10讲动态规划（下）：如何求得状态转移方程并进行编程实现

你好，我是黄申。

上一节，我从查询推荐的业务需求出发，介绍了编辑距离的概念，今天我们要基于此，来获得状态转移方程，然后才能进行实际的编码实现。

状态转移方程和编程实现

上一节我讲到了使用状态转移表来展示各个子串之间的关系，以及编辑距离的推导。不过，我没有完成那张表格。现在我把它补全，你可以和我的结果对照一下。

	空B	m	o	u	s	e
空A	0	1	2	3	4	5
m	1	min(2, 2, 0)=0	min(3, 1, 2)=1	min(4,2,3)=2	min(5,3,4)=3	min(6,4,5)=4
o	2	min(1, 3, 2)=1	min(2, 3, 0)=0	min(3,1,2)=1	min(4,2,3)=2	min(5,3,4)=3
u	3	min(2, 4, 3)=2	min(1, 3, 2)=1	min(2,2,0)=0	min(3,1,2)=1	min(4,2,3)=2
u	4	min(3, 5, 4)=3	min(2, 4, 3)=2	min(1,3,1)=1	min(2,2,1)=1	min(3,2,2)=2
s	5	min(4, 6, 5)=4	min(3, 5, 4)=3	min(2,4,3)=2	min(2,3,1)=1	min(3,2,2)=2
e	6	min(5, 7, 6)=5	min(4,6,5)=4	min(3,5,4)=3	min(2,4,3)=2	min(3,3,1)=1

这里面求最小值的min函数里有三个参数，分别对应我们上节讲的三种情况的编辑距离，分别是：替换、插入和删除字符。在表格的右下角我标出了两个字符串的编辑距离1。

概念和分析过程你都理解了，作为程序员，最终还是要落脚在编码上，我这里带你做些编码前的准备工作。

我们假设字符数组A[]和B[]分别表示字符串A和B，A[i]表示字符串A中第i个位置的字符，B[i]表示字符串B中第i个位置的字符。二维数组d[,]表示刚刚用于推导的二维表格，而d[i,j]表示这张表格中第i行、第j列求得的最终编辑距离。函数r(i, j)表示替换时产生的编辑距离。如果A[i]和B[j]相同，函数的返回值为0，否则返回值为1。

有了这些定义，下面我们用迭代来表达上述的推导过程。

- 如果i为0，且j也为0，那么d[i, j]为0。
- 如果i为0，且j大于0，那么d[i, j]为j。
- 如果i大于0，且j为0，那么d[i, j]为i。
- 如果i大于0，且 j大于0，那么d[i, j]=min(d[i-1, j] + 1, d[i, j-1] + 1, d[i-1, j-1] + r(i, j))。

这里面最关键的一步是d[i, j]=min(d[i-1, j] + 1, d[i, j-1] + 1, d[i-1, j-1] + r(i, j))。这个表达式表示的是动态规划中从上一个状态到下一个状态之间可能存在的一些变化，以及基于这些变化的最终决策结果。我们把这样的表达式称为**状态转移方程**。我上节最开始就说过，在所有动态规划的解法中，状态转移方程是关键，所以你一定要掌握它。

有了状态转移方程，我们就可以很清晰地用数学的方式，来描述状态转移及其对应的决策过程，而且，有了状态转移方程，具体的编码其实就很容易了。基于编辑距离的状态转移方程，我在这里列出了一种编码的实现，你可以看看。

我们首先要定义函数的参数和返回值，你需要注意判断一下a和b为null的情况。

```
public class Lesson10_1 {

/**
   * @Description: 使用状态转移方程，计算两个字符串之间的编辑距离
   * @param a-第一个字符串，b-第二个字符串
   * @return int-两者之间的编辑距离
   */

public static int getStrDistance(String a, String b) {

 if (a == null || b == null) return -1;
```

然后，初始化状态转移表。我用int型的二维数组来表示这个状态转移表，并对i为0且j大于0的元素，以及i大于0且j为0的元素，赋予相应的初始值。

```
  // 初始用于记录化状态转移的二维表
  int[][] d = new int[a.length() + 1][b.length() + 1];

  // 如果i为0，且j大于等于0，那么d[i, j]为j
  for (int j = 0; j <= b.length(); j++) {
   d[0][j] = j;
  }

  // 如果i大于等于0，且j为0，那么d[i, j]为i
  for (int i = 0; i <= a.length(); i++) {
   d[i][0] = i;
  }
```

我这里实现的时候，i和j都是从0开始，所以我计算的d[i+1, j+1]，而不是d[i, j]。而d[i+1, j+1] = min(d[i, j+1] + 1, d[i+1, j] + 1, d[i, j] + r(i, j)。

```
  // 实现状态转移方程
  // 请注意由于Java语言实现的关系，代码里的状态转移是从d[i, j]到d[i+1, j+1]，而不是从d[i-1, j-1]到d[i, j]。本质上是一样的。
  for (int i = 0; i < a.length(); i++) {
   for (int j = 0; j < b.length(); j++) {

    int r = 0;
    if (a.charAt(i) != b.charAt(j)) {
     r = 1;
    }

    int first_append = d[i][j + 1] + 1;
    int second_append = d[i + 1][j] + 1;
    int replace = d[i][j] + r;

    int min = Math.min(first_append, second_append);
    min = Math.min(min, replace);
    d[i + 1][j + 1] = min;

   }
  }

  return d[a.length()][b.length()];

 }

}
```

最后，我们用测试代码测试不同字符串之间的编辑距离。

```
public static void main(String[] args) {
  // TODO Auto-generated method stub
  System.out.println(Lesson10_1.getStrDistance("mouse", "mouuse"));

 }
```

从推导的表格和最终的代码可以看出，我们相互比较长度为m和n的两个字符串，一共需要求mxn个子问题，因此计算量是mxn这个数量级。和排列法的m^n相比，这已经降低太多太多了。

我们现在可以快速计算出编辑距离，所以就能使用这个距离作为衡量字符串之间相似度的一个标准，然后就可以进行查询推荐了。

到这里，使用动态规划来实现的编辑距离其实就讲完了。我把两个字符串比较的问题，分解成很多子串进行比较的子问题，然后使用状态转移方程来描述状态（也就是子问题）之间的关系，并根据问题的定义，保留最小的值作为当前的编辑距离，直到过程结束。

如果我们使用动态规划法来实现编辑距离的测算，那就能确保查询推荐的效率和效果。不过，基于编辑距离的算法也有局限性，它只适用于拉丁语系的相似度衡量，所以通常只用于英文或者拼音相关的查询。如果是在中文这种亚洲语系中，差一个汉字（或字符）语义就会差很远，所以并不适合使用基于编辑距离的算法。

实战演练：钱币组合的新问题

和排列组合等穷举的方法相比，动态规划法关注发现某种最优解。如果一个问题无需求出所有可能的解，而是要找到满足一定条件的最优解，那么你就可以思考一下，是否能使用动态规划来降低求解的工作量。

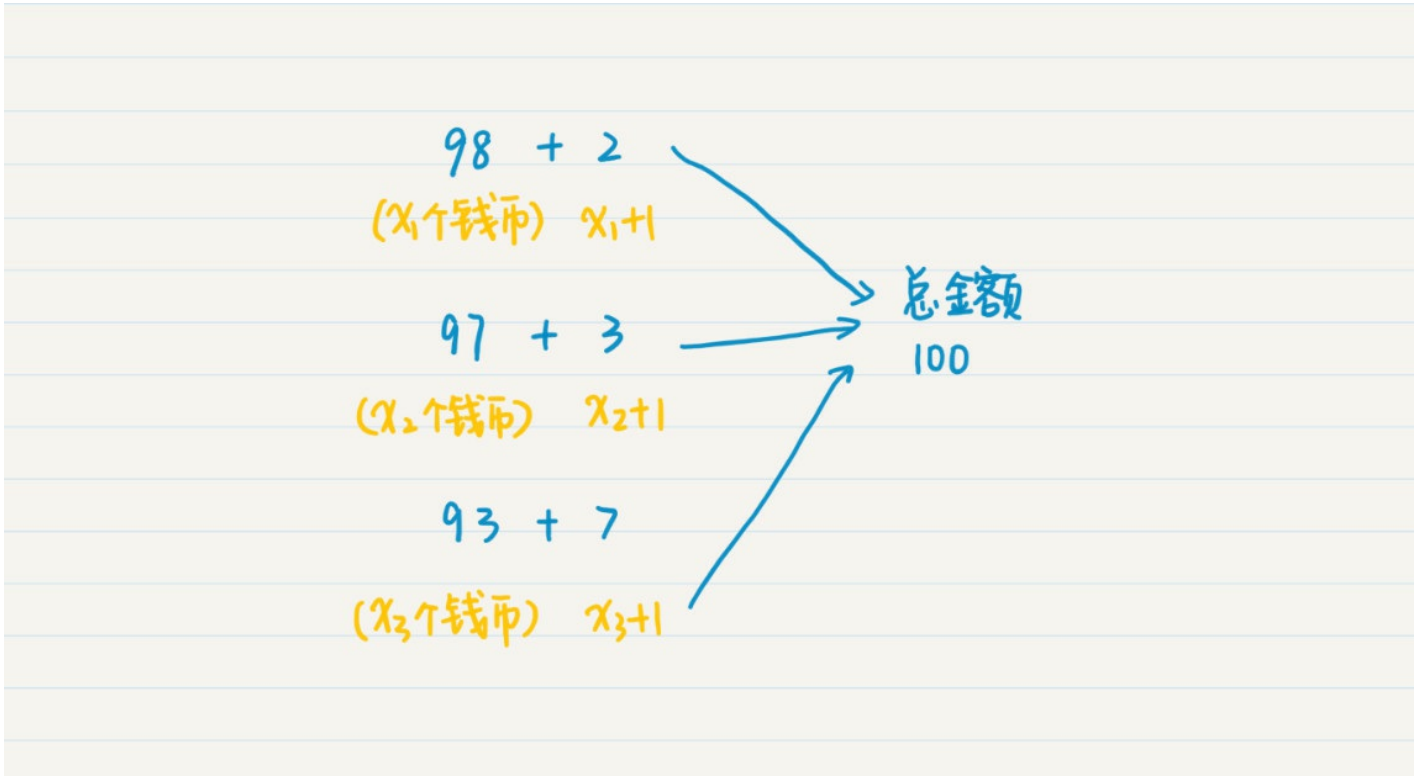
还记得之前我们提到的新版舍罕王奖赏的故事吗？国王需要支付一定数量的赏金，而宰相要列出所有可能的钱币组合，这使用了排列组合的思想。如果这个问题再变化为“给定总金额和可能的钱币面额，能否找出钱币数量最少的奖赏方式？”，那么我们是否就可以使用动态规划呢?

思路和之前是类似的。我们先把这个问题分解成很多更小金额的子问题，然后试图找出状态转移方程。如果增加一枚钱币c，那么当前钱币的总数量就是增加c之前的钱币总数再加上当前这枚。举个例子，假设这里我们有三种面额的钱币，2元、3元和7元。为了凑满100元的总金额，我们有三种选择。

第一种，总和98元的钱币，加上1枚2元的钱币。如果凑到98元的最少币数是x_{1}，那么增加一枚2元后就是(x_{1} + 1)枚。

第二种，总和97元的钱币，加上1枚3元的钱币。如果凑到97元的最少币数是x_{2}，那么增加一枚3元后就是(x_{2} + 1)枚。

第三种，总和93元的钱币，加上1枚7元的钱币。如果凑到93元的最少币数是x_{3}，那么增加一枚7元后就是(x_{3} + 1)枚。



比较一下以上三种情况的钱币总数，取最小的那个就是总额为100元时，最小的钱币数。换句话说，由于奖赏的总金额是固定的，所以最后选择的那枚钱币的面额，将决定到上一步为止的金额，同时也决定了上一步为止钱币的最少数量。根据这个，我们可以得出如下状态转移方程：

$$c[i] = \arg\min_{j=1}^{n} (c[i-value(j)]+1)$$

其中，c[i]表示总额为i的时候，所需要的最少钱币数，其中j=1,2,3,…,n，表示n种面额的钱币，value[j]表示第j种钱币的面额。c[i - values(j)]表示选择第j种钱币的时候，上一步为止最少的钱币数。需要注意的是，i - value(j)需要大于等于0，而且c[0] = 0。

我这里使用这个状态转移方程，做些推导，具体的数据你可以看下面这个表格。表格每一行表示奖赏的总额，前3列表示3种钱币的面额，最后一列记录最少的钱币数量。表中的“/”表示不可能，或者说无解。

总额\面额	2	3	7	最少钱币数c(x)
1	/	/	/	/
2	1	/	/	1
3	/	1	/	1
4	2	/	/	2
5	c(3) + 1 = 2	c(2) + 1 = 2	/	min(2, 2) = 2
6	c(4) + 1 = 3	c(3) + 1 = 2	/	min(3, 2) = 2
7	c(5) + 1 = 3	c(4) + 1 = 3	1	min(3, 3, 1) = 1
8	c(6) + 1 = 3	c(5) + 1 = 3	/	min(3, 3) = 3
9	c(7) + 1 = 2	c(6) + 1 = 3	c(2) + 1 = 2	min(2, 3, 3) = 2
10	c(8) + 1= 4	c(7) + 1 = 2	c(3) + 1 = 2	min(4, 2, 2) = 2
11	...	...	...	...

这张状态转移表同样可以帮助你来理解状态转移方程的正确性。一旦状态转移方程确定了，要编写代码来实现就不难了。

小结

通过这两节的内容，我讲述了动态规划主要的思想和应用。如果仅仅看这两个案例，也许你觉得动态规划不难理解。不过，在实际应用中，你可能会产生这些疑问：什么时候该用动态规划？这个问题可以用动态规划解决啊，为什么我没想到？我这里就讲一些我个人的经验。

首先，如果一个问题有很多种可能，看上去需要使用排列或组合的思想，但是最终求的只是某种最优解（例如最小值、最大值、最短子串、最长子串等等），那么你不妨试试是否可以使用动态规划。

其次，状态转移方程是个关键。你可以用状态转移表来帮助自己理解整个过程。如果能找到准确的转移方程，那么离最终的代码实现就不远了。当然，最好的方式，还是结合工作中的项目，不断地实践，尝试，然后总结。

今日学习笔记

第10节 动态规划（下）

1.状态转移方程是什么？

从上一个状态到下一个状态之间可能存在的一些变化，以及基于这些变化的最终决策结果。我们把这样的表达式称为状态转移方程。我上节最开始就说过，在所有动态规划的解法中，状态转移方程是关键，所以你一定要掌握它。

2. 基于编辑距离的算法有什么局限性？

基于编辑距离的算法也有局限性，它只适用拉丁语系的相似度衡量，所以通常只用于英文或者拼音相关的查询。如果在中文这种亚洲语系中，差一个汉字（或字符）语义就会差很远，所以并不适合使用基于编辑距离的算法。

黄申 · 程序员的数学基础课

思考题

对于总金额固定、找出最少钱币数的题目，用循环或者递归的方式该如何进行编码呢？

欢迎在留言区交作业，并写下你今天的学习笔记。你可以点击“请朋友读”，把今天的内容分享给你的好友，和他一起精进。

精选留言

云开

还是弄不明白编辑距离 为什么插入时是从空串开始 替换确并不计算从空串到有字符的过程

2019-01-31 19:19

作者回复

你可以参考那张状态转移表，看看是从哪一格到哪一格，字符串是如何变换的，相邻格子的变换就三种方式，插入、删除和替换。替换可以将字符串中的某个字符替换成另一个字符

2019-01-31 23:25

我心留

```
public class Lesson10_2 {
/**
* 动态规划求最小钱币数
* @param c 用一维数组记录每一步的总金额 * @param value 用一维数组记录三种面额的纸币
* @return
*/
public static int getMinMoney(int[] c, int[] value) {
int[] t = new int[3];
for (int i = 0; i < c.length; i++) {
for (int j = 0; j < value.length; j++) {
if (i - value[j] >= 0) {
t[j] = c[i - value[j]] + 1;
}
}
int min = Math.min(t[0], t[1]);
min = Math.min(min, t[2]);
c[i] = min;
}
return c[c.length - 1];
```

```
}
public static void main(String[] args) {
int[] c = new int[100];
int[] value = new int[] { 2, 3, 7 };
System.out.println(getMinMoney(c, value)+1);
}
}
```

老师看一下代码对吗，运行结果是15

2019-01-05 09:22

作者回复

代码的逻辑是对的

2019-01-14 01:51

lianlian

方法1，动态规划，最快。方法2递归有点慢，方法三递归，超级慢。在aim数值大于30的时候，三种写法，在我电脑速度快慢特别明显。用2元,3元,5元去找开100块，用递归方法，我的电脑要等到地老天荒O(∩_∩)O哈哈~

```
#include<iostream>
#include<vector>

using namespace std;

int dp_solve(int *a, int n, int aim){
vector<vector<int>> dp(n, vector<int>(aim+1, 0));

for(int j = 1; j <= aim; j++){
dp[0][j] = INT_MAX;
if(j >= a[0] && dp[0][j - a[0]] != INT_MAX)
dp[0][j] = dp[0][j - a[0]] + 1;
}

for(int i = 1; i < n; i++){
for(int j = 1; j <= aim; j++)
{
int tmp = INT_MAX;
if(j - a[i] >= 0 && dp[i][j - a[i]] != INT_MAX)
tmp = dp[i][j - a[i]] + 1;

dp[i][j] = min(dp[i-1][j], tmp);
}
}

return dp[n-1][aim] == INT_MAX ? -1 : dp[n-1][aim];
}

int min_res = INT_MAX;
void recur_solve(int *a, int n, int aim, int k){
if(aim == 0){
min_res = min(min_res, k);
return;
}
if(aim < 0)
return;
```

```
for(int i = 0; i < n; i++){
aim -= a[i];
k++;
recur_solve(a, n, aim, k);
aim += a[i];
k--;
}
}

int min_res2 = INT_MAX;
void recur_solve2(int *a, int n, int aim, vector<int> res){
if(aim == 0){
int size = res.size();
min_res2 = min(min_res2, size);
return;
}
if(aim < 0)
return;
for(int i = 0; i < n; i++){
res.push_back(a[i]);
recur_solve2(a, n, aim - a[i], res);
res.pop_back();
}
}

int main(){
int a[] = {2,3,7};
int sum = 25;
/***dp最快**/
cout << dp_solve(a, 3, sum) << endl;

/***这种递归有点久**/
recur_solve(a, 3, sum, 0);
cout << min_res << endl;

/**这个太久了**/
vector<int> result;
recur_solve2(a, 3, sum, result);
cout << min_res2 << endl;
return 0;
}
```

2019-01-04 14:40

作者回复

动手实验，比较不同的实现，

2019-01-14 01:44

冰木

老大，我可能没有得到要领，可以推到下，表格中，第一行，第二列吗?

2019-01-26 08:50

作者回复

是min(3, 1, 2)对吧，这个是mo和m的比较，3表示增加一个m再增加一个o，再删掉一个o，编辑距离是2+1=3。1表示两个字符串都是m，其中一个再增加一个o，编辑距离是1。2表示一个m增加o，一个从空集到m，编辑距离是2。你可以顺着第9讲最后

的表格来推导。

2019-01-27 11:45

mickey

```
package Part01;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class Lesson10_ex {
public static void main(String[] args) {
switchMoney(2, 3, 7, 100);
}

private static void switchMoney(int mz1, int mz2, int mz3, int total) {
List<Integer[]> list = new ArrayList<Integer[]>();
int s1 = total / mz1;
int s2 = total / mz2 + 1;
int s3 = total / mz3 + 1;
for (int i = 0; i <= s1; i++) {
for (int j = 0; j <= s2; j++) {
for (int k = 0; k <= s3; k++) {
if (mz1 * i + mz2 * j + mz3 * k == 100) {
list.add(new Integer[] { i, j, k });
}
}
}
}
Integer[] result = new Integer[3];
int min = total;
for (Integer[] integers : list) {
int sum = 0;
for (Integer num : integers) {
sum += num;
}
if (min > sum) {
min = sum;
result = integers;
}
}
System.out.println("最小数：" + min);
System.out.println(Arrays.toString(result));
}
}
```

2019-01-04 16:53

梅坊帝卿

按照面值排序优先取最大的方法 不一定能取到解 除非有万能的面额1 比如 2 5 7 总数15

2019-01-04 16:48

作者回复

是的 可能无解

2019-01-05 08:38

caohuan

本篇所得： 1.求解 最值可用动态规划 方法;
2.状态转移 可以把 大问题 分解为 小问题，再分解为 可以处理的问题，即 把 不可以处理的问题 分解为可以 处理的小问题（也为子问题）；
3.动态规划 适用于 下一个 状态与上一个状态有固定关系;
4.搜索引擎的 搜索词的查询推荐， 英文可用 编辑距离，中文 需要 转化 比 如转为英文 再使用 编辑距离;
5.从问题开始 ，初步分解 大问题为可解的子问题 为动态规划的方法，由问题 推到答案，也为反向思维法。

回答老师的问题：固定金额，找最小钱币数量，可用倒推法，总金额 减去 最大的 钱币数额 ，然后从钱币中寻找该数额，没有再把该数额逐渐减去 大的数额，一步步分解，可得 钱币的数量，该方法是 动态规划，但不能保证寻找的是最小的数量，局部最优 不一定全局最优，如果 需要寻找全部最优 需要运用 排列和组合。

2019-01-21 18:09

xiaobang

min的三个参数应该分别是插入删除替换，或者插入插入替换吧

2019-01-15 21:45

作者回复

是的

2019-01-15 23:46

Joe

1.C++实现，对总金额100的最小纸币是15.
2.用递归法总金额为30就要算很久。
3.另外的数学办法可以用总金额依次对最大金额纸币求余数，直到为0.商相加为答案。如：若 {1, 2, 3, 7}为纸币金额，对于100，所需最小纸币数：100/7=14余2; 2/2 = 1余0;则纸币数为14+1=15.

```
// 动态规划问题
#include <iostream>
#include <string>
#include <vector>
using namespace std;

class DynamicProgramming {
private:
vector<int> money = {1, 2, 3, 7}; // 纸币种类

public:
/**
* Description: 对于金额固定，找出最少钱币数及方式。
* prams: amountMoney- 输入总金额
* return: 所需最小纸币数
*/
int findFewerstMethod(int amountMoney) {
int c[amountMoney];
c[0] = 0;

int temp;
for (unsigned int i = 1; i < amountMoney; i++) {
// 用最大值初始化
int tempMin = amountMoney;
for (unsigned int j = 0; j < money.size(); j++) {
int diff = i - money[j];
if (0 <= diff) {
temp = c[diff] + 1;
```

```
} else {
// 此情况表示该纸币无效，选择最大值。
temp = amountMoney;
}
// 求出最小值
if (temp < tempMin) {
tempMin = temp;
}
}
c[i] = tempMin;
}

return c[amountMoney - 1];
}
};

// test
int main(void) {
DynamicProgramming test;
int res = test.findFewerstMethod(100);
cout << res << endl;
return 0;
}
```

2019-01-14 09:05

作者回复

答案正确

2019-01-16 01:41

清如許

https://github.com/somenzz/geekbang/blob/master/mathOfProgramer/chapter10_dynamic_programming.py
实现了循环和递归，循环的方式快，递归的方式特别慢。
个人感觉递归是从后往前推导的，每一步的结果不论是否最优都保存在堆栈中，都占用了内存空间，算法上已经不属于动态规划。

循环的方式不论 num 有多大，仅占用了7个变量的内存空间，每一步都保留上一步的最优解，因此效率较高，而且可以方便地打印出有最小数量的组合。

循环方式的代码的输出如下：

```
1 -> None
2 -> (1, [2])
3 -> (1, [3])
4 -> (2, [2, 2])
5 -> (2, [2, 3])
6 -> (2, [3, 3])
7 -> (1, [7])
8 -> (3, [2, 3, 3])
9 -> (2, [2, 7])
10 -> (2, [3, 7])
100 -> (15, [2, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7])
```

2019-01-07 12:32

作者回复

确实，动态规划使用循环更快

2019-01-14 01:56

菩提

思考题编码：

```
public static int least_count(int num) {
if (num < 0)
return -1;
int len = num;
if (num < 9)
len = 8;
int[] c = new int[len + 1];
c[0] = -1;
c[1] = -1;
c[2] = 1;
c[3] = 1;
c[4] = 2;
c[5] = 2;
c[6] = 2;
c[7] = 1;
c[8] = 3;

if (num < 9) {
return c[num];
}

for (int i = 9; i <= num; i++) {
int a = c[i - 2] + 1;
int b = c[i - 3] + 1;
int min = Math.min(a, b);
int m = c[i - 7] + 1;
min = Math.min(min, m);
c[i] = min;
}

return c[num];
}

public static void main(String[] args) {
System.out.println(least_count(100));
}
```

运行结果： 15

2019-01-06 20:59

作者回复

思路正确 ，编码可以稍微再通用点，用循环访问不同的币值

2019-01-14 01:53

lianlian

老师早上好(^_^) / ，在第一题求r的时候，条件判断语句应该是a.charAt(i+1) != b.charAt(j+1)吧 ？

2019-01-04 08:20

作者回复

应该是i和j没错，只是这里的i和j对应于d[i+1][j+1]。这里比较容易让人混淆，主要是因为d中的0下表表示的是空串，而不是字符串中的第一个字符。我之后将代码注释加一下

2019-01-05 09:10

qinggeouye

python 动态规划和递归两种解法

需要注意的是各种边界情况的处理，有解无解。总金额=50 的时候，递归速度慢了下来，但还是比较快的，总金额=100 运行到电脑发烫也没个结果...

```
import copy

def min_coins_recruit(target_money, coins):
"""
对于总金额固定，使用函数的递归(嵌套调用)，找出最少钱币数
:param target_money: 总金额大小
:param coins: 可选面额的钱币列表，如 [2, 3, 7]
:return:
"""
if target_money == 0:
return 0
elif target_money < min(coins):
return None # 不存在或无解
elif target_money in coins:
return 1

result = list()
for j in coins:
count = min_coins_recruit(target_money - j, coins)
if count is None:
continue
else:
result.append(count)
if len(result) > 0:
return min(result) + 1
else:
return None # 不存在或无解


def min_coins_dyna_pro(target_money, coins):
"""
对于总金额固定，使用动态规划算法，找出最少钱币数
:param target_money: 总金额大小
:param coins: 可选面额的钱币列表
:return:
"""
if min(coins) > target_money:
return None
# 每个额度的钱币数，初始化值为 None，假设不存在或无解
count = list([None]*(target_money+1))
# 一些确定额度的钱币数
count[0] = 0
```

```
for j in coins:
if j <= target_money:
count[j] = 1
# 动态规划求解钱币数
for i in range(min(coins) + 1, target_money+1):
tmp_count = list() # 临时保存钱币数
tmp_coins = [j for j in coins if j <= i] # 求解 count[i-j] 时，i-j 大于 0
for j in tmp_coins:
if count[i-j] is not None:
tmp_count.append(count[i-j])
if len(tmp_count) > 0:
count[i] = min(tmp_count) + 1

return count[target_money]


# 如何获取钱币组合?
if __name__ == "__main__":
print("动态规划，最少钱币数：", min_coins_dyna_pro(50, [2, 3, 7]))
print("递归方法，最少钱币数：", min_coins_recruit(50, [2, 3, 7]))
```

2019-02-19 22:24

作者回复

动态规划的优势体现出来了，实现了两种方式还进行了比较，赞

2019-02-20 01:49

qinggeouye

表格的行表示字符串 A，表格的列表示字符串 B

A = 空，B = 空，对应表格第 0 行第 0 列，编辑距离 d[0][0]=0

A = 空，B = m，对应表格第 0 行第 1 列，编辑距离 d[0][1]=1

A = 空，B = mo，对应表格第 0 行第 2 列，编辑距离 d[0][2]=2

以此类推....

第 1 行第 2 列 min(3, 1, 2) = 1 是怎么来的，是否可以这样理解：

第 1 行第 2 列 这个格子（对应 A=m ，B=mo），其相邻的格子分别为：

第 0 行第 2 列（A=空，B=mo），A 从空增加一个 m ，B 从空增加一个 m 再增加一个 o ，编辑距离为 3 ；（这里 B 从空开始增加，不太理解...）

第 1 行第 1 列（A=m，B=m），两个字符串都是 m ，其中 B 增加 o，编辑距离为 1 ；

第 0 行第 1 列（A=空，B=m），A 从空集增加一个 m ，B 从 m 增加一个 o ，编辑距离为 2 ；（而这里 B 又是从 m 开始增加...）

按这样理解，第 1 行第 1 列 min(2, 2, 0)=0 似乎又说不通，请老师点拨一下，谢谢 ～

2019-02-17 22:24

作者回复

你的这段理解

"第 0 行第 2 列（A=空，B=mo），A 从空增加一个 m ，B 从空增加一个 m 再增加一个 o ，编辑距离为 3 ；（这里 B 从空开始增加，不太理解...）

第 1 行第 1 列（A=m，B=m），两个字符串都是 m ，其中 B 增加 o，编辑距离为 1 ；

第 0 行第 1 列（A=空，B=m），A 从空集增加一个 m ，B 从 m 增加一个 o ，编辑距离为 2 ；（而这里 B 又是从 m 开始增加...）"

是正确的。至于你说的“按这样理解，第 1 行第 1 列 min(2, 2, 0)=0 似乎又说不通”具体是指什么？

2019-02-18 01:50

zzz

total = 100
c = [0] * (total + 1)
value = [2, 3, 7]
t = [0] * 3
t[0] = 9999
t[1] = 9999
t[2] = 9999

for i in range(1, total+1, 1):
log = '' # 用于记录该使用的上一级动态规则，最后输出
for j in range(0, len(value)):
if i - value[j] >= 0:
t[j] = c[i - value[j]] + 1
log = log + '第1个数字选' + str(value[j]) + '时，剩余c[' + str(i - value[j]) + ']+1=' + str(c[i - value[j]]) + '+1 ； '
c[i] = min(t[0], t[1], t[2])
print('c[', i, ']最少钱币数', c[i], ':', log)

输出：
...
...
c[97]最少钱币数 15 : 第1个数字选2时，剩余c[95]+1=15+1 ； 第1个数字选3时，剩余c[94]+1=14+1 ； 第1个数字选7时，剩余c[90]+1=14+1 ；
c[98]最少钱币数 14 : 第1个数字选2时，剩余c[96]+1=15+1 ； 第1个数字选3时，剩余c[95]+1=15+1 ； 第1个数字选7时，剩余c[91]+1=13+1 ；
c[99]最少钱币数 16 : 第1个数字选2时，剩余c[97]+1=15+1 ； 第1个数字选3时，剩余c[96]+1=15+1 ； 第1个数字选7时，剩余c[92]+1=15+1 ；
c[100]最少钱币数 15 : 第1个数字选2时，剩余c[98]+1=14+1 ； 第1个数字选3时，剩余c[97]+1=15+1 ； 第1个数字选7时，剩余c[93]+1=14+1 ；

2019-02-17 21:13

予悠悠

用python交作业
用递归来实现时，运行非常慢。用循环实现时，由于记录了每一步的计算结果，不需要重复计算，速度快很多。

递归：
import sys
def least_bills_recursion(total):
if total == 0:
return 0
if total < 0:
return sys.maxint
min_bills = min(1 + least_bills_recursion(total-2), 1 + least_bills_recursion(total - 3),
1 + least_bills_recursion(total-7))
return min_bills

循环：

```
def least_bills_iteration(total):
current = 0
dp = [0] * (total + 1)
dp[2] = 1
dp[3] = 1
dp[7] = 1
for i in xrange(3, total+1, 1):
if i >= 7:
dp[i] = min(dp[i-2], dp[i-3], dp[i-7]) + 1
elif i >= 3 and i < 7:
dp[i] = min(dp[i-2], dp[i-3]) + 1
else:
dp[i] = dp[i-2] + 1
return dp[total]
```

当总金额为100时，答案为15.

2019-01-22 00:00

作者回复

实现了两种方法，并进行了对比，赞

2019-01-22 09:57

pyhhou

这道题对于每种钱币的数量没有限制，应该就是背包问题里面的“完全背包”一类问题，代码如下，最后输出为15，望老师指点：

```
public int calculate(int[] coins, int totalValue) {
if (totalValue <= 0) {
return 0;
}

int[] dp = new int[totalValue + 1];

for (int i = 1; i <= totalValue; ++i) {
int min = Integer.MAX_VALUE;

for (int j = 0; j < coins.length; ++j) {
if ((coins[j] <= i) && (dp[i - coins[j]] != Integer.MAX_VALUE)) {
min = Math.min(min, dp[i - coins[j]] + 1);
}
}

dp[i] = min;
}

return dp[totalValue];
}
```

2019-01-16 09:14

作者回复

答案是正确的

2019-01-17 07:43

会飞的猪

python实现

```
import copy
list={0:0}
def getCoin(money):
list={0:[0,[]]}
coin=[2,3,7]
for i in range(0,money):
if(i in list.keys()):
for k in coin:
newMoney=i+k
if(newMoney<=money):
number=list[i][0]+1
newlist=copy.copy(list[i][1])
newlist.append(k)
if( not list.get(newMoney)):
list[newMoney]=[number,newlist]
elif(list[newMoney][0]>number):
list[newMoney] = [number,newlist]
# sorted(list.keys())
print(list)
getCoin(100)
```

2019-01-15 16:04

作者回复

看逻辑应该是对的，结果也可以贴一下

2019-01-16 01:53

会飞的猪

php实现代码

```
public function getMoney($money){
$coin=[2,3,7];
$list=[0];
for ($i=0;$i<=$money;$i++){
if(isset($list[$i])){
for($k=0;$k<count($coin);$k++){
$nowMoney=$i+$coin[$k];
if($nowMoney<=$money){
$number=$list[$i]+1;
if(!isset($list[$nowMoney])||$number<$list[$nowMoney]){
$list[$nowMoney]=$number;
}
}
}
}
}

ksort($list);
if(isset($list[$money])){
return $list[$money];
}else{
return '该金额无法通过硬币组合输出';
}
}
```

2019-01-15 15:25

Ricky

```
void getLeastReward(const int *amount, int n, int target, int &least,
vector<int> &result, vector<int> &tmp) {

if (target < 0) return;
if (target == 0) {
if (tmp.size() < least) {
least = tmp.size();
result = tmp;
}
}

for (int i = 0; i < n; ++i) {
if (target < amount[i]) continue;
int j = target / amount[i];
vector<int> rs = tmp;
while (j > 0) {
rs.push_back(amount[i]);
j--;
}
getLeastReward(amount, n, target % amount[i], least, result, rs);
}
}

void getLeastReward(const int *amount, int n, int target) {
if (target <= 0) return;
vector<int> rs, tmp;
int least = INT16_MAX;
getLeastReward(amount, n, target, least, rs, tmp);
cout << "\nThe least amount is " << least << endl;
cout << "Details are ";
for (int x: rs) {
cout << x << " ";
}
cout << endl;
}
```

主程序

```
int main() {
int amount[] = {2, 3, 7};
int target = 100;
getLeastReward(amount, 3, target);
return 0;
}
```

结果

```
The least amount is 15
Details are 7 7 7 7 7 7 7 7 7 7 7 7 7 2
```

2019-01-10 19:49

作者回复

结果正确

2019-01-14 02:59