

课程报告

****

**题 目 动态规划解决最少零钱兑换问题**

学生姓名 成凯

学 号 201833050025

学 院 应用技术学院

专 业 计算机科学与技术

指导教师 庞亚伟

**二Ｏ二二 年 6 月 5 日**

目 录

[1 绪论 2](#_Toc495332927)

[1.1 课题背景 2](#_Toc1413888317)

[1.2 最少零钱兑换的定义 2](#_Toc1314389764)

[1.3 研究内容和创新点 2](#_Toc1931970506)

[2 算法技术及概念 2](#_Toc675551702)

[2.1 动态规划 2](#_Toc251413825)

[2.1.1 定义 2](#_Toc1411823126)

[2.2.2 动态规划的步骤 2](#_Toc964462979)

[3 最少零钱兑换问题 3](#_Toc522720497)

[3.1 问题 3](#_Toc7793202)

[3.2 动态规划 3](#_Toc2131327194)

[3.2.1 解析 3](#_Toc1188917598)

[3.2.2 设计 3](#_Toc1950217898)

[3.3 广度优先搜索 3](#_Toc269307525)

[3.3.1 解析 3](#_Toc1503528446)

[3.3.2 设计 4](#_Toc362517673)

[3.4 贪心算法+BFS 6](#_Toc423423572)

[3.4.1 解析 6](#_Toc1866652093)

[3.4.2 设计 6](#_Toc233128028)

[3.5 暴力破解 8](#_Toc1172594468)

[3.5.1 普通递归 8](#_Toc337795157)

[3.5.2 备忘录递归 9](#_Toc1523924678)

[4 算法对比 10](#_Toc1712089024)

[4.1 动态规划和分治区别 10](#_Toc946840215)

[4.2 算法性能比较 10](#_Toc689669235)

[4.3 算法实验及结果分析 10](#_Toc1301589786)

[5 总结 10](#_Toc1551104960)

[致谢 12](#_Toc1117071787)

[附录-完整代码 13](#_Toc1323482035)

动态规划解决最少零钱兑换问题

成凯

南京信息工程大学 应用技术学院，江苏 南京 210044

摘要：本文主要内容是基于算法设计与分析课程要求，通过动态规划算法设计方法对最少零钱兑换问题进行求解。找零问题解决了找到最小数量的硬币（某些面额）加起来等于给定金额的问题。它是整数背包问题的一个特例，其应用范围不仅限于货币。它也是硬币找零问题的最常见变体，这是一种划分的一般情况，在这种情况下，给定无限组硬币的可用面额，目标是找出针对特定硬币进行找零的可能方式的数量。金额，不考虑硬币的顺序。

关键词：动态规划、算法、最少零钱兑换。

Dynamic programming to solve the least change exchange problem

KaiCheng

Nanjing University of Information and Science Technology Applied Technology College

Nanjing Jiangsu 210044

**Abstract：**The main content of this paper is to solve the minimum change exchange problem through the dynamic programming algorithm design method based on the course requirements of algorithm design and analysis. The change problem solves the problem of finding the smallest number of coins (certain denominations) that add up to a given amount. It is a special case of the integer knapsack problem, and its application is not limited to money. It is also the most common variant of the coin change problem, which is a general case of partitioning where, given the available denominations of an infinite set of coins, the goal is to find out the possible ways of making change for a particular coin quantity. Amount, regardless of the order of coins.

**Key Words：**Dynamic programming, algorithm, Minimum change exchange.

# 

# 1 绪论

## 1.1 课题背景

动态规划 (DP) 是一种通过将优化问题分解为更简单的子问题并利用整体问题的最优解取决于其子问题的最优解这一事实来解决优化问题的算法技术。

动态规划主要是对普通递归的优化。无论我们在哪里看到重复调用相同输入的递归解决方案，我们都可以使用动态编程对其进行优化。这个想法是简单地存储子问题的结果，这样我们就不必在以后需要时重新计算它们。这种简单的优化将时间复杂度从指数降低到多项式。例如如果我们为斐波那契数编写简单的递归解决方案，我们会得到指数时间复杂度，如果我们通过存储子问题的解决方案来优化它，时间复杂度会降低到线性。

动态规划可以很好的用来解决最优子结构和重叠子问题。

本文将通过动态规划、贪心算法等方法对最少零钱兑换问题进行求解。

## 1.2 最少零钱兑换的定义

给定一个整数数组coins，表示不同面额的硬币，假设每种硬币的数量都是无限的；以及一个目标整数amount，表示总金额。第一种，我们需要计算并返回可以凑成总金额所需的最少的硬币个数。还有一类变种，即计算并返回可以凑成总金额的硬币组合数。

## 1.3 研究内容和创新点

动态规划问题的求解的两种方式：

1. 自顶向下的备忘录法；
2. 自底向上的动态规划；

理解动态规划的基本步骤：将问题拆分成多阶段子问题，然后选择状态变量，使其满足无后效性；确定决策集合；确定状态转移方程、分离指标函数。

对于求解最少零钱兑换问题其实有多种解决方法，例如贪心算法+BFS、广度优先搜索算法、动态规划算法。本文将以动态规划算法求解最少零钱兑换问题作为重点。

# 2 算法技术及概念

## 2.1 动态规划

### 2.1.1 定义

把多阶段过程转化为一系列单阶段问题，利用各阶段之间的关系，逐个求解，创立了解决这类过程优化问题的新方法。动态规划算法（Dynamic Programming-DP）是通过拆分问题，定义问题状态和状态之间的关系，将待求解的问题分解为若干个子问题（阶段），按顺序求解子阶段，前一子问题的解，为后一子问题的求解提供了有用的信息。在求解任一子问题时，列出各种可能的局部解，通过决策选取那些有可能达到最优的局部解。依次解决各子问题，最后一个子问题就是初始问题的解。

### 2.2.2 动态规划的步骤

1）定义子问题：子问题是和原问题相似，但规模较小的问题。第一步就是缩小规模，利用小规模子问题刻画其结构特征。

2）写出子问题的递推关系：找出状态转换方程，也就是说找到每个状态跟他上一个状态的关系，根据状态转化方程写出代码。

3）确定 DP 数组的计算顺序：在确定了子问题的递推关系之后，下一步就是依次计算出这些子问题了。一般地，动态规划有两种计算顺序：自顶向下的、使用备忘录的递归方法；自底向上的、使用dp数组的循环方法。

4）空间优化（可选）

# 3 最少零钱兑换问题

## 3.1 问题

给定一个整数数组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。设定每种硬币的数量是无限的。例如输入: coins = [1, 2, 5], amount = 11，输出: 3，解释: 11 = 5 + 5 + 1

## 3.2 动态规划

### 3.2.1 解析

动态规划是这道题最常见的解法。利用递归的思想，要想知道凑成零钱 amount 的最少硬币数量，只需要知道所有 amount - coin[i] 对应的零钱需要的最少硬币数量，然后求出最少的那个数量再加上 1 即可。然后因为中间的零钱数量可能重复，因此需要采用一个数组保存中间结果。

### 3.2.2 设计

具体解法如下：

class Solution:

    def coinChange(self, coins: List[int], amount: int) -> int:

        if amount <= 0: return 0

# dp 数组存储中间结果，初始化为最大值

        dp = [(amount + 1)] \* (amount + 1)

        dp[0] = 0

# 从 1 开始计算各个零钱数需要的硬币数量

        for i in range(1, amount + 1):

            for c in coins:

                if i - c >= 0:

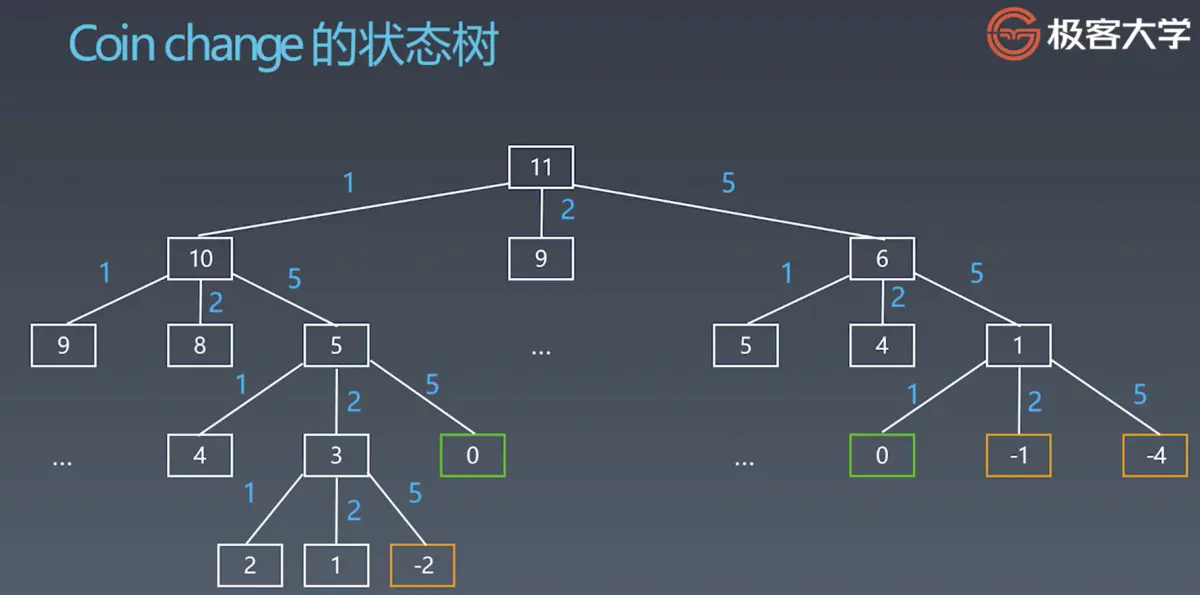
                    dp[i] = min(dp[i], dp[i - c] + 1)

        return -1 if dp[-1] == amount + 1 else dp[-1]

## 3.3 广度优先搜索

### 3.3.1 解析

我们还可以使用广度优先搜索（BFS）。如下图所示，我们可以将零钱根据硬币面额展开成一棵多叉树，其中多叉树的每个分支对应硬币的面额。然后使用广度优先搜索，当扩展出来的节点值为 0 时，表示已经完成兑换，多叉树当前的层次就是结果。



### 3.3.2 设计

解法1，代码如下：

class Solution:

    def coinChange(self, coins: List[int], amount: int) -> int:

        if amount <= 0: return 0

        count = 0

# 从 amount 开始递减

        value1, value2 = [amount], []

        visited = [False] \* amount + [True]

        while value1:

            count += 1

            for val in value1:

                for coin in coins:

                    newval = val - coin

                    if newval >= 0:

                        if not visited[newval]:

    # 终止条件

                            if newval == 0:

                                return count

                            visited[newval] = True

                            value2.append(newval)

            value1, value2 = value2, []

        return -1

解法2，代码如下：

class Solution:

    def coinChange(self, coins: List[int], amount: int) -> int:

        if amount <= 0: return 0

        count = 0

        # 从 0 开始累加

        value1, value2 = [0], []

        visited = [True] + [False] \* amount

        while value1:

            count += 1

            for val in value1:

                for coin in coins:

                    newval = val + coin

                    if newval <= amount:

                        if not visited[newval]:

# 终止条件

                            if newval == amount:

                                return count

                            visited[newval] = True

                            value2.append(newval)

            value1, value2 = value2, []

        return -1

## 3.4 贪心算法+BFS

### 3.4.1 解析

贪心法的思路是每次都优先选择可以选择的最大面额的硬币。比如硬币的面额分别是：[1, 2, 5]；需要兑换的零钱是：11。按照贪心法的思路，优先选择 2个面额为 5 的硬币，然后剩下的零钱是 11 - 2 \* 5 = 1。然后再选择 1 个面额为 1 的硬币，剩下的零钱是 1 - 1 \* 1 = 0。最终得到的结果是 3。可以验证答案是正确的。贪心法的思路很直观，而且大部分情况下能够正确解决问题。但可惜对于某些特殊情况，贪心法得到的答案是错误的。比如硬币的面额分别是：[1, 7, 10]；需要兑换的零钱是：14。按照贪心法的思路，优先选择 1 个面额为 10 的硬币，然后剩下的零钱是 14 - 10 = 4，只能选择 4 个面额为 1 的硬币。最终得到的结果是需要 5 个硬币。可以验证，这不是正确答案。正确答案是选择 2 个面额为 7 的硬币。贪心法思路简单、直接，但不能保证答案是正确的。我们可以使用贪心法的思想，每次优先选择可能的最大面额的硬币。但在此之上，为了保证得到正确解，我们需要按深度优先搜索的方法，遍历整个递归状态树。不过，我们可以加上一些剪枝条件，加快搜索速度。

### 3.4.2 设计

具体C++解法如下：

class Solution {

public:

    int coinChange(vector<int>& coins, int amount) {

        if (amount <= 0 || coins.empty())   return 0;

        // 对币值按照从大到小的顺序排序

        sort(coins.rbegin(), coins.rend());

        int ans = INT\_MAX;

        backtrack(coins, amount, 0, 0, ans);

        return ans == INT\_MAX ? -1 : ans;

    }

private:

    void backtrack(vector<int> &coins, int amount, int idx, int count, int & ans) {

        // 递归终止条件

        if (amount == 0) {

            ans = min(count, ans);  // 保证得到的是正确答案

            return;

        }

        if (idx == coins.size())    return;

        // count + k < ans 剪枝条件

        for (int k = amount / coins[idx]; k >= 0 && count + k < ans; k--)

            // k \* coins[idx] 加快搜索速度

            backtrack(coins, amount - k \* coins[idx], idx + 1, count + k, ans);

    }

};

另Python解法如下：

class Solution:

    def coinChange(self, coins: List[int], amount: int) -> int:

        if amount <= 0 or not coins: return 0

// 对币值按照从大到小的顺序排序

        coins.sort(reverse=True)

        self.res = amount + 1

        self.dfs(coins, amount, 0, 0)

        return -1 if self.res == amount + 1 else self.res

    def dfs(self, coins, amount, idx, count):

// 递归终止条件

        if amount == 0:

            self.res = min(self.res, count)  // 保证得到的是正确答案

            return

        if idx >= len(coins):

            return

        k = amount // coins[idx]

    // count + k < ans 剪枝条件

        while k >= 0 and count + k < self.res:

// k \* coins[idx] 加快搜索速度

            self.dfs(coins, amount - k \* coins[idx], idx + 1, count + k)

            k -= 1

## 3.5 暴力破解

### 3.5.1 普通递归

暴力破解即穷举，把各种可能组成总金额的情况都匹配一遍，得到所有满足的组合，然后取硬币数量最少的那组。遍历出所有的可能性，然后寻找出它的最优解，暴力解法，这种解法会导致超时。

实现思路：剩余金额减去当前使用的硬币金额；如果大于0，继续使用硬币来组合；如果等于0，匹配完成，将当前组使用的硬币数与最小组合硬币数对比，取较小者；如果小于0，直接淘汰。代码实现如下：

def coinChange(coins,amount):

#如果当前的金额数为0，那么就不需要硬币了

if amount==0:

return 0

#如果当前的金额数为负数，那么久凑不出来

if amount<0:

return -1

#我们设置一个非常非常大的数，来比较和第一个遍历情况所需硬币数的大小

#这里可能难以理解

#是这样的，比如你的第一种情况是4个，但是你又不好保存下来，因为它在递归树里面，所以用一个超大的数把第一个结果保存

#之后再不停得递归，永远和当前最小值进行比较

#就可以获得全局最小值

res=10000000

#开始递归面值

for i in coins:

#这个递归的详细解读就是，先遍历到最底部，让amount慢慢长大

#在amount长大的过程中如果刚好有这样的面额，就把它赋值给subProlem

#比如在i==1的时候，我们amount也是1，那么就返回0，表示不需要额外的硬币了

#比如在i==1的时候我们的amount是2，这时候subProblem就是2了

#但是在我们遍历i==2的时候，这个时候的amount还是2，但是这个时候的subProblem 就是0了

#明显最优解是给面值为2的钞票

subProblem=coinChange(coins,amount-i)

# subProblem为-1的情况就是实在是没法给钞票的情况

if subProblem==-1:

continue

#这里需要额外加1，因为一开始我们都是用返回值为0表示不需要额外的硬币，但是这个时候已经花了硬币除去了

res=min(res,subProblem+1)

if res==10000000:

return -1

else:

return res

print(coinChange([1,2,5],11))

### 3.5.2 备忘录递归

通过使用备忘录解法，来适当的减少时间复杂度

思想与代码实现如下：

def coinChange(coins,amount):

#全局变量，备忘录

global memo

#将备忘录所有的值都设置成一个比较“离谱”的值

memo=[-100 for i in range(amount+1)]

return dp(coins,amount)

def dp(coins,amount):

if amount==0:

return 0

if amount<0:

return -1

#如果备忘录里面有数据，就读取备忘录

if memo[amount]!=-100:

return memo[amount]

res=10\*\*10

for i in coins:

subProblem=dp(coins,amount-i)

if subProblem==-1:

continue

res=min(res,subProblem+1)

#把结果存到备忘录

if res==10\*\*10:

memo[amount]=-1

else:

memo[amount]=res

return memo[amount]

#这里备忘录的作用就是记录功能

#可以说，比如我有amount==11的时候

#但是我肯定要经历amount==3的时候

#这个时候备忘录记录这我amount==3的时候的最小硬币个数是2，我就可以直接用了，不需要再递归遍历

#再比如我amount==5，这个时候备忘录里面记录了1,2,3,4没有记录5，这个5就要被记录下来

print(coinChange([1,2,5],11))

# 4 算法对比

### 4.1 动态规划和分治区别

动态规划算法：它通常用于求解具有某种最优性质的问题。在这类问题中，可能会有许多可行解。每一个解都对应于一个值，我们希望找到具有最优值的解。动态规划算法与分治法类似，其基本思想也是将待求解问题分解成若干个子问题，先求解子问题，然后从这些子问题的解得到原问题的解。与分治法不同的是，适合于用动态规划求解的问题，经分解得到子问题往往不是互相独立的。

分治法：若用分治法来解这类问题，则分解得到的子问题数目太多，有些子问题被重复计算了很多次。如果我们能够保存已解决的子问题的答案，而在需要时再找出已求得的答案，这样就可以避免大量的重复计算，节省时间。我们可以用一个表来记录所有已解的子问题的答案。

分治算法不强调记录算过的数据，动态规划为了避免重复计算，一定会记录数据。

### 4.2 算法性能比较

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 算法 | 时间复杂度 | 空间复杂度 | 执行用时 | 内存消耗 |
| 记忆搜索递归 | O(S^n) | O(S) | 68ms | 17.2MB |
| DFS | O(nm) | O(S) | 12ms | 41MB |
| 贪心+BFS | O(Sn) | O(S) | 10ms | 30MB |
| 动态规划 | O(Sn) | O(S) | 16ms | 41.1MB |

### 4.3 算法实验及结果分析

解决零钱兑换问题，至少有3种完全不同的解法。递归解法是自顶向下的思路，动态规划的自底向上的思路，两者殊途同归。不过可以看出，最常见的动态规划解法其实并不是最优的。以我的提交情况来看，“贪心法+DFS”优于 BFS，BFS 优于动态规划。

# 5 总结

本文通过对最少零钱兑换求解。通过暴力解法、动态规划解法、贪心算法解法等算法设计的技巧求解问题，深入了解了动态规划算法等高级算法的解体步骤及其思路。

对最少零钱兑换问题算法中用到了动态规划的解法，动态规划常常适用于有重叠子问题和最优子结构性质的问题，动态规划方法所耗时间往往远少于朴素解法。动态规划背后的基本思想非常简单。大致上，若要解一个给定问题，我们需要解其不同部分（即子问题），再根据子问题的解以得出原问题的解。通常许多子问题非常相似，为此动态规划法试图仅仅解决每个子问题一次，从而减少计算量：一旦某个给定子问题的解已经算出，则将其[记忆化](https://zh.wikipedia.org/wiki/%E8%AE%B0%E5%BF%86%E5%8C%96)存储，以便下次需要同一个子问题解之时直接查表。这种做法在重复子问题的数目关于输入的规模呈[指数增长](https://zh.wikipedia.org/wiki/%E6%8C%87%E6%95%B8%E5%A2%9E%E9%95%B7)时特别有用。

同时也用到了贪心算法来对问题进行求解，对于某些计算问题而言，贪心算法（又称贪婪算法）是指，在对问题求解时，总是做出在当前看来是最好的选择。也就是说，不从整体最优上加以考虑，他所做出的是在某种意义上的局部最优解。贪心选择是指所求问题的整体最优解可以通过一系列局部最优的选择，即贪心选择来达到。这是贪心算法可行的第一个基本要素。当一个问题的最优解包含其子问题的最优解时，称此问题具有最优子结构性质。运用贪心策略在每一次转化时都取得了最优解。问题的最优子结构性质是该问题可用贪心算法求解的关键特征。贪心算法的每一次操作都对结果产生直接影响。贪心算法对每个子问题的解决方案都做出选择，不能回退。贪心算法的基本思路是从问题的某一个初始解出发一步一步地进行，根据某个优化测度，每一步都要确保能获得局部最优解。每一步只考虑一个数据，他的选取应该满足局部优化的条件。若下一个数据和部分最优解连在一起不再是可行解时，就不把该数据添加到部分解中，直到把所有数据枚举完，或者不能再添加算法停止。实际上，贪心算法适用的情贪心算法(贪婪算法)况很少。一般对一个问题分析是否适用于贪心算法，可以先选择该问题下的几个实际数据进行分析，就可以做出判断。

在计算计科学与技术专业中，算法设计与分析是一门非常重要的课程，很多人为它如痴如醉。很多问题的解决，程序的编写都要依赖它，在软件还是面向过程的阶段，就有程序=算法+数据结构这个公式。算法的学习对于培养一个人的逻辑思维能力是有极大帮助的，这门课培养了我思考分析问题，解决问题的能力。

如果一个算法有缺陷，或不适合某个问题，执行这个算法将不会解决这个问题。不同的算法可能用不同的时间、空间或效率来完成同样的任务。一个算法的优劣可以用空间复杂性和时间复杂度来衡量。算法可以使用自然语言、伪代码、流程图等多种不同的方法来描述。计算机系统中的操作系统、语言编译系统、数据库管理系统以及各种各样的计算机应用系统中的软件，都必须使用具体的算法来实现。算法设计与分析是计算机科学与技术的一个核心问题。因此，学习算法无疑会增强自己的竞争力，提高自己的修为，为自己增彩。

# 致谢

感谢南京信息工程大学庞亚伟老师的悉心指导。

# 附录-完整代码

暴力解法：

def coinChange(coins,amount):

if amount==0:

return 0

if amount<0:

return -1

res=10000000

for i in coins:

subProblem=coinChange(coins,amount-i)

if subProblem==-1:

continue

res=min(res,subProblem+1)

if res==10000000:

return -1

else:

return res

print(coinChange([1,2,5],11))

备忘录递归解法：

def coinChange(coins,amount):

global memo

memo=[-100 for i in range(amount+1)]

return dp(coins,amount)

def dp(coins,amount):

if amount==0:

return 0

if amount<0:

return -1

if memo[amount]!=-100:

return memo[amount]

res=10\*\*10

for i in coins:

subProblem=dp(coins,amount-i)

if subProblem==-1:

continue

res=min(res,subProblem+1)

if res==10\*\*10:

memo[amount]=-1

else:

memo[amount]=res

return memo[amount]

print(coinChange([1,2,5],11))

动态规划：

def coinChange(coins,amount):

dp=[amount+1 for \_ in range(amount+1)]

dp[0]=0

for i in range(len(dp)):

for coin in coins:

if i-coin<0:

continue

dp[i]=min(dp[i],1+dp[i-coin])

if dp[amount]==amount+1:

return -1

else:

return dp[amount]

print(coinChange([1,2,5],11))

贪心算法：

class Solution {

public:

    int coinChange(vector<int>& coins, int amount) {

        if (amount <= 0 || coins.empty())   return 0;

        sort(coins.rbegin(), coins.rend());

        int ans = INT\_MAX;

        backtrack(coins, amount, 0, 0, ans);

        return ans == INT\_MAX ? -1 : ans;

    }

private:

    void backtrack(vector<int> &coins, int amount, int idx, int count, int & ans) {

        if (amount == 0) {

            ans = min(count, ans);

            return;

        }

        if (idx == coins.size())    return;

        for (int k = amount / coins[idx]; k >= 0 && count + k < ans; k--)

            backtrack(coins, amount - k \* coins[idx], idx + 1, count + k, ans);

    }

};