04 动态规划：完美解决硬币找零

你好，我是卢誉声。今天我们来继续学习动态规划。

在前面的几节课中，我们经历了贪心算法求解硬币找零的问题，并从中发现了贪心算法本身的局限性：它几乎只考虑了局部最优，因此无法应对需要考虑整体最优的算法面试问题。

针对这一问题，我们重新思考了解决方案，用递归的方法来**穷举**出所有可能的组合，从这些可能组合中找出最优解。虽然这么做考虑了整体最优，而且真的可以解决问题，但效率太低。因此，为了解决这个低效问题，我们又提出了备忘录的概念，并在硬币找零案例中应用它解决了问题。

你应该发现了，我们在解决硬币找零问题时的思路是一以贯之的：发现问题，找解决方案；如果方案有局限性，那么就看如何扩展视野，找寻更优的方法。

不知道你还记不记得，我在上一节课的结尾有提到：含有备忘录的递归算法已经与动态规划思想十分相似了，从效率上说也是如此。

事实上，你已经在使用动态规划的思想解决问题了。但**“真正”的动态规划解法跟备忘录法又有什么区别呢?**

接下来我们就带着这个问题，一起来学习今天的内容吧！

动态规划的问题描述

我们曾不止一次提到重叠子问题，并在上一课对其做了深入探讨。其实，重叠子问题是考虑一个问题是否为动态规划问题的先决条件，除此之外，我还提到了无后效性。

嗯，是时候对这些问题做个总结了，动态规划问题一定具备以下三个特征：

1. 重叠子问题：在穷举的过程中（比如通过递归），存在重复计算的现象；
2. 无后效性：子问题之间的依赖是单向性的，某阶段状态一旦确定，就不受后续决策的影响；
3. 最优子结构：子问题之间必须相互独立，或者说后续的计算可以通过前面的状态推导出来。

什么是最优子结构?

这是我第一次在课程中提出**最优子结构**这一概念，所以咱们先了解一下。这东西乍一听有些玄乎，什么叫子问题之间必须相互独立？我举一个简单的例子，你就明白了。

比如说，假设你在外卖平台购买5斤苹果和3斤香蕉。由于促销的缘故，这两种水果都有一个互相独立的促销价。如果原问题是让你以最低的价格购买这些水果，你该怎么买？显然，由于这两种水果的促销价格相互独立、互不影响，你只需直接购买就能享受到最低折扣的价格。

现在我们得到了正确的结果：最低价格就是直接购买这两种折扣水果。因为这个过程符合最优子结构，打折的苹果和香蕉这两个子问题是互相独立、互不干扰的。

但是，如果平台追加一个条件：折扣不能同时享用。即购买了折扣的苹果就不能享受折扣的香蕉，反之亦然。这样的话，你肯定就不能同时以最低的苹果价格和最低的香蕉价格享受到最低折扣了。

按刚才那个思路就会得到错误的结果。因为子问题并不独立，苹果和香蕉的折扣价格无法同时达到最优，这时最优子结构被破坏。

回过头来，我们再读一下最优子结构的定义。首先，你应该已经理解了什么是子问题之间必须相互独立。

其次，所谓后续的计算可以通过前面的状态推导，是指：如果你准备购买了5斤折扣苹果，那么这个价格（即子问题）就被确定了，继续在购物车追加3斤折扣香蕉的订单，只需要在刚才的价格上追加折扣香蕉的价格，就是最低的总价格（即答案）。

现在，让我们回到硬币找零的问题上来，它满足最优子结构吗？满足。

假设有两种面值的硬币 c[0]=5, c[1]=3，目标兑换金额为 k=11。原问题是求这种情况下求最少兑换的硬币数。

如果你知道凑出 k=6 最少硬币数为 “2” （注意，这是一个子问题），那么你只需要再加 “1” 枚面值为 c[0]=5 的硬币就可以得到原问题的答案，即 2 + 1 = 3。

原问题并没有限定硬币数量，你应该可以看出这些子问题之间没有互相制约的情况，它们之间是互相独立的。因此，硬币找零问题满足最优子结构，可以使用动态规划思想来进行求解。

使用动态规划求解硬币找零

当动态规划最终落到实处，就是一个状态转移方程，这同样是一个吓唬人的名词。不过没关系，其实我们已经具备了写出这个方程的所有工具。现在，就让我带你一起看看如何写出这个状态转移方程。

首先，任何穷举算法（包括递归在内）都需要一个终止条件。那么对于硬币找零问题来说，终止条件是什么呢？当剩余的金额为 0 时结束穷举，因为这时不需要任何硬币就已经凑出目标金额了。在动态规划中，我们将其称之为**初始化状态**。

接着，我们按照上面提到的凑硬币的思路，找出子问题与原问题之间会发生变化的变量。原问题指定了硬币的面值，同时没有限定硬币的数量，因此它们俩无法作为“变量”。唯独剩余需要兑换的金额是变化的，因此在这个题目中，唯一的变量是目标兑换金额 k。

在动态规划中，我们将其称之为**状态参数**。同时，你应该注意到了，这个状态在不断逼近初始化状态。而这个不断逼近的过程，叫做状态转移。

接着，既然我们确定了状态，那么什么操作会改变状态，并让它不断逼近初始化状态呢？每当我们挑一枚硬币，用来凑零钱，就会改变状态。在动态规划中，我们将其称之为**决策**。

终于，我们构造了一个初始化状态->确定状态参数->设计决策的思路。现在万事俱备，只欠东风，让我们一起来写这个**状态转移方程**。通常情况下，状态转移方程的参数就是状态转移过程中的变量，即状态参数。而函数的返回值就是答案，在这里是最少兑换的硬币数。

我在这里先用递归形式（伪代码形式）描述一下状态转移的过程，这跟我们在上面讨论的挑硬币的过程是一致的。

```
DP(values, k) {
  res = MAX
  for c in values
    // 作出决策，找到需要硬币最少的那个结果
    res = min(res, 1 + DP(values, k-c)) // 递归调用

  if res == MAX
    return -1

  return res
}
```

顺着这个思路，我把状态转移方程给写出来，它是这样的：

\[F(n)=\\left\\{\\begin{array}{c}- 0,n=0\\\\\\- \-1,n<0\\\\\\- min\\{ 1+DP(n-c) | c \\in values \\}- \\end{array}\\right.\]

还记得吗？我们曾在讲斐波那契数列时展示了一个重要发现，那就是：递归形式的求解几乎就是简单的把题设中的函数表达式照搬过来，因此我们说从数学意义上讲，递归更直观，且易于理解。

递归与动态规划

不知道你有没有发现，上面这个动态规划的状态转移方程，与上一课当中的递归优化方案的方程式几乎一样。

在上一课中，我们为了优化递归中的重叠子问题，设计了一个缓存用于存储重叠子问题的结果，避免重复计算已经计算过的子问题。而这个缓存其实就是存储了动态规划里的状态信息。

因此，带备忘录的递归算法与你现在看到的动态规划解法之间，有着密不可分的关系。它们要解决的核心问题是一样的，即消除重叠子问题的重复计算。

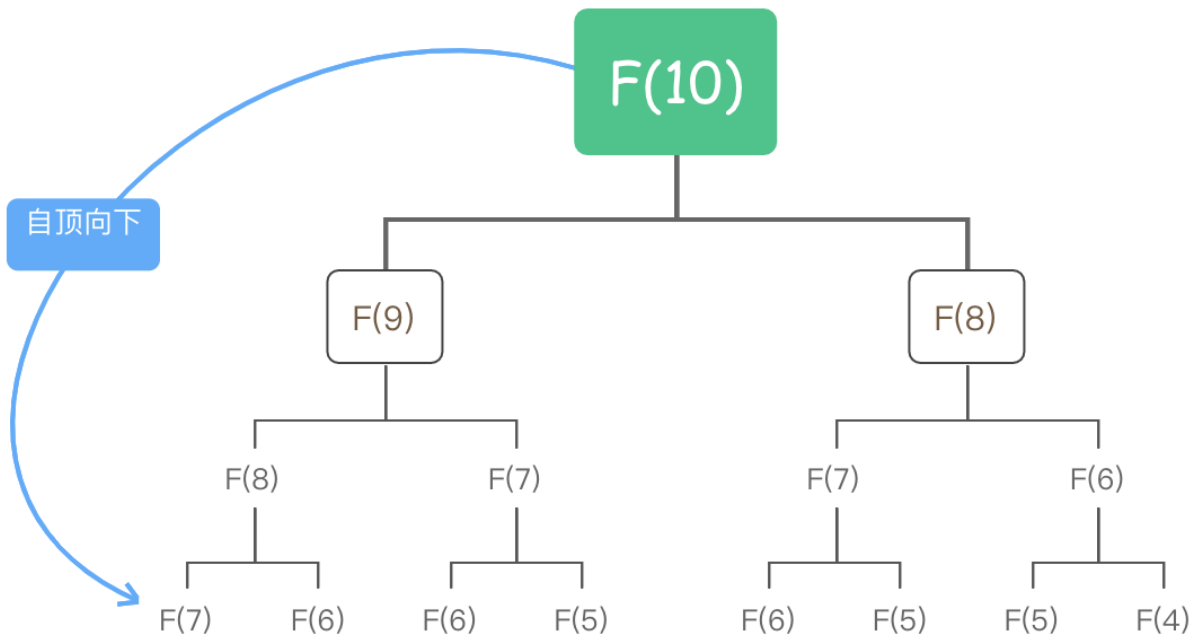
事实上，带备忘录的递归算法也是一种动态规划解法。但是，我们为何不把这种方法作为动态规划面试题的常规解法呢？

这是递归带来的固有问题。

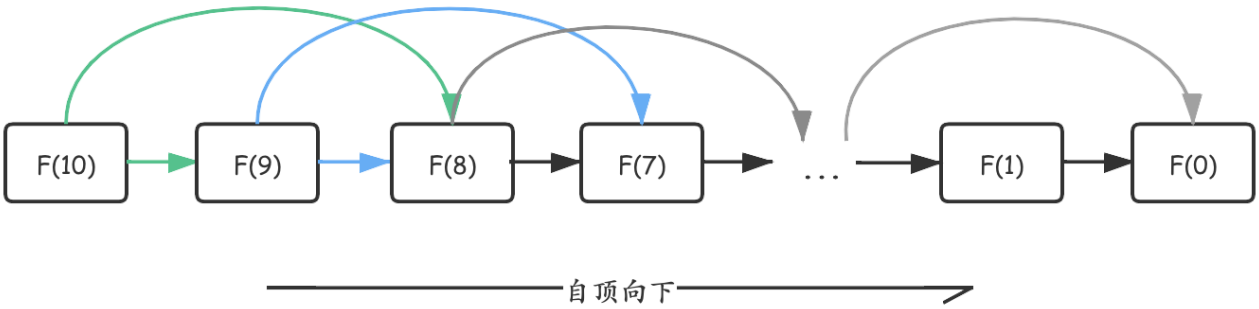
首先，从理论上说，虽然带备忘录的递归算法与动态规划解法的时间复杂度是相同规模的（稍后我就会展示新的动态规划解法），但在计算机编程的世界里，递归是依赖于函数调用的，而每一次函数调用的代价非常高昂。

递归调用是需要基于堆栈才能实现的。而对于基于堆栈的函数调用来说，在每一次调用的时候都会发生环境变量的保存和还原，因此会带来比较高的额外时间成本。这是无法通过时间复杂度分析直接表现出来的。

更重要的是，即便我们不考虑函数调用带来的开销，递归本身的处理方式是**自顶向下**的。所谓自顶向下，是指访问递归树的顺序是自顶向下的，这么说还是很抽象？我把递归处理斐波那契数列的顺序画出来，你就明白了：



如果从解路径的角度看递归的自顶向下处理，那么它的形式可以用由左至右的链表形式表示。



因此每次都需要查询子问题是否已经被计算过，如果该子问题已经被计算过，则直接返回备忘录中的记录。也就是说，在带备忘录的递归解法中，无论如何都要多处理一个分支逻辑，只不过这个分支的子分支是不需要进行处理的。

这样的话，我们就可以预想到，如果遇到子问题分支非常多，那么肉眼可见的额外时间开销在所难免。我们不希望把时间浪费在递归本身带来的性能损耗上。

那么，有什么好的办法来规避这个问题呢？我们需要设计一种新的缓存方式，并考虑使用迭代来替换递归。接下来，让我们来一起看看该如何改造我们的算法。

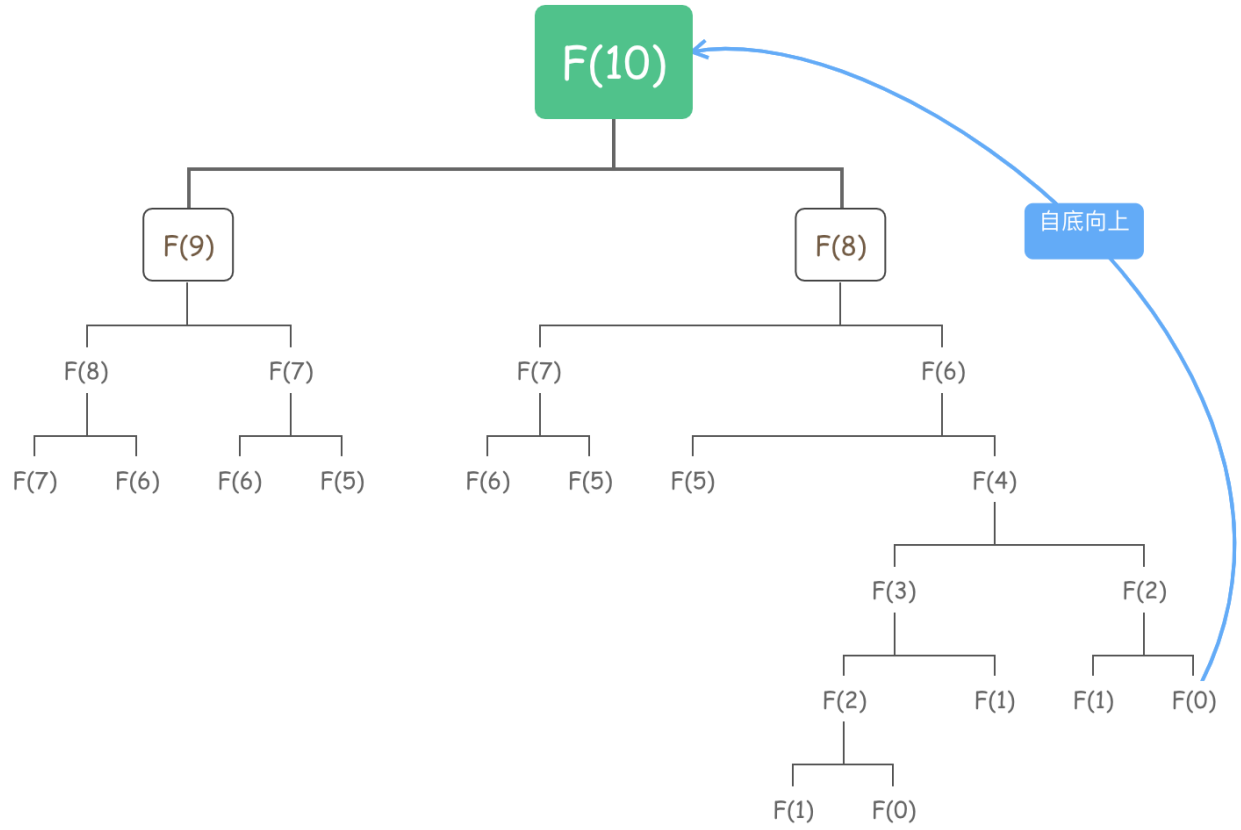
状态缓存与循环

在带备忘录的递归算法中，每次都需要查询子问题是否已经被计算过。针对这一问题，我们可以思考一下，是否有方法可以不去检查子问题的处理情况呢？在执行A问题的时候，确保A的所有子问题一定已经计算完毕了。

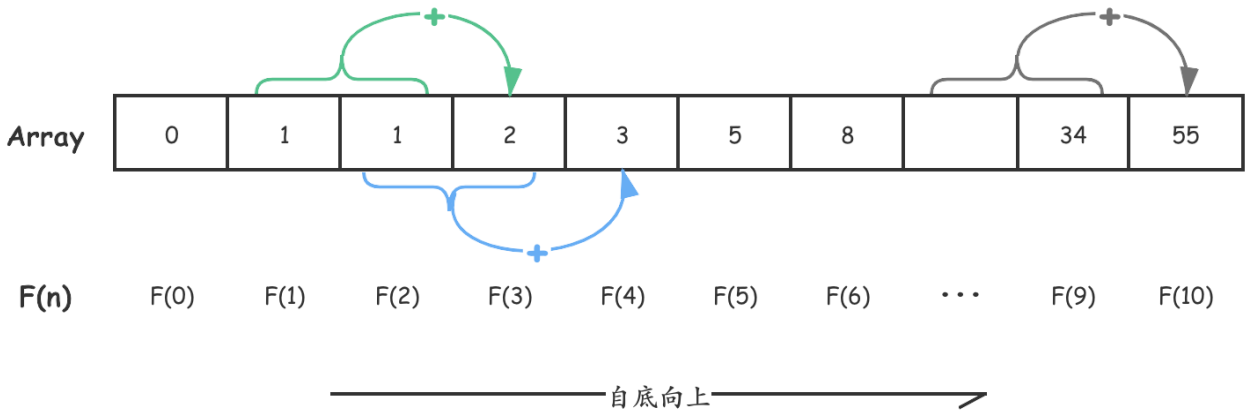
仔细想一想，这不就是把处理方向倒过来用**自底向上**嘛！那么我们具体要怎么做呢？

回顾一下自顶向下的方法，我们的思路是从目标问题开始，不断将大问题拆解成子问题，然后再继续不断拆解子问题，直到子问题不可拆解为止。通过备忘录就可以知道哪些子问题已经被计算过了，从而提升求解速度。

那么如果要自底向上，我们是不是可以首先求出所有的子问题，然后通过底层的子问题向上求解更大的问题。我还是通过斐波那契数列来画出自底向上的处理方式：



如果从解路径的角度看动态规划的自底向上处理方式，那么它的形式可以用一个数组来进行表示，而这个数组事实上就是实际的备忘录存储结构。



这样有一个好处，当求解大问题的时候，我们已经可以确保该问题依赖的所有子问题都已经计算过了，那么我们就无需检查子问题是否已经求解，而是直接从缓存中取出子问题的解。

通过自底向上，我们完美地解决掉了递归中由于“试探”带来的性能损耗。有了思路之后，让我们把上一课中的递归代码做些修改，变成新的迭代实现：

Java 实现：

```java
int getMinCounts(int k, int[] values) {
    int[] memo = new int[k + 1];
    Arrays.fill(memo, -1);
    memo[0] = 0; // 初始化状态

    for (int v = 1; v <= k; v++) {
        int minCount = k + 1; // 模拟无穷大
        for (int i = 0; i < values.length; ++i) {
            int currentValue = values[i];

            // 如果当前面值大于硬币总额，那么跳过
            if (currentValue > v) { continue; }

            // 使用当前面值，得到剩余硬币总额
            int rest = v - currentValue;
            int restCount = memo[rest];

            // 如果返回-1，说明组合不可信，跳过
            if (restCount == -1) { continue; }

            // 保留最小总额
            int kCount = 1 + restCount;
            if (kCount < minCount) { minCount = kCount; }
        }

        // 如果是可用组合，记录结果
        if (minCount != k + 1) { memo[v] = minCount; }
    }

    return memo[k];
}

int getMinCountsDPSol() {
    int[] values = { 3, 5 }; // 硬币面值
    int total = 22; // 总值

    // 求得最小的硬币数量
    return getMinCounts(total, values); // 输出答案
}
```

C++ 实现：

```cpp
int GetMinCounts(int k, const std::vector<int>& values) {
    std::vector<int> memo(k + 1, -1); // 创建备忘录
    memo[0] = 0; // 初始化状态

    for (int v = 1; v <= k; v++) {
        int minCount = k + 1; // 模拟无穷大
        for (int i = 0; i < values.size(); ++i) {
            int currentValue = values[i];

            // 如果当前面值大于硬币总额，那么跳过
```

```
            if (currentValue > v) { continue; }

            // 使用当前面值，得到剩余硬币总额
            int rest = v - currentValue;
            int restCount = memo[rest];

            // 如果返回-1，说明组合不可信，跳过
            if (restCount == -1) { continue; }

            // 保留最小总额
            int kCount = 1 + restCount;
            if (kCount < minCount) { minCount = kCount; }
        }

        // 如果是可用组合，记录结果
        if (minCount != k + 1) { memo[v] = minCount; }
    }

    return memo[k];
}

int GetMinCountsDPSol() {
    std::vector<int> values = { 3, 5 }; // 硬币面值
    int total = 11; // 总值

    // 求得最小的硬币数量
    return GetMinCounts(total, values); // 输出答案
}
```

我们的关注点在GetMinCounts函数上，该函数先定义了一个“新款”状态备忘录，用数组memo来表示（通常将其称之为 DP 数组，DP 是 Dynamic Programming 的缩写即动态规划。你看，是不是高级起来了）。

这个备忘录由数组构成，其定义是：当目标兑换金额为 i 时，至少需要 memo[i] 枚硬币才能凑出。

有了备忘录的定义后，我们接下来再依据状态转移方程的指导来**初始化状态**：

1. 将 F(0) 初始化成 0，即 memo[0]=0；
2. 把备忘录中剩余的位置初始化成 k + 1。凑成金额 k 的硬币数至多只可能等于 k （如果硬币的最低面值是 1），因此初始化为 k + 1 就相当于将这些位置初始化成正无穷大，便于后续**决策**时取最小值。

接着，我们从1开始遍历，求解 F(1) 的结果，直到求解 F(k) 的结果为止。循环结束后我们想要的结果就存储在 memo[k] 中，也就是 F(k) 的解。

在这个基于原来递归代码上改进得到的代码中，我们来看一下每次循环中做了什么。每一次循环都包含一个小循环，这个小循环会遍历所有的面值。

1. 先看当前面额总值是否小于当前硬币面额。如果是，说明组合不存在，直接进入下一轮循环。-
2. 否则，我们就可以认为已经使用了这一枚硬币，那么就求得使用完硬币后的余额 rest，并从备忘录中获取 F(rest) 的结果：

> a. 如果 F(rest) 为 -1，说明 F(rest) 组合不存在，子问题不成立那么当前问题也就无解，直接进入下一轮循环；- b.如果返回的值不是 -1，说明组合存在，那么求 F(rest) + 1，并和当前最小硬币总数比较，取最小值。

1. 内部循环结束后，我们看一下 minCount 的值：

> a. 如果是 -1，说明 F(v) 不存在，那么不做任何处理，保留 F(v)=-1 即可；- b. 否则将最小值存入 memo[v]，表示已经求得f(v)的值，准备为后续的问题使用。

这样我们就通过这种自下而上的方法将递归转换成了循环。但是，这段代码还是跟我们常见的动态规划代码有些出入，不过没有关系，经过简单的调整就可以把它变漂亮。我先给出代码，然后再对其进行解释。

Java 实现：

```
int getMinCounts(int k, int[] values) {
   int[] memo = new int[k + 1]; // 创建备忘录
   memo[0] = 0; // 初始化状态
   for (int i = 1; i < k + 1; i++) { memo[i] = k + 1; }

   for (int i = 1; i < k + 1; i++) {
       for (int coin : values) {
           if (i - coin < 0) { continue; }
           memo[i] = Math.min(memo[i], memo[i - coin] + 1); // 作出决策
       }
   }

   return memo[k] == k + 1 ? -1 : memo[k];
}

int getMinCountsDPSolAdvance() {
   int[] values = { 3, 5 }; // 硬币面值
   int total = 22; // 总值

   return getMinCounts(total, values); // 输出答案
}
```

C++实现：

```
int GetMinCounts(int k, const std::vector<int>& values) {
    int memo[k + 1]; // 创建备忘录
```

```
    memo[0] = 0; // 初始化状态
    for (int i = 1; i < k + 1; i++) { memo[i] = k + 1; }

    for (int i = 1; i < k + 1; i++) {
        for (auto coin : values) {
            if (i - coin < 0) { continue; }
            memo[i] = min(memo[i], memo[i - coin] + 1); // 作出决策
        }
    }

    return memo[k] == k + 1 ? -1 : memo[k];
}

int GetMinCountsDPSolAdvance() {
    std::vector<int> values = { 3, 5 }; // 硬币面值
    int total = 11; // 总值

    return GetMinCounts(total, values); // 输出答案
}
```

现在我们看一下，每一次循环中是如何做**决策**的。每一次循环都包含一个小循环，这个小循环会遍历所有的面值。

1. 跟之前一样，我们先看当前面额总值是否小于当前硬币面额。如果是，则说明组合不存在，直接进入下一轮循环。-
2. 否则，就可以认为已经使用了这一枚硬币，这时我们要作出决策：

> a. 如果采纳了这枚硬币，则凑的硬币数量需要 +1，这时“状态A”是 memo[i - coin] + 1；- b. 如果不采纳这枚硬币，则凑的硬币数量不变，这时“状态B”是 memo[i]；- c. 显然，硬币找零问题是求最值问题（即最少需要几枚硬币凑出总额k）。因此，我们在这里作出决策，在状态A与状态B中谁的硬币数量更少，即取最小值 min(状态A, 状态B)。

1. 当循环结束后，我们看一下备忘录中位置为 k 的值是多少，即 memo[k]：

> a. 如果是 k + 1，就意味着在初始化状态时的值没有被更新过，是“正无穷大”。这时按照题目要求，返回 -1；- b. 否则，我们就找到了最少凑出硬币的数量，返回它，就是我们的答案。

这样一来，借助于自底向上的方法，我们成功的将递归转换成了迭代。

这段代码的时间复杂度是非常标准的 O(m*n)。它不会有任何额外的性能开销，我们通过动态规划完美地解决了硬币找零问题。

通用的动态规划

在掌握了如何使用标准的动态规划来解决硬币找零问题后，我们有必要将其推而广之，来看看解决动态规划面试问题的通用框架，或者说套路。

在这里，我会给出一个经验总结，而非严格的数学推导。

动态规划问题的核心是写出正确的状态转移方程，为了写出它，我们要先确定以下几点：

1. 初始化状态：由于动态规划是根据已经计算好的子问题推广到更大问题上去的，因此我们需要一个“原点”作为计算的开端。在硬币找零问题中，这个初始化状态是 memo[0]=0；
2. 状态：找出子问题与原问题之间会发生变化的变量。在硬币找零问题中，这个状态只有一个，就是剩余的目标兑换金额 k；
3. 决策：改变状态，让状态不断逼近初始化状态的行为。在硬币找零问题中，挑一枚硬币，用来凑零钱，就会改变状态。

一般来说，状态转移方程的**核心参数**就是**状态**。

接着，我们需要自底向上地使用备忘录来消除重叠子问题，构造一个备忘录（在硬币找零问题中，它叫 memo。为了通用，我们以后都将其称之为 DP table）。

最后，我们需要实现决策。在硬币找零问题中，决策是指挑出需要硬币最少的那个结果。通过这样几个简单步骤，我们就能写出状态转移方程：

\[DP(n)=\\left\\{\\begin{array}{c}- 0,n=0\\\\\\- \-1,n<0\\\\\\- min(DP(n), 1+DP(n-c)), c \\in values- \\end{array}\\right.\]

由于是经验，因此它在90%以上的情况下都是有效的，而且易于理解。至于严格的数学推导和状态转移方程框架，我会在后续的课程中给出。

从贪心算法到动态规划

我们从最开始的贪心算法，到暴力递归、带备忘录的递归，通过分析问题最终推导出了动态规划解法。这么做的目的在于，当我们在后续课程中扩展到复杂面试问题时，你仍然能够拥有清晰的核心思路。

到这儿，我们已经完美地解决了硬币找零问题，是时候做个小小的总结了（基于第1～4课）。

首先，贪心算法是根据当前阶段得到局部最优解，然后再看下一个阶段，逐个求解。这样导致的问题就是，我们可能永远无法得到真正的最优解：整体最优解。

为了解决这个问题，我们在贪心算法中加入了回溯的过程。如果无法求解的时候，就会返回，然后重新尝试当前阶段的“局部次优方案”，重新计算这种情况下的解。这样一来，我们至少

保证了所有问题都能求得一个解。

但是如果遇到一些局部最优解前提条件不一定满足全局最优解的情况，这种方法也不一定能让我们找到整体最优解，因为贪心算法里我们找到一个解就结束了，如果约束不足，那么返回可能不一定是整体最优解。

为了解决贪心算法的问题，真正求得整体最优解，我们就必须得到问题解的所有可能组合。这个时候我们就要利用递归来解决问题。

递归就是自顶向下求得满足问题条件的所有组合，并计算这些组合的解，最后从这些组合的解中取出最优解，这样暴力计算出来的结果必定是整体最优解。

但是这样就又出现了效率问题，暴力递归在计算量巨大的情况下，时间复杂度实在太高了，几乎会呈现指数爆炸形式。那么我们就得考虑是否有些问题可以进行剪枝优化。

我提出了一些剪枝优化的方法，重点介绍的就是利用重叠子问题进行优化。在递归求解过程中我们会把一个大问题分解成多个子问题，那些在求解计算分支中可能被反复求解的子问题就是所谓的重叠子问题。

如果这些重叠子问题无后效性，那么我们就可以利用缓存的方法，在求得每个子问题的解之后将求解结果存入缓存数组中。如果在后续的计算分支中遇到相同的子问题，就直接从备忘录中取出我们已经计算过的结果。

这样一来，我们就不需要浪费时间重复求解已经求解的问题，在这种情况下可以将时间复杂度约束在多项式级别。

但是递归求解最后还是会有性能损耗问题，因此这时我正式引入了动态规划。在经历了这些讨论与探索后，你现在应该能够理解动态规划与贪心、回溯、递归的关系了。

课程总结

带备忘录的递归解法从传统意义上说已经是动态规划思想的范畴了，但它使用的是自顶向下的处理方式来解题，它离我们日常看到的动态规划还有差距。

这个差距不仅仅体现在代码的形式上，更重要的是它仍然还不够好：递归本身的性质导致了算法执行时额外的存储开销。

为此，我们正式引入自底向上的一种处理方式，并用迭代代替了递归，实现了较为简洁的硬币找零动归解法。在多项式级别的算法时间复杂度内，我们用最快的速度得到了我们想要的结果。

另外，动态规划的关键是状态转移方程，为了写出它，我们需要按照套路找出以下项目：

1. 初始化状态：由于动态规划是根据已经计算好的子问题推广到更大问题上去的，因此我们需要一个“原点”作为计算的开端；
2. 状态：找出子问题与原问题之间会发生变化的变量，这个变量就是状态转移方程中的参数；
3. 决策：改变状态，让状态不断逼近初始化状态的操作。这个决策，就是状态转移方程状态转移的方向。

最后，我们要将以上信息组装起来，一般来说就可以得到动态规划的状态转移方程了。

课后思考

如果给你一个包含正数和负数的整数数组，你能否找到一个具有最大和的连续子数组，其中子数组最少包含一个元素，返回其最大和。比如输入的子数组是 [-2, 1, -3, 1, -1, 6, 2, -5, 4]，输出是 8，因为连续子数组 [1, -1, 6, 2] 的和最大。请你思考一下，并使用动态规划的方法来求解此问题。

最后，欢迎留言和我分享你的答案，也不妨把问题发给你的好朋友看看，邀请他一起讨论。

上一页 下一页