**华南农业大学**

**《算法分析与设计》  
课程实验**

专业年级： 2014级信息与计算科学3班

学生学号： 201430120308

学生姓名： 何 唯

实验题目： 找零钱(动态规划法)

指导老师： 赵 峰

实验时间： 2017/03/14

1. **实验题目**

• [题目描述]

某国发行n种面额的钞票。现需要找零钱给顾客，设计一个算法，用最少的钞票张数找给顾客。（每种钞票都有足够的张数）

• [输入说明]

n值，钞票面额d[i]

• [输出说明]

对于1-30元的每种情况，给出最少钞票张数，以及每种面额钞票各几张

• [实验要求]

1.至少运行3组不同的输入数据；

2.用两种方法求解：动态规划、备忘录法.

1. **实验实现**

**• 1．**[编程语言]:实验用Java完成算法设计和程序设计并调试通过。

**• 2．**[解题思想、方法]

本实验采用两种动态规划算法来完成，首先是使用自底向上的动态规划(源码见附件源码1)，其次是使用备忘录法（源码见附件源码2）。

对于第一种方法，在兑换零钱的问题上，首先要找到其**最优子结构**，这也是动态规划算法的核心所在。

**2.1 找零钱问题的最优子结构性质**

**首先先进行符号的说明：**

* **n**为输入的零钱种数；
* **d[i]**为第i种零钱面额，；
* **C[j]**为当前需要找零的额度为j时，需要最少的零钱个数，即问题的最优解。
* **j**为当前需要找零的额度；
* **M**为所需要找零的额度最大值；
* **P[i][j]**为按照上述**C[j]**最优解的时候，各种零钱的个数，即问题的最优解对应的设计方案。(在备忘录法的时候，本人将**P[i][j]**写为了**P[j][i]**的形式，方便输出，本质也是一样的，对应**C[j]**这个最优解的时候，各个零钱的分配方案)

下面证明找零钱问题是有最优子结构的:

首先，在此将其最优子结构表述为，当利用**d[n]**中**n**个不同额度的零钱对当前需要找零的额度**j**进行找零的时候，最优分配方案为



此时对应最优解为**C[j]。**

**则分配方案肯定是子额度**



**的最优分配方案。**

**证明：**



假设不是最优设计方案，而P1[i][j]才是最优设计方案，其中P1[i][j]表示第i个零钱对应的找零个数。即为子额度的最优解。由于我们知道额度j对应最

优解为，即对于父问题额度j来说，

，

可以说明P[i][j]才是最优解。因此，最优子结构性质得证。

**2.2建立递归关系**

由于问题没有规定零钱的额度以及数量(即最低额度面额零钱可以不为1,零钱数目可以为1甚至0)，这里将分类讨论考虑所有情况。

当n=1时，即只能用一种钱币兑换零钱，钱币的面值为 d[1]，有





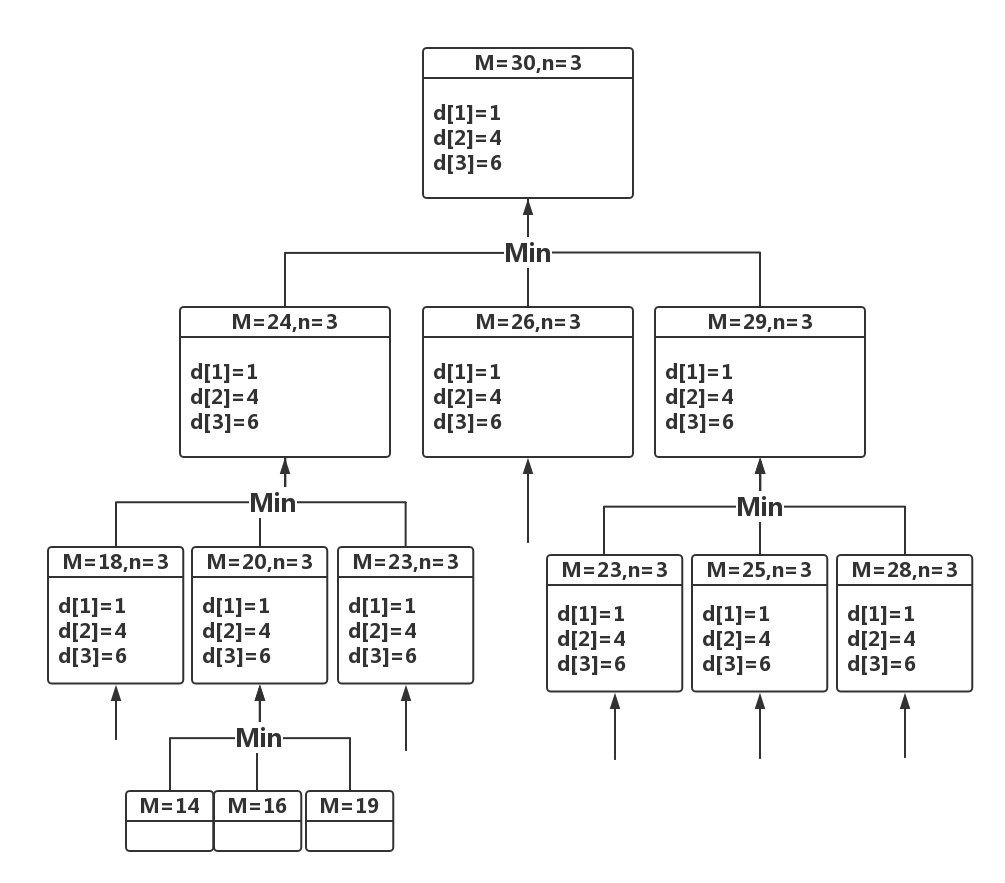
当n>1时，即零钱总数大于1的时候，

若，即对于第i种零钱额度比当前需要找零的额度大，因此可以采用递归关系。即可以找到,使得C[j]达到最优解，此时，并且最优解为。

若,即用第i种零钱找零j面额的时候，j恰好为第i种零钱的额度。此时用一张第i种零钱进行找零即可，直接可以获得最优解。

若时，即用第i种零钱找零j面额的时候，j大于第i种零钱的额度。此时只需要考虑前i-1中零钱就可以了，因为第i种零钱已经不能用于此次找零了。

下面可以画出递归树，以题目示例的输入输出(3 1 4 6)为例:



图一 递归树

由图一可见找零钱问题是具有最优子结构，并且计算最优值的时候可以取各子结构最优解中最小值+1构造出问题的最优解。比如在示例中，需要找零29块钱的时候，对应自顶向下找到29-1,29-4,29-6三个子问题中最优解的最小值即可，最后加上本身的一个面额为1或4或6零钱即为问题最优解。即，



构造的方案就是，



**2.3计算最优值**

事实上，不同子问题的个数最多只有n个。由此可见，在递归计算的时候，许多子问题被重复计算，比如图一递归树中，M=23在深度为2的时候被递归计算了两次。这也是动态规划求解问题的显著特征，子问题的最优解要使用的时候只需要递归调用即可，子问题被重复计算的次数越多，动态规划的优越性就越高。

要计算最优解以及对应分配方案，本实验采用数组来存储，首先是C[M]，代表的是需要找零M对应的最优解。P[i][j]表示为当需要找零的面额为j时，第i种零钱要多少个。在**方法一**中采用自底向上的方法，首先计算出各个子问题的解，再逐步构造父问题的解。

但是在**方法二**中，即采用备忘录法的时候，需要将方法一变形，编程自顶向下的递归方式，并且按照以下步骤进行设计：

1. 初始化时，最优解数组C[M]设置为特殊值0；最优分配方案数组P[][]同时也是初始化为0。
2. 从原问题出发，对每个待求的子问题，首先查看其对应记录项。本实验采用递归函数的调用方法min>change(C,P,d,M-d[i-z],n)，查看其记录是否最小。通过判断M-d[i-z]>0得知是否到达最底层子问题。
3. 在递归中计算子问题的最优解，最底层的子问题为当前需要找零面额等于当前零钱面额，此时只需要一张当前零钱即可

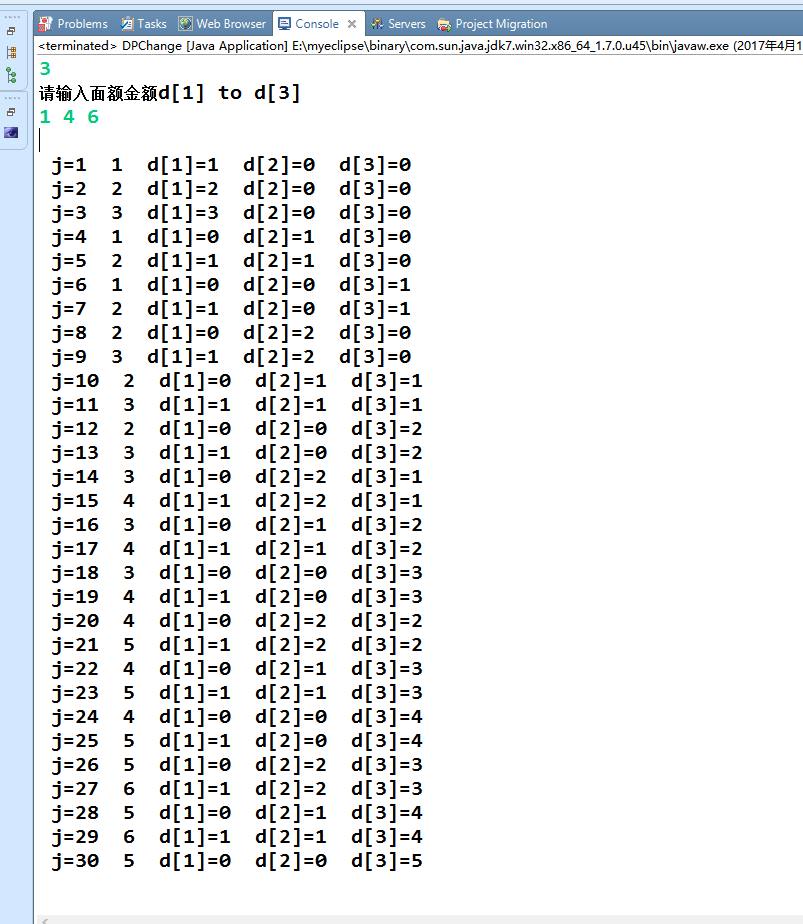
采用数组的方式保存已解决的子问题的答案，在下次需要解此子问题的时候，只要简单的查看该子问题的解答，而不必重新计算。

**• 3．**[实验说明，包括输入、输出、结果截图]

使用第一种方法的时候，考虑到了有可能最小的零钱额度不为1，此时将输出Java中最大整数Integer.*MAX\_VALUE，*表示当前问题无法求最优解，并且把最优设计方案对应各个零钱数目设置为0。如图三所示，当零钱最小额度为2的时候，对于1块钱是不可能找零的，于是最优解设置为最大整型数。

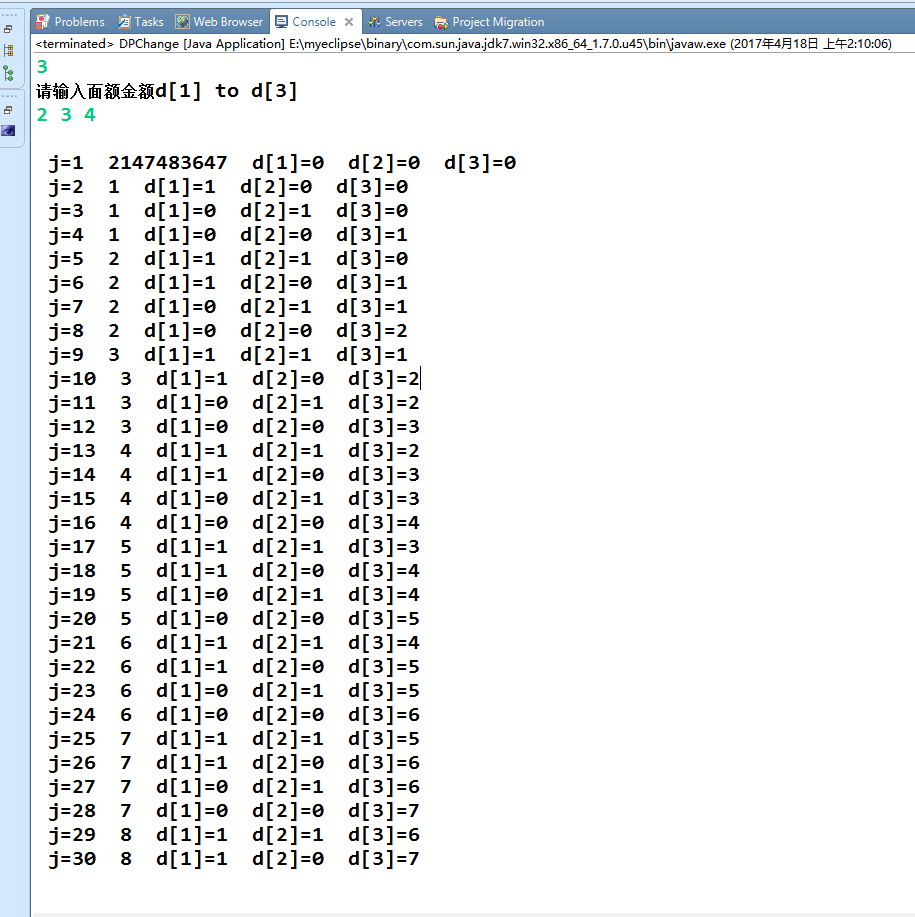
下面是方法一的输出：

输入**3 d[1] to d[3]为 2 3 4**的时候，结果如下图二



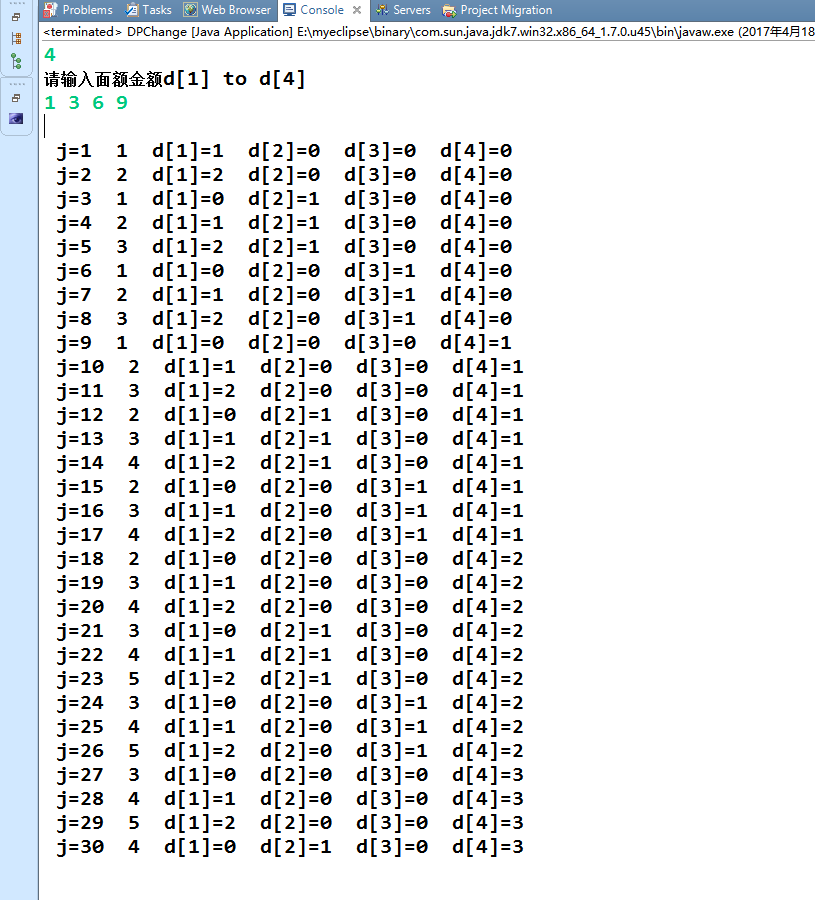
**图二** **d[1] to d[3]为 2 3 4**

输入**3 d[1] to d[3]为 2 3 4**的时候，结果如下图三

****

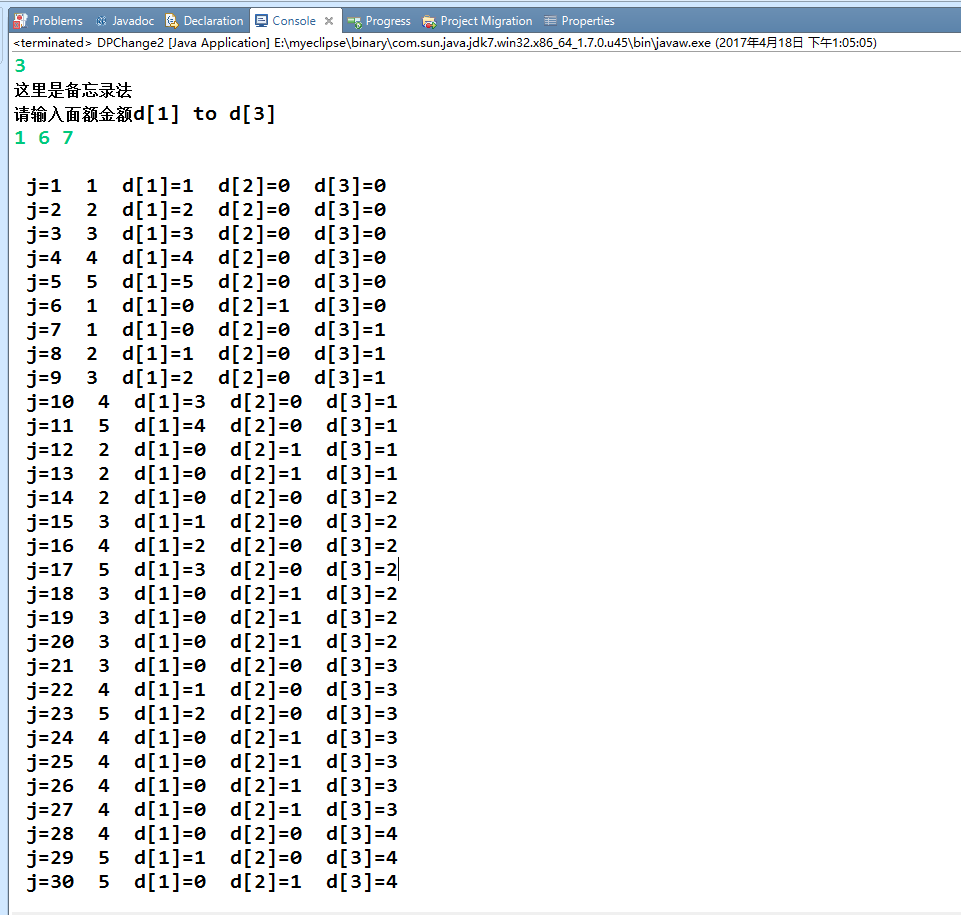
**图三** **d[1] to d[3]为 2 3 4**

输入**4 d[1] to d[4]为 1 3 6 9**的时候，结果如下图四



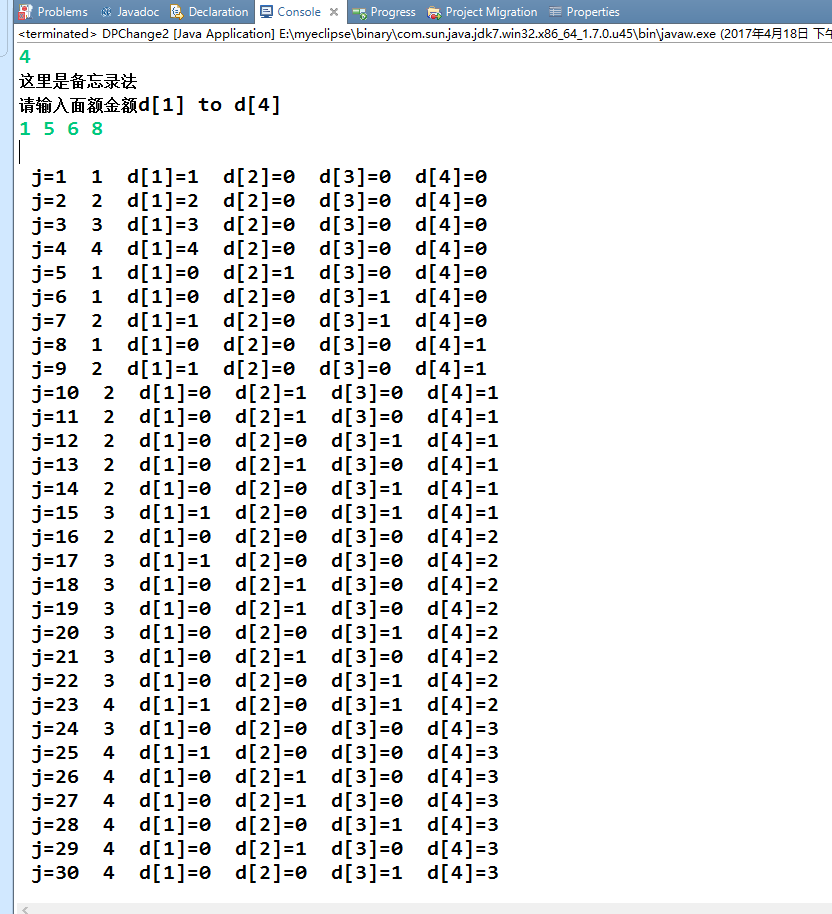
**图四** **d[1] to d[4]为 1 3 6 9**

下面是**方法二备忘录法**的输出，由于设置的输出跟方法一一致，这里再换三组输入进行测试：输入**3 d[1] to d[3]为 1 6 7**的时候，结果如下图五



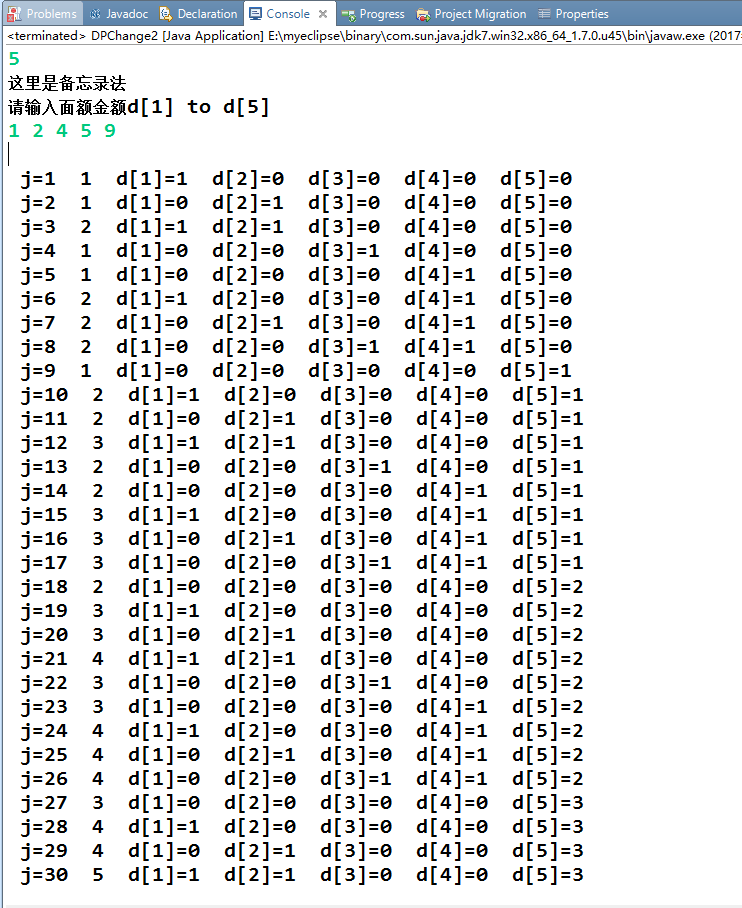
**图五** **d[1] to d[3]为 1 6 7**

输入**4 d[1] to d[4]为 1 5 6 8**的时候，结果如下图六



**图六** **d[1] to d[4]为 1 5 6 8**

输入5 **d[1] to d[5]为 1 2 4 5 9**的时候，结果如下图七

****

**图七 d[1] to d[5]为 1 2 4 5 9**

**• 4．**[算法复杂性计算过程]

**4.1 时间复杂性**

从上面算法可知,最优值C[j]的计算过程,最外层为循环嵌着while循环其中该参数k与n有关，而while循环中又嵌套着三个并列的for循环。因此本算法最坏情况下的复杂度是；最好的情况当然是里面for循环的条件不满足而不执行，此时的复杂度为 。其中：M表示需要兑换的零钱数，或者说最大需要兑换零钱额度(题目示例为30)，对于M来说，该值一般不是很大；n表示零钱的种类 ，n值一般也不会很大，如钱币总的有3种 (1元、4元、6元)。经过以上分析，如是最坏情况时的复杂度应为 ，则该值对于内存和运行速度较小的计算机，应用前景则不会很好。但本实验程序中的递归结构 在。可见对于需要找零的额度M，求 最优解时 ，并不需要对所有子问题情况都进行求最优解，而是只求一次即可，因此其效率大为上升。最坏的情况下需要执行 。这也是动态规划本身的子问题最优解重叠性的这个性质决定的，子问题重叠得越多，动态规划的优势也就越大。

对于方法二，备忘录法的话，时间复杂度也是容易得到的，同样是。这是因为程序中最多也是只有三层嵌套for循环，其次初始化二维数组最优设计方案数组耗费，初始化最优解数组耗费。填入备忘录的时候，耗费的时间复杂度也是和初始化的时候一致，综上所述，时间复杂度取决于程序哪一个三层嵌套for循环，即递归求解部分。

**4.2空间复杂性**

从上面算法方法一可知 ，用到了三个数组 ，分别为d[n]，C[M]，P[M][n]。空间复杂性主要由 P[M][n]决定，为 0(M×n)。因此，该算法的空间复杂性为0(M×n) ，可以应用到即使是小内存、低速度的计算机上。因此本算法有广阔的应用前景。

对于方法二备忘录法来说，空间复杂度同样是取决于最优分配方案数组P[M][n],易得空间复杂度也是同样地 0(M×n)。**小结和心得**

[例如：设计中遇到的问题，如何解决，实验心得，对算法改进的设想等]

讲到实验中遇到的问题，首先是把题目理解错误了，采用了贪心算法去计算，贪心策略为将需要找零的额度依次模最大面值的零钱，将其余数继续模下一个更小面值的零钱。这样子的话在本题示例(3 1 4 6)可能可以得到最优解，但是对于一些特殊情况就不可以了，比如说需要找零63元，零钱面值分别为1 21 50，贪心算法会得到的解为1张50,13张1块；而最优解应该是3张21块钱。这就是首先遇到的问题所在，下面附上当时写的代码。

**package** Algorism;

**import** java.util.Scanner;

/\*\*

\*

\* **@Project**：random

\* **@File**：DPChange2

\* **@Author**：何唯

\* **@Date**：2017年4月17日 上午08:48:23

\* **@Description**：

\*/

public class CoinChanger {

public int[] makeChange(int money,int[] coins){

int[] change=new int[coins.length];

int rest=money;

for(int i=coins.length-1 ; rest>0 && i>=0 ;i--){

if(coins[i] <= money){

change[i]=rest/coins[i];

  rest=rest%coins[i];             }

}

return change;

}

public void print(int[] coins,int[] methods){

System.out.print("需要");

int i=coins.length-1 ; rest>0 && i>=0 ;i--){if(methods[i]!=0){

System.out.print(methods[i]+"张"+coins[i]+"元.");

}}

System.out.println();

}

public static void main(String[] args){

CoinChanger changer=new CoinChanger();

**int** n,i=0;

Scanner scan =**new** Scanner(System.*in*);

n=scan.nextInt();

**int**[] d=**new** **int**[100];

System.*out*.println("请输入面额金额d[1] to d["+n+"]");

**for** (i=1;i<=n;i++)

d[i]=scan.nextInt();

**int** [] C=**new** **int** [100];

**int** [][] P=**new** **int** [100][100];

int[] methods=changer.makeChange(money, coins);

changer.print(coins,methods);

}

}

其次，遇到的困难就是，网上大多数博客的动态规划方法都是错误的，特别的备忘录法，一开始被误导了，浪费了许多时间，后面自己熬了一夜才把备忘录法写了出来。最困难的地方就要数，构造这个问题的最优子结构，以及递归求解的代码。抽象的去想的话是不可行的，最好还是用纸笔画出几个简单的解，再通过这几个解去寻找递归规律。切忌大脑空想，动态规划是不可能斋想可以得到规律的，需要不断尝试。

最后一个就是，不清楚自己得到的递归结构是否正确，要通过不断的程序输出调试才能得到最后的正确答案。本次实验由于自己时间拖得比较后才去完成，下次应该提前完成实验才行。

对于贪心算法以及动态规划的有了自己的理解，有时候虽然说贪心算法比较好想出来，贪心策略比较多可以选择，但是正是这样迷惑了不少人，这种贪心算法有时候是不能得到最优解的。即使程序比较好写，我们还是要吧问题考虑周全，看看动态规划是否能解决此问题。除了本实验问题之外，还有许多拓展的问题不能用贪心算法解决的，比如说0-1背包问题以及背包问题的讨论等等。

算法设计对于本人来说，一开始比较找不准规律，现在完成了两个实验之后，对算法设计有了自己的一点小心得，希望后面的学习能继续进步，按时完成实验内容。

**源程序清单**

动态规划源文件1：DPChange.java

**package** Algorism;

**import** java.util.Arrays;

**import** java.util.Scanner;

/\*\*

\*

\* **@Project**：random

\* **@File**：DPChange

\* **@Author**：何唯

\* **@Date**：2017年4月17日 上午10:49:07

\* **@Description**：

\*/

**public** **class** DPChange {

**public** **static** **void** charge(**int** d[], **int** M,**int** n){

**int** i,k,a,b,min,flag;

**int**[] C=**new** **int** [100];

**int**[][] P=**new** **int** [100][100];

**for**(**int** j=1;j<=M;j++)//遍历子问题

{

**if**(n>1)//假如大于一种货币

{

k=n;//货币种数给k

flag=1;//未解决

**while**(k>1&&flag==1)

{

flag=1;

**if**(j==d[k])//当前需要找零面额等于当前零钱面额

{

C[j]=1;//一张当前零钱就可以了

//j是最大需要找零面额，输出解决信息

**if**(j==M)

System.*out*.print("\n j="+j+" "+C[j]);

flag=0;

**for**(i=1;i<=n;i++)

{

P[i][j]=0;

P[k][j]=1;

**if**(j==M)

System.*out*.print(" d["+i+"]="+P[i][j]);

}

}**else** **if**(j<d[k])//当前需要找零面额小于当前零钱面额

k--;//下一个零钱

**else**{

//当前需要找零面额大于当前零钱面额

min=C[j-d[k]];//初始化min为j-当前零钱面额

a=j-d[k];//子问题要解决的零钱面额

b=k;//当前零钱种类

**for**(i=1;i<k;i++)

**if**(min>C[j-d[k-i]])//找到子问题最优解

{

min=C[j-d[k-i]];

a=j-d[k-i];

b=k-i;

}

C[j]=min+1;//最优解为子问题最优解加上1枚硬币

//j是最大需要找零面额，输出解决信息

**if**(j==M)

System.*out*.print("\n j="+j+" "+C[j]);

flag=0;

**for**(i=1;i<=n;i++)//存储对于j的最佳方案

{

**if**(C[j]>=1000)P[i][j]=0;

**else**{

P[i][j]=P[i][a];//子问题a的对应方案

P[b][j]=P[b][a]+1;//子问题a的对应方案+1

}

**if**(j==M)

System.*out*.print(" d["+i+"]="+P[i][j]);

}

}

}**if**(j%d[1]!=0&&flag==1)//k==1

{

C[j]=Integer.*MAX\_VALUE*;

**if**(j==M)System.*out*.print("\n j="+j+" "+C[j]);

**for**(i=1;i<=n;i++)

{

P[i][j]=0;

**if**(j==M)System.*out*.print(" d["+i+"]="+P[i][j]);

}

}**else** **if**(flag==1){

C[j]=j/d[1];

**if**(j==M)System.*out*.print("\n j="+j+" "+C[j]);

P[1][j]=j/d[1];

**if**(j==M)System.*out*.print("\t d[1]="+P[1][j]);

**for**(i=2;i<=n;i++)

{

P[i][j]=0;

**if**(j==M)System.*out*.print("\t d["+i+"]="+P[i][j]);

}

}

}

**else**//一种货币

{

**if**(j%d[1]!=0)C[j]=Integer.*MAX\_VALUE*;

**else** C[j]=j/d[1];

**if**(j==M)System.*out*.print("\n j="+j+" "+C[j]);

P[1][j]=j/d[1];

**if**(j==M)System.*out*.print("\t d[1]="+P[1][j]);

}

}

}

**public** **static** **void** main(String[] args) {

**int** n,i=0;

Scanner scan =**new** Scanner(System.*in*);

n=scan.nextInt();

**int**[] d=**new** **int**[100];

System.*out*.println("请输入面额金额d[1] to d["+n+"]");

**for** (i=1;i<=n;i++)

d[i]=scan.nextInt();

**for**(i=1;i<=30;i++)

*charge*(d,i,n);

}

}

动态规划源文件2(备忘录法)：DPChange2.java

**package** Algorism;

**package** Algorism;

**import** java.util.Scanner;

/\*\*

\*

\* **@Project**：random

\* **@File**：DPChange2

\* **@Author**：何唯

\* **@Date**：2017年4月17日 上午10:48:23

\* **@Description**：

\*/

**public** **class** DPChange2 {

**public** **static** **int** change(**int** C[],**int** [][]P,**int** d[], **int** M,**int** n){

**int** i,j,a,b,min;

**int** k=n,t;

**for**(i=n;i>=1;i--)

{

**if**(M>d[i])

{

min=*change*(C,P,d,M-d[i],n);

a=M-d[i];//子问题要解决的零钱面额

b=i;//当前零钱种类

**for**(k=1;k<=n;k++)

{

P[M][k]=P[M-d[i]][k];

P[M][i]=P[M-d[i]][i]+1;

}

**for**(**int** z=1;z<i;z++)

{

**if**(min>*change*(C,P,d,M-d[i-z],n)&&M-d[i-z]>0){

min=*change*(C,P,d,M-d[i-z],n);

**for**(k=1;k<=n;k++)

{

P[M][k]=P[M-d[i-z]][k];

P[M][i]=P[M-d[i-z]][i]+1;

}

}

}

C[M]=min+1;

**return** C[M];

}**else** **if**(M==d[i])//当前需要找零面额等于当前零钱面额

{

**for**(k=1;k<=n;k++)

{

P[M][k]=0;

P[M][i]=1;

}

C[M]=1;

**return** C[M];

}**else**//当前需要找零面额小于当前零钱面额

{**continue**;}

}

**return** C[M];

}

**public** **static** **void** main(String[] args) {

// **TODO** Auto-generated method stub

**int** n,i=0;

Scanner scan =**new** Scanner(System.*in*);

n=scan.nextInt();

**int**[] d=**new** **int**[100];

System.*out*.println("请输入面额金额d[1] to d["+n+"]");

**for** (i=1;i<=n;i++)

d[i]=scan.nextInt();

**int** [] C=**new** **int** [100];

**int** [][] P=**new** **int** [100][100];

*change*(C,P,d,30,n);

**for**(**int** M=1;M<=30;M++)

{

System.*out*.print("\n j="+M+" "+C[M]);

**for**(i=1;i<=n;i++)

{

System.*out*.print(" d["+i+"]="+P[M][i]);

}

}

}

}

**五、评语及评分**

|  |
| --- |
| **评语及评分**  评阅人签名： |