



下载APP

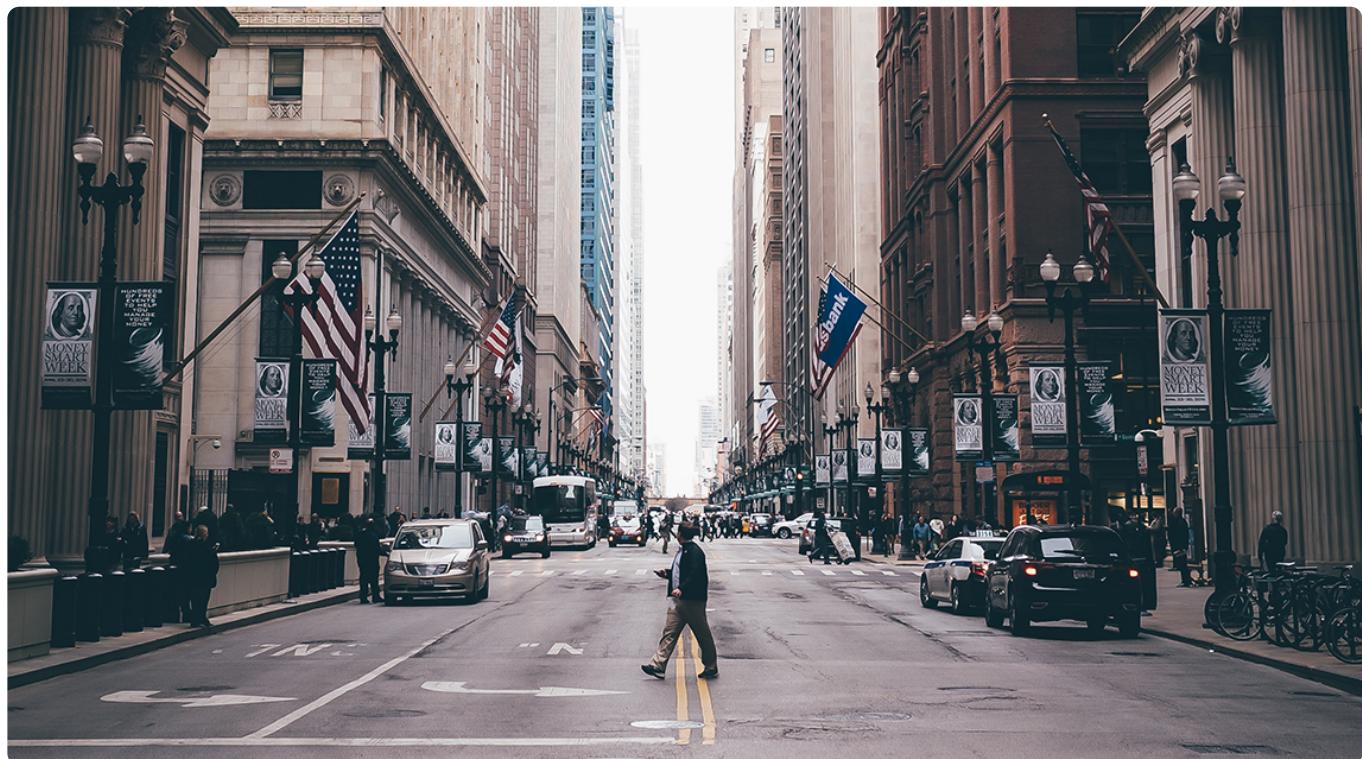


01 | 硬币找零问题：从贪心算法说起

2020-09-14 卢誉声

动态规划面试宝典

[进入课程 >](#)



讲述：卢誉声

时长 09:41 大小 8.88M



你好，我是卢誉声。

作为“初识动态规划”模块的第一节课，我会带着你一起从贪心算法开始了解整个知识体系的脉络。现实中，我们往往不愿意承认自己贪婪。事实上，贪婪是渴望而不知满足，它是人的一种基本驱动力。既然是基本驱动力，那它自然就不会太难。

所以你可能会说贪心算法很简单啊，但其实不然，这里面还真有不少门道值得我们说说。而且，它还跟动态规划问题有着千丝万缕的联系，能够帮助我们理解真正的动归问题。



接下来我们就从一个简单的算法问题开始探讨，那就是硬币找零。在开始前，我先提出一个问题：**任何算法都有它的局限性，贪心算法也如此，那么贪心算法能解决哪些问题呢？**

你不妨带着这个问题来学习下面的内容。

硬币找零问题

移动支付已经成为了我们日常生活当中的主流支付方式，无论是在便利店购买一瓶水，还是在超市或菜市场购买瓜果蔬菜等生活用品，无处不在的二维码让我们的支付操作变得异常便捷。

但在移动支付成为主流支付方式之前，我们常常需要面对一个简单问题，就是找零的问题。

虽然说硬币找零在日常生活中越来越少，但它仍然活跃在编程领域和面试问题当中，主要还是因为它极具代表性，也能多方面考察一个开发人员或面试者解决问题的能力。

既然如此，我们就先来看看这个算法问题的具体描述。

问题：给定 n 种不同面值的硬币，分别记为 $c[0], c[1], c[2], \dots c[n]$ ，同时还有一个总金额 k ，编写一个函数计算出**最少**需要几枚硬币凑出这个金额 k ？每种硬币的个数不限，且如果没有任何一种硬币组合能组成总金额时，返回 -1。

 复制代码

- 1 示例 1:
- 2
- 3 输入: $c[0]=1, c[1]=2, c[2]=5, k=12$
- 4 输出: 3
- 5 解释: $12 = 5 + 5 + 2$

 复制代码

- 1 示例 2:
- 2
- 3 输入: $c[0]=5, k=7$
- 4 输出: -1
- 5 解释: 只有一种面值为5的硬币，怎么都无法凑出总价值为7的零钱。

题目中有一个醒目的提示词，那就是“**最少**”。嗯，看起来这是一个求最值的问题，其实也好理解，如果题目不在这里设定这一条件，那么所求结果就不唯一了。

举个简单的例子，按照示例 1 的题设，有三种不同面值的硬币，分别为 $c_1=1$, $c_2=2$, $c_3=5$ ，在没有“最少”这一前提条件下你能罗列出几种不同的答案？我在这里随意列出几个：

 复制代码

- 1 解1：输出：5，因为 $5 + 2 + 2 + 2 + 1 = 12$ 。
- 2 解2：输出：6，因为 $2 + 2 + 2 + 2 + 2 + 2 = 12$ 。
- 3 解3：输出：12，因为 $1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 = 12$ 。

所以，这是一个求最值的问题。那么求最值的核心问题是什么呢？嗯，无非就是**穷举**，显然，就是把所有可能的凑硬币方法都穷举出来，然后找找看最少需要多少枚硬币，那么最少的凑法，就是这道题目的答案。

在面试中，一般来说穷举从来都不是一个好方法。除非你要的结果就是所有的不同组合，而不是一个最值。但即便是求所有的不同组合，在计算的过程中也仍然会出现重复计算的问题，我们将这种现象称之为**重叠子问题**。

请你记住这个关键概念，它是动态规划当中的一个重要概念。但现在你只需要知道所谓重叠子问题就是：我们在罗列所有可能答案的过程中，可能存在重复计算的情况。我会在后续课程中与你深入探讨这个概念。

在尝试解决硬币找零问题前，我们先用较为严谨的定义来回顾一下贪心算法的概念。

贪心算法

所谓贪心算法，就是指它的每一步计算作出的都是在当前看起来最好的选择，也就是说它所作出的选择只是在某种意义上的局部最优选择，并不从整体最优考虑。在这里，我把这两种选择的思路称作**局部最优解**和**整体最优解**。

因此，我们可以得到贪心算法的基本思路：

1. 根据问题来建立数学模型，一般面试题会定义一个简单模型；
2. 把待求解问题划分成若干个子问题，对每个子问题进行求解，得到子问题的局部最优解；

3. 把子问题的局部最优解进行合并，得到最后基于局部最优解的一个解，即原问题的答案。

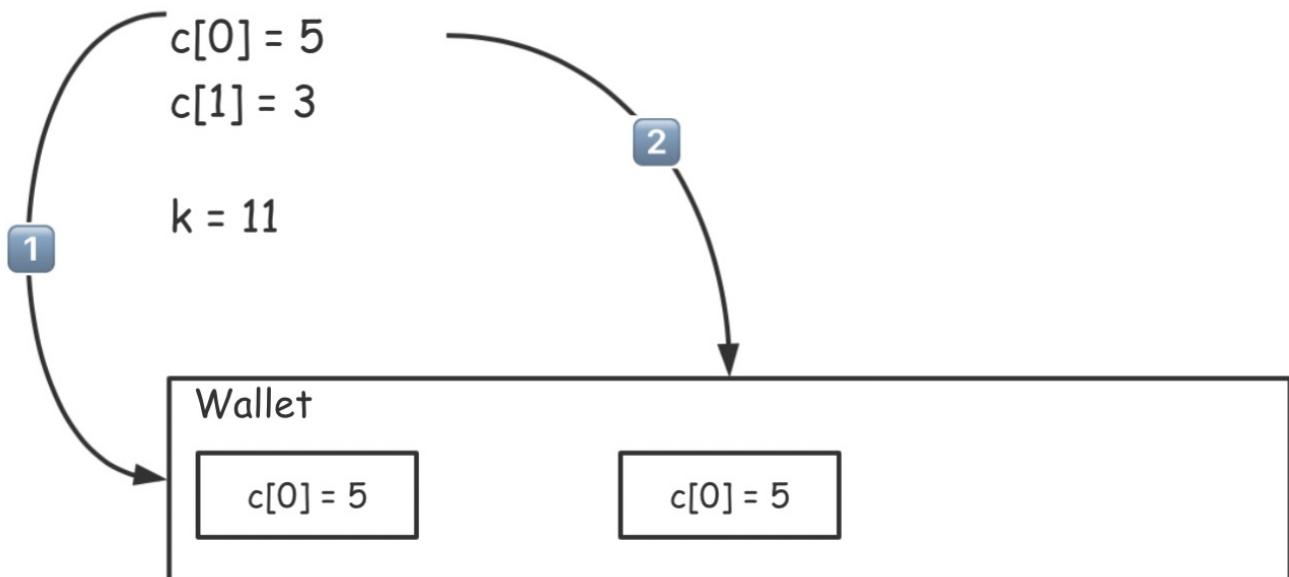
解题思路

现在让我们回到这个问题上来。

既然这道题问的是**最少**需要几枚硬币凑出金额 k ，那么是否可以尝试使用贪心的思想来解这个问题呢？从面值最大的硬币开始兑换，最后得出的硬币总数很有可能就是最少的。

这个想法不错，让我们一起来试一试。

我用一个例子，带你看下整个贪心算法求解的过程，我们从 $c[0]=5, c[1]=3$ 且 $k=11$ 的情况下寻求最少硬币数。按照“贪心原则”，我们先挑选面值最大的，即为 5 的硬币放入钱包。接着，还有 6 元待解（即 $11-5 = 6$ ）。这时，我们再次“贪心”，放入 5 元面值的硬币。



这样来看，贪心算法其实不难吧。我在这里把代码贴出来，你可以结合代码再理解一下算法的执行步骤。

Java 实现：

复制代码

```

1 int getMinCoinCountHelper(int total, int[] values, int valueCount) {
2     int rest = total;
3     int count = 0;
4
5     // 从大到小遍历所有面值
6     for (int i = 0; i < valueCount; ++ i) {
7         int currentCount = rest / values[i]; // 计算当前面值最多能用多少个
8         rest -= currentCount * values[i]; // 计算使用完当前面值后的余额
9         count += currentCount; // 增加当前面额用量
10
11     if (rest == 0) {
12         return count;
13     }
14 }
15
16     return -1; // 如果到这里说明无法凑出总价，返回-1
17 }
18
19
20 int getMinCoinCount() {
21     int[] values = { 5, 3 }; // 硬币面值
22     int total = 11; // 总价
23     return getMinCoinCountHelper(total, values, 2); // 输出结果
24 }
```

C++ 实现：

 复制代码

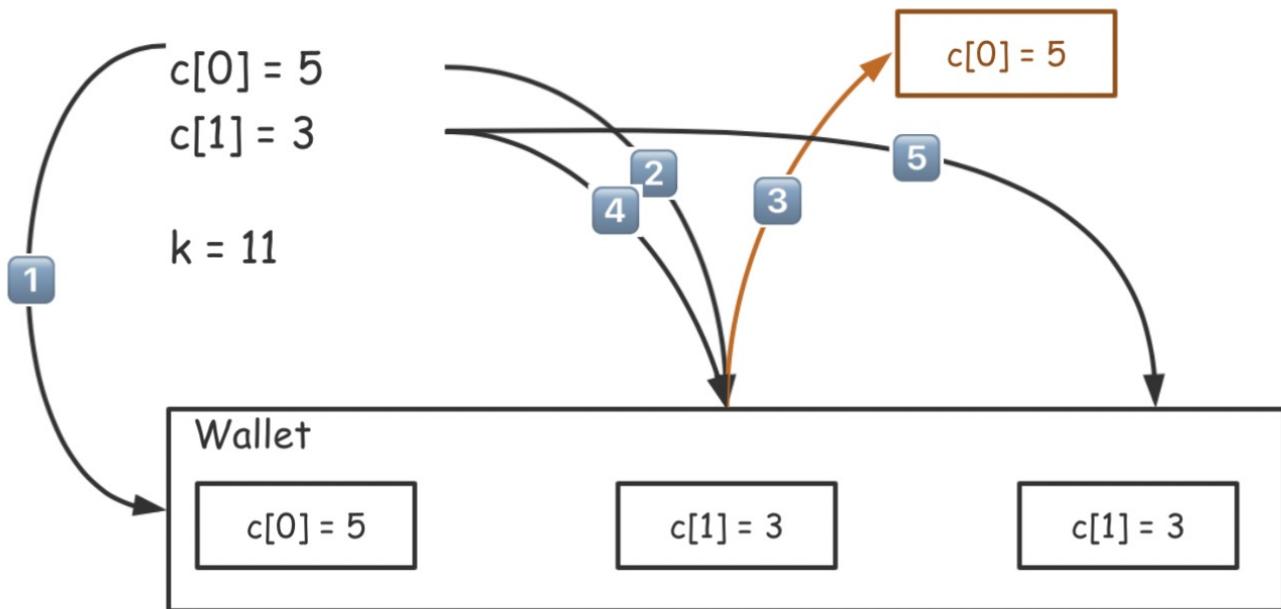
```

1 int GetMinCoinCountHelper(int total, int* values, int valueCount) {
2     int rest = total;
3     int count = 0;
4
5     // 从大到小遍历所有面值
6     for (int i = 0; i < valueCount; ++ i) {
7         int currentCount = rest / values[i]; // 计算当前面值最多能用多少个
8         rest -= currentCount * values[i]; // 计算使用完当前面值后的余额
9         count += currentCount; // 增加当前面额用量
10
11     if (rest == 0) {
12         return count;
13     }
14 }
15
16     return -1; // 如果到这里说明无法凑出总价，返回-1
17 }
18
19 int GetMinCoinCount() {
20     int values[] = { 5, 3 }; // 硬币面值
21     int total = 11; // 总价
22     return GetMinCoinCountHelper(total, values, 2); // 输出结果
23 }
```

这段代码就是简单地从最大的面值开始尝试，每次都会把当前面值的硬币尽量用光，然后才会尝试下一种面值的货币。

嗯。。。你有没有发现问题？那就是还剩 1 元零钱待找，但是我们只有 $c[0]=5, c[1]=3$ 两种面值的硬币，怎么办？这个问题无解了，该返回 -1 了吗？显然不是。

我们把第 2 步放入的 5 元硬币取出，放入面值为 3 元的硬币试试看。这时，你就会发现，我们还剩 3 元零钱待找。



正好我们还有 $c[1]=3$ 的硬币可以使用，因此解是 $c[0]=5, c[1]=3, c[1]=3$ ，即**最少**使用三枚硬币凑出了 $k=11$ 这个金额。

我们对贪心算法做了改进，引入了回溯来解决前面碰到的“过于贪心”的问题。同样地，我把改进后的代码贴在这，你可以再看看跟之前算法实现的区别。

Java 实现：

复制代码

```
1 int getMinCoinCountOfValue(int total, int[] values, int valueIndex) {
2     int valueCount = values.length;
3     if (valueIndex == valueCount) { return Integer.MAX_VALUE; }
```

```
4
5     int minResult = Integer.MAX_VALUE;
6     int currentValue = values[valueIndex];
7     int maxCount = total / currentValue;
8
9     for (int count = maxCount; count >= 0; count --) {
10         int rest = total - count * currentValue;
11
12         // 如果rest为0, 表示余额已除尽, 组合完成
13         if (rest == 0) {
14             minResult = Math.min(minResult, count);
15             break;
16         }
17
18         // 否则尝试用剩余面值求当前余额的硬币总数
19         int restCount = getMinCoinCountOfValue(rest, values, valueIndex + 1);
20
21         // 如果后续没有可用组合
22         if (restCount == Integer.MAX_VALUE) {
23             // 如果当前面值已经为0, 返回-1表示尝试失败
24             if (count == 0) { break; }
25             // 否则尝试把当前面值-1
26             continue;
27         }
28
29         minResult = Math.min(minResult, count + restCount);
30     }
31
32     return minResult;
33 }
34
35 int getMinCoinCountLoop(int total, int[] values, int k) {
36     int minCount = Integer.MAX_VALUE;
37     int valueCount = values.length;
38
39     if (k == valueCount) {
40         return Math.min(minCount, getMinCoinCountOfValue(total, values, 0));
41     }
42
43     for (int i = k; i <= valueCount - 1; i++) {
44         // k位置已经排列好
45         int t = values[k];
46         values[k] = values[i];
47         values[i]=t;
48         minCount = Math.min(minCount, getMinCoinCountLoop(total, values, k + 1
49
50         // 回溯
51         t = values[k];
52         values[k] = values[i];
53         values[i]=t;
54     }
55 }
```

```

56     return minCount;
57 }
58
59 int getMinCoinCountOfValue() {
60     int[] values = { 5, 3 }; // 硬币面值
61     int total = 11; // 总价
62     int minCoin = getMinCoinCountLoop(total, values, 0);
63
64     return (minCoin == Integer.MAX_VALUE) ? -1 : minCoin; // 输出答案
65 }
```

C++ 实现：

[复制代码](#)

```

1 int GetMinCoinCountOfValue(int total, int* values, int valueIndex, int valueCo
2     if (valueIndex == valueCount) { return INT_MAX; }
3
4     int minResult = INT_MAX;
5     int currentValue = values[valueIndex];
6     int maxCount = total / currentValue;
7
8     for (int count = maxCount; count >= 0; count --) {
9         int rest = total - count * currentValue;
10
11         // 如果rest为0，表示余额已除尽，组合完成
12         if (rest == 0) {
13             minResult = min(minResult, count);
14             break;
15         }
16
17         // 否则尝试用剩余面值求当前余额的硬币总数
18         int restCount = GetMinCoinCountOfValue(rest, values, valueIndex + 1, v
19
20         // 如果后续没有可用组合
21         if (restCount == INT_MAX) {
22             // 如果当前面值已经为0，返回-1表示尝试失败
23             if (count == 0) { break; }
24             // 否则尝试把当前面值-1
25             continue;
26         }
27
28         minResult = min(minResult, count + restCount);
29     }
30
31     return minResult;
32 }
33
34 int GetMinCoinCountLoop(int total, int* values, int valueCount, int k) {
35     int minCount = INT_MAX;
```

```

36     if (k == valueCount) {
37         return min(minCount, GetMinCoinCountOfValue(total, values, 0, valueCou
38     }
39
40     for (int i = k; i <= valueCount - 1; i++) {
41         // k位置已经排列好
42         int t = values[k];
43         values[k] = values[i];
44         values[i]=t;
45         minCount = min(minCount, GetMinCoinCountOfValue(total, values, 0, valu
46         minCount = min(minCount, GetMinCoinCountLoop(total, values, valueCount
47
48         // 回溯
49         t = values[k];
50         values[k] = values[i];
51         values[i]=t;
52     }
53
54     return minCount;
55 }
56
57 int GetMinCoinCountOfValue() {
58     int values[] = { 5, 3 }; // 硬币面值
59     int total = 11; // 总价
60     int minCoin = GetMinCoinCountLoop(total, values, 2, 0);
61
62     return (minCoin == INT_MAX) ? -1 : minCoin;
63 }
```

改进后的算法实现在之前的基础上增加上了一个**回溯**过程。简单地说就是多了一个**递归**，不断尝试用更少的当前面值来拼凑。只要有一个组合成功，我们就返回总数，如果所有组合都尝试失败，就返回 -1。

嗯，这样就没问题了，对硬币找零问题来说，我们得到了理想的结果。

贪心算法的局限性

从上面这个例子我们可以看出，如果只是简单采用贪心的思路，那么到用完 2 个 5 元硬币的时候我们就已经黔驴技穷了——因为剩下的 1 元无论如何都没法用现在的硬币凑出来。这是什么问题导致的呢？

这就是贪心算法所谓的**局部最优**导致的问题，因为我们每一步都尽量多地使用面值最大的硬币，因为这样数量肯定最小，但是有的时候我们就进入了死胡同，就好比上面这个例子。

所谓**局部最优**，就是只考虑“当前”的最大利益，既不向前多看一步，也不向后多看一步，导致每次都只用当前阶段的最优解。

那么如果纯粹采用这种策略我们就永远无法达到**整体最优**，也就无法求得题目的答案了。至于能得到答案的情况那就是我们走狗屎运了。

虽然纯粹的贪心算法作用有限，但是这种求解**局部最优**的思路在方向上肯定是对的，毕竟所谓的**整体最优**肯定是从很多个**局部最优**中选择出来的，因此所有最优化问题的基础都是贪心算法。

回到前面的例子，我只不过是在贪心的基础上加入了失败后的回溯，稍微牺牲一点当前利益，仅仅是希望通过下一个硬币面值的**局部最优**达到最终可行的**整体最优**。

所有贪心的思路就是我们最优化求解的根本思想，所有的方法只不过是针对贪心思路的改进和优化而已。回溯解决的是正确性问题，而动态规划则是解决时间复杂度的问题。

贪心算法是求解整体最优的真正思路源头，这就是为什么我们要在课程的一开始就从贪心算法讲起。

课程总结

硬币找零问题本质上是求最值问题。事实上，动态规划问题的一般形式就是求最值，而求最值的核心思想是**穷举**。这是因为只要我们能够找到所有可能的答案，从中挑选出最优的解就是算法问题的结果。

在没有优化的情况下，穷举从来就不算是一个好方法。所以我带你使用了贪心算法来解题，它是一种使用**局部最优**思想解题的算法（即从问题的某一个初始解出发逐步逼近给定的目标，以尽可能快的速度去求得更好的解，当达到算法中的某一步不能再继续前进时，算法停止）。

但是通过硬币找零问题，我们也发现了贪心算法本身的局限性：

1. 不能保证求得的最后解是最佳的；
2. 不能用来求最大或最小解问题；

3. 只能求满足某些约束条件的可行解的范围。

我们往往需要使用**回溯**来优化贪心算法，否则就会导致算法失效。因此，在求解最值问题时，我们需要更好的方法来解。在后面课程讲到递归和穷举优化问题的时候，我会讲到解决最值问题的正确思路和方法：**考虑整体最优**的问题。

课后思考

在递归问题中，回溯是一种经典的优化算法性能的方法。递归对动态规划来说也十分重要。你能否举出使用回溯算法来解的面试问题？并给出你的解。希望你能在课后提出问题，进行练习。

最后，欢迎留言和我分享你的思考，我会第一时间给你反馈。如果今天的内容对你有所启发，也欢迎把它分享给你身边的朋友，邀请他一起学习！

12 人觉得很赞 | [提建议](#)

极客时间 3 周年

做任务 得千元礼包



[【点击】图片，立即参加 >>>](#)

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 [导读 | 动态规划问题纷繁复杂，如何系统学习和掌握它？](#)

下一篇 [02 | 暴力递归：当贪心失效了怎么办？](#)

精选留言 (30)



AshinInfo 置顶

2020-09-16

重新调整得了java代码部分，提高代码的可读性

```
private static void getMinCoinCountOfValue() {  
    // 硬币面值  
    int[] values = {5, 3};  
    // 总价...
```

展开 ▼

作者回复: 赞，感谢你的分享，顶一顶，让更多人看到。

写留言



2

15



sanyinch

2020-09-16

上述回溯+贪心并不能取到最优解，
比如[1,7,10] amount=14
那么根据递归深搜 $10 + 1 + 1 + 1 + 1$ 会比 $7 + 7$ 先到

作者回复: 嗯，这里的代码逻辑需要再加上“回溯时比较保留最小值”，这样就没有问题了。

贪心真的很容易过于贪婪，代码已更新。



梅亮宏@创造力

2020-09-15

老师说的很生动！用递归加局部最优的方法一定能得到正解。但是如果问题变得更加复杂的情况下，例如我们有1亿中硬币可以用，总币值为几万亿。可能还需要优化一下算法性能或者用分布式计算把性能提高？

这让我想到了ai中的reinforcement learning。个人认为有些偏全局优化？就如alphaG...
展开▼

作者回复: 没错，寻找最优解的时候，本质上还是需要进行枚举（穷举），因此优化算法性能是最重要的一步，如果使用了动态规划，就可以大幅压缩计算时间，在此基础上，如果性能仍不满意（即数据规模实在太巨大），那么则可以考虑使用工程化的方法，比如你提到的分布式计算来提高计算性能。

AI 中的强化学习，本质上它的目的是寻找全局最优解。但是，我们最终能找的只是一定范围内的局部最优解，使得这个局部最优解在一定条件下，最接近我们期望的全局最优解。这个发散点很好，算法发展到一定高度就会进入机器学习领域，无论如何，基本的算法思路其实没有本质差别。



Karl

2020-09-14

老师，第一段代码的第22行，是不是应该为调用GetMinCoinCountHelper?

作者回复: 你好 Karl!

感谢你提出此问题。代码已经做了相应调整和更新。并追加了 Java 版本的代码供大家参考。

**好运来**

2020-09-14

测试在原有贪心基础上加上回溯可以找到一组可行解：

```
int[] values = new int[] {5, 3}; // 硬币面值
```

```
int total = 11; // 总价
```

贪心策略求出可行解不是全局最优解：

```
values = new int[]{5, 4, 1}; // 硬币面值...
```

展开 ▼

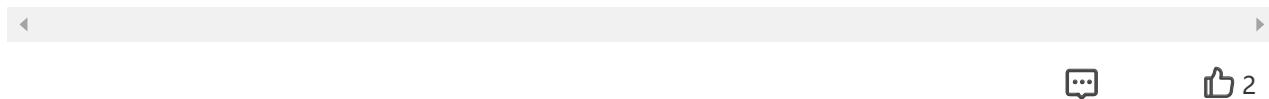
作者回复: 赞。事实上，纯粹的贪心算法只考虑了局部最优的情况，因此在绝大多数情况下是得不到最优解的。在回溯版本中，需要枚举所有的组合，并保留最小的结果。本质上，这里是通过回溯来进行穷举的，因此效率还是不够好。后面的课程会给出更好的解决方法。

**Geek_98ba19**

2020-09-15

C++语法看不懂啊，能否用Java这种绝大多数人的入门语言写例子啊？

作者回复: 提供全方位服务。Java语言描述的代码已经跟随专栏上线，每一段代码都会提供Java + C++两种语言描述。

**KipJiang**

2020-09-15

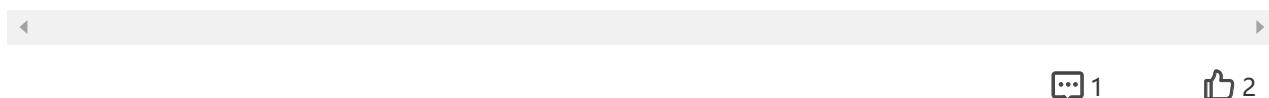
编译、运行通过：

```
#include <iostream>
```

```
int GetMinCoinCountOfValueHelper(int total, int* values, int valueIndex, int value...)
```

展开 ▼

作者回复: 上手实际操作是非常重的，鼓励这种学习的方法哟。



**EncodedStar**

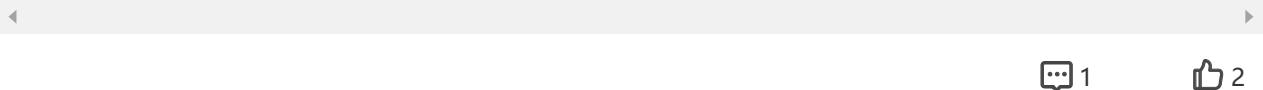
2020-09-14

对动规有了新的了解，感谢老师！

展开 ▼

作者回复: 欢迎进入动归世界。

动态规划其实也不难，是有规可循的。跟随课程，我们一起进步。

**子夜**

2020-09-15

之前对贪心算法的理解是：因为总是局部最优，所以不能用来解决实际问题。学完了这一节，明白了贪心算法的局限性及其应用场景。

展开 ▼

作者回复: 嗯嗯，你的理解是正确的。贪心算法的局限性体现在它在每一步计算中只考虑局部最优解，这导致了它的局限性。对于需要考虑整体最优的问题，我们需要别的方法。

后面的课程就会提到穷举、回溯以及动态规划。

**托尼斯威特**

2020-09-15

原来如此，用搜索解这个题，可以带上贪心的思路

展开 ▼

作者回复: 贪心算法只是在一定条件下，希望能加快搜索的速度。但是，往往在大多数情况下，都不满足这种条件。因此，贪心很难直接得到整体最优解。

贪心算法的本质决定了它能解决问题的范围和高度。如果不辅以别的工具函数或算法，它关注的是局部最优解。

**廖熊猫**

2020-09-15

感觉带有回溯的贪心算法最差的情况应该就是进行了穷举吧..

作者回复: 没错, 是的。

1

1



赵国辉

2020-09-14

老师, 按照本节课的算法, 我的理解是, 由于没有对所有解进行比较。会不会出现找到的解不是最优解呢?

作者回复: 你问的这个问题非常好。没错, 的确会出现这种情况, 所以为了确保最优解, 必须得对所有得到的结果进行比较, 保留最小的那个。

1

1



赵国辉

2020-09-14

老师从最直接的解法入手, 然后指出其局限和不足。然后再针对此问题进行优化和解决。而不是直接给出答案。感觉很自然, 更容易理解问题本质, 很棒。

展开 ▼

作者回复: 感谢你的留言。

咱们以实用为王, 通用思路是从贪心算法开始, 然后考虑是否能用穷举来解决 (如果穷举效率低, 就看能否用回溯来加速穷举), 接着才是备忘录, 最后如果还是不够好, 那么选择动态规划来解决。

1

1



qxz

2020-10-24

这回溯算法的参考代码比较难理解。

展开 ▼

1

1



Everlaa

2020-10-20

两种语言实现, 棒棒哒

展开 ▼

作者回复：感谢你的肯定！一般来说ACM竞赛使用C/C++比较多，但是为了方便大家的理解，提供Java的实现也是顺应趋势。



落曦

2020-10-19

我们上课老师和我们讲贪心比dp还难，很多dp的题目有板子，而贪心不同的问题之间跨度非常的大

作者回复：是的，贪心算法本身的局限性太大，它只能解决局部最优解的问题。如果非得使用贪心算法解决更大规模的问题（比如需要考虑整体最优的情况的题目），那么就必须辅以辅助函数、辅助算法等形式来解决问题。这么做其实得不偿失，针对不同类型的问题，考虑使用不同的算法或思想解决，有的放矢，才是上上策。



夏铭志

2020-10-19

dp本质还是状态转移方程

$dp(n) = \min(dp(n-c[i])+1) \quad i >= 0 \quad \&& \quad i < c.length - 1$

展开 ▼

作者回复：DP函数就是状态转移方程的函数化。



AshinInfo

2020-09-27

递归的目的是求解

回溯+递归的目的是枚举所有组合的解，并取最优解返回

没有回溯，递归只能获得一个解或者无解，获得的解不一定是最优解

递归是一种算法结构，回溯是一种算法思想

一般回溯多用递归实现

展开 ▼

作者回复：是的，回溯就相当于穷举过程中递归的一个补丁（patch）。

另外，你对递归和回溯的理解、比喻，比较恰当。可以这么理解。

**AshinInfo**

2020-09-27

第二段的代码

getMinCoinCountLoop，这个方法有点无法理解

```
minCount = min(minCount, GetMinCoinCountOfValue(total, values, 0, valueCount)); minCount = min(minCount, GetMinCoinCountLoop(total, values, valueCount, k + 1)); // 考虑后一位...
```

展开 ▼

作者回复: 这段代码多余，虽然它的执行与否不影响最终结果，但显然多余且没有必要，已删除。

感谢你认真的发现和反馈！

**Scott**

2020-09-26

每种硬币有[0,maxCount]个之间的选择，选择了第一种硬币后，就递归到下一种选择即可。

```
int getMinCoinCountLoop(int total, int[] values, int startIndex, int currentCount) {  
    int minCount = Integer.MAX_VALUE;...
```

展开 ▼

作者回复: 正解，顶上去。