    }
}
return dp[n][amount];
```

04 基础迭代算法的复杂度

4.1 时间复杂度

基础迭代算法中包含三层循环，第一层循环次数由硬币数决定，第二、三层循环次数由金额数决定，所以总的时间复杂度为： $O(N \times M^2), where\ N = coins.length,\quad M = Amount$ 。

4.2 空间复杂度

我们开辟了一个大小为 $(N+1) \times (M+1)$ 的矩阵，因此空间复杂度为 $O(N \times M)$.

05 优化迭代算法1

1. 基础迭代算法需要三层循环，有没有什么办法优化呢?
2. 我们观察基础算法中的热点代码段。发现是这一句 dp[i][j] = dp[i][j] + dp[i - 1][j - k * coins[i - 1]]。
3. 不难发现，计算dp[i][j] 时，仅仅用到了dp[i]和dp[i-1]这两行数据，其余行的数据，一个都没用上！所以基础迭代算法显然有优化的空间。这里利用一个"滚动数组"的思想，就是只保留最近一行数据，计算前i-1枚硬币的找零情况，存到dp[0]这一行中，那么，当我们计算前i个硬币的时候，利用dp[0]中的数据，存到dp[1]这一行中。最后我们将dp[1]行的数据替换掉dp[0]的数据即可开始下一次循环。伪代码如下所示:

```
int[][] dp = new int[2][amount + 1];
dp[0][0] = 1;
for (int i = coins[0]; i <= amount; i += coins[0]) {
dp[0][i] = 1;
}


for (int i = 1; i < len; i++) {
// 当前要填充的行清零
Arrays.fill(dp[i & 1], 0);
for (int j = 0; j <= amount; j++) {
for (int k = 0; j - k * coins[i] >= 0; k++) {
dp[i & 1][j] += dp[(i - 1) & 1][j - k * coins[i]];
}
}


return dp[(len - 1) & 1][amount];
```

4. 这个优化思路最容易想到，但这仅仅降低了空间复杂度（ $O(N \times M) \Rightarrow O(M)$ ），时间复杂度仍然不变。还有什么优化的思路呢?

06 优化迭代算法2

1. 我们来重新观察上文推到的最优子结构，这是我们已经推导得到的状态转移方程，记作公式4，如下所示。

$$dp[i][j] = \sum \begin{cases} dp[i-1][j-0 \times coins[i]] \\ dp[i-1][j-1 \times coins[i]] \\ dp[i-1][j-2 \times coins[i]] \\ \dots \\ dp[i-1][j-k \times coins[i]], \\ \qquad when \quad k \times coins[i] \le j \end{cases}$$

2. 假定 $j \ge coins[i]$ 时，我们对公式4稍加变换，得到公式5:

$$dp[i][j-coins[i]] = \sum \begin{cases} dp[i-1][j-coins[i]-0\times coins[i]] \\ dp[i-1][j-coins[i]-1\times coins[i]] \\ dp[i-1][j-coins[i]-2\times coins[i]] \\ \ldots \\ dp[i-1][j-coins[i]-k'\times coins[i]], \\ \qquad when \quad (k'+1)\times coins[i] \le j \end{cases}$$

3. 把公式5整理一下，得到公式6：在满足 $j \ge coins[i]$ 的条件下

$$dp[i][j-coins[i]] = \sum \begin{cases} dp[i-1][j-1\times coins[i]] \\ dp[i-1][j-2\times coins[i]] \\ dp[i-1][j-3\times coins[i]] \\ \ldots \\ dp[i-1][j-k\times coins[i]], \\ \qquad when \quad k\times coins[i] \le j \end{cases}$$

4. 我们最终关心的还是如何得到 $dp[i][j]$ ，所以令公式4减去公式6得公式7，如下所示

$$dp[i][j] - dp[i][j-coins[i]] = dp[i-1][j]$$

5. 公式7整理一下，得到公式8，如下所示：

$$dp[i][j] = dp[i-1][j] + dp[i][j-coins[i]]$$

6. 我们惊讶地发现，计算当前 $dp[i][j]$ 的值的时候，仅仅参考了第i行的前面的值、以及 $dp[i-1][j]$ 这两个值。换言之，我们的“历史表格”只有一行就足够了！下面是最终优化版本的伪代码：

```
int[] dp = new int[amount + 1];
dp[0] = 1;


for (int i = coins[0]; i <= amount; i += coins[0]) {
dp[i] = 1;
}


for (int i = 1; i < len; i++) {
for (int j = coins[i] ; j <= amount; j++) {
dp[j] += dp[j - coins[i]];
}
}
return dp[amount];
```

7. 对于这个优化算法2，其时间复杂为 $O(N \times M)$ ，空间复杂度为 $O(M)$ 。相较于原始算法的时间与空间复杂度。优化算法2的时空复杂度均降低了一个乘法因子的复杂度！

07 个人感悟

1. 遇见陌生的问题先从最基本的情况分析，最基本的情况分析透彻了再考虑复杂的情况。
2. 对于一个问题的求解，不要妄图一上手就直接能给出一个精巧的算法，每一个精巧的算法的背后都是从一个最笨拙的算法开始，一点一点优化出来的。（类比卜老师上课时介绍的那样：从动态规划到贪心，可能有些问题的贪心策略非常精巧，这时候我们就应该先给出一个笨拙的动态规划算法，然后逐步构造那个目标问题的贪心策略。）

结语：由于本人的算法水平也处于初学者的级别，若本文有疏忽或谬误，希望读者通过邮件联系本人进行更正(mail: wangzelin20s@ict.ac.cn)；或者某些公式、语句、伪代码有更好的、更优雅的表达方式，也可以给本人发邮件，进一步讨论。