第4章 递归和动态规划

斐波那契系列问题的递归和动态规划

【题目】

给定整数N，返回斐波那契数列的第N项。

【补充题目1】

给定整数N，代表台阶数，一次可以跨2个或者1个台阶，返回有多少种走法。

【举例】

N=3。可以三次都跨1个台阶；也可以先跨2个台阶，再跨1个台阶；还可以先跨1个台阶，再跨2个台阶。所以有三种走法，返回3。

【补充题目2】

假设农场中成熟的母牛每年只会生1头小母牛，并且永远不会死。第一年农场有1只成熟的母牛，从第二年开始，母牛将开始生小母牛。每只小母牛3年之后成熟又可以开始生小母牛。给定整数N，求出N年后牛的数量。

【举例】

N=6。第1年1头成熟母牛记为a。第2年a生了新的小母牛记为b，总牛数为2。第3年a生了新的小母牛记为c，总牛数为3。第4年a生了新的小母牛记为d，总牛数为4。第5年b成熟了，a和b分别生了新的小母牛，总牛数为6。第6年c也成熟了，a、b和c分别生了新的小母牛，总牛数为9。返回9。

【要求】

对以上所有问题，请实现时间复杂度O(logN)的解法。

【难度】

将 ★★★★

【解答】

原问题。O(2^N)的方法。斐波那契数列为1，1，2，3，5，8...，也就是除了第1项和第2项为1以外，对于第N项，有F(N)=F(N-1)+F(N-2)，于是很轻松的写出暴力递归的代码。请参看如下代码中的f1方法。

```java
public int f1(int n) {
	if (n < 1) {
		return 0;
	}
	if (n == 1 || n == 2) {
		return 1;
	}
	return f1(n - 1) + f1(n - 2);
}
```

O(N)的方法。斐波那契数列可以从左到右依次求出每一项的值，那么通过顺序计算求到第N项即可。请参看如下代码中的f2方法。

```java
public int f2(int n) {
```

```
		if (n < 1) {
			return 0;
		}
		if (n == 1 || n == 2) {
			return 1;
		}
		int res = 1;
		int pre = 1;
		int tmp = 0;
		for (int i = 3; i <= n; i++) {
			tmp = res;
			res = res + pre;
			pre = tmp;
		}
		return res;
	}
```

O(logN)的方法。如果递归式严格遵循F(N)=F(N-1)+F(N-2)，对于求第N项的值，有矩阵乘法的方式可以将时间复杂度降至O(logN)。F(n)=F(n-1)+F(n-2)，是一个二阶递推数列，一定可以用矩阵乘法的形式表示，且状态矩阵为2*2的矩阵：

$$(F(n), F(n-1)) = (F(n-1), F(n-2)) \times \begin{vmatrix} a & b \\ c & d \end{vmatrix}$$

把斐波那契数列的前4项F(1)==1，F(2)==1，F(3)==2，F(4)==3，代入可以求出状态矩阵：

$$\begin{vmatrix} a & b \\ c & d \end{vmatrix} = \begin{vmatrix} 1 & 1 \\ 1 & 0 \end{vmatrix}$$

求矩阵之后，当n>2时，原来的公式可化简为：

$$(F(3), F(2)) = (F(2), F(1)) \times \begin{vmatrix} 1 & 1 \\ 1 & 0 \end{vmatrix} = (1, 1) \times \begin{vmatrix} 1 & 1 \\ 1 & 0 \end{vmatrix}$$

$$(F(4), F(3)) = (F(3), F(2)) \times \begin{vmatrix} 1 & 1 \\ 1 & 0 \end{vmatrix} = (1, 1) \times \begin{vmatrix} 1 & 1 \\ 1 & 0 \end{vmatrix}^2$$

$$\vdots$$

$$(F(n), F(n-1)) = (F(n-1), F(n-2)) \times \begin{vmatrix} 1 & 1 \\ 1 & 0 \end{vmatrix} = (1, 1) \times \begin{vmatrix} 1 & 1 \\ 1 & 0 \end{vmatrix}^{n-2}$$

所以求斐波那契数列第N项的问题就变成了如何用最快的方法求一个矩阵的N次方的问题，而求矩阵N次方的问题，明显是一个更够在O(logN)时间内解决的问题。为了表述方便，我们现在用求一个整数N次方的例子来说明，因为只要理解了如何在O(logN)的时间复杂度内求整数N次方的问题，对于求矩阵N次方的问题是同理的，区别是矩阵乘法和整数乘法在细节上有些不一样，但是对于怎么乘更快，两者的道理相同。
假设一个整数是10，如何最快的求解10的75次方。
1，75的二进制形式为1001011。
2，10的75次方=(10^64) * (10^8) * (10^2) * (10^1)。在这个过程中，我们先求出10^1，然后根据10^1求出10^2，再根据10^2求出10^4，...，最后根据10^32求出10^64次方，即75的二进制形式总共有多少位，我们就使用了几次乘法。
3，在步骤2进行的过程中，把应该累乘的值乘起来即可。10^64、10^8、10^2、10^1应该累乘起来，因为64、8、2、1对应到75的二进制中，相应的位上是1。而10^32、10^16、10^4不应该累乘，因为32、16、4对应到75的二进制中，相应的位上是0。
对于矩阵来说同理，求矩阵m的p次方请参看如下代码中的matrixPower方法。其中muliMatrix方法是两个矩阵相乘的具体实现。

```java
public int[][] matrixPower(int[][] m, int p) {
	int[][] res = new int[m.length][m[0].length];
	// 先把res设为单位矩阵，相等于整数中的1。
	for (int i = 0; i < res.length; i++) {
		res[i][i] = 1;
	}
	int[][] tmp = m;
	for (; p != 0; p >>= 1) {
		if ((p & 1) != 0) {
			res = muliMatrix(res, tmp);
		}
		tmp = muliMatrix(tmp, tmp);
	}
	return res;
}

public int[][] muliMatrix(int[][] m1, int[][] m2) {
	int[][] res = new int[m1.length][m2[0].length];
	for (int i = 0; i < m2[0].length; i++) {
		for (int j = 0; j < m1.length; j++) {
			for (int k = 0; k < m2.length; k++) {
				res[i][j] += m1[i][k] * m2[k][j];
			}
		}
	}
	return res;
}
```

用矩阵乘法求解斐波那契数列第N项的全部过程请参看如下代码中的f3方法。

```java
public int f3(int n) {
	if (n < 1) {
		return 0;
	}
	if (n == 1 || n == 2) {
		return 1;
	}
	int[][] base = { { 1, 1 }, { 1, 0 } };
	int[][] res = matrixPower(base, n - 2);
	return res[0][0] + res[1][0];
}
```

补充问题1。如果台阶只有1节，方法只有1种。如果台阶有2节，方法有2种。如果台阶有N

节，最后跳上第N节的情况，要么是从N-2级台阶直接跨2个台阶上来，要么是从N-1级台阶跨1个台阶上来，所以台阶有N节的方法数为跨到N-2台阶的方法数加上跨到N-1台阶的方法数，即S(N)=S(N-1)+S(N-2)，初始项S(1)==1，S(2)==2。所以类似斐波那契数列，唯一的不同就是初始项不同。可以很轻易的写出O(2^N)与O(N)的方法，请参看如下代码中的s1和s2方法。

```
	public int s1(int n) {
		if (n < 1) {
			return 0;
		}
		if (n == 1 || n == 2) {
			return n;
		}
		return s1(n - 1) + s1(n - 2);
	}

	public int s2(int n) {
		if (n < 1) {
			return 0;
		}
		if (n == 1 || n == 2) {
			return n;
		}
		int res = 2;
		int pre = 1;
		int tmp = 0;
		for (int i = 3; i <= n; i++) {
			tmp = res;
			res = res + pre;
			pre = tmp;
		}
		return res;
	}
```

O(logN)的方法。表达式S(n)=S(n-1)+S(n-2)，是一个二阶递推数列，同样用上文矩阵乘法的方法，根据前4项S(1)==1，S(2)==2，S(3)==3，S(4)==5求出状态矩阵：

$$\begin{vmatrix} a & b \\ c & d \end{vmatrix} = \begin{vmatrix} 1 & 1 \\ 1 & 0 \end{vmatrix}$$

同样根据上文的过程得到：

$$(S(n), S(n-1)) = (S(2), S(1)) \times \begin{vmatrix} 1 & 1 \\ 1 & 0 \end{vmatrix}^{n-2} = (2, 1) \times \begin{vmatrix} 1 & 1 \\ 1 & 0 \end{vmatrix}^{n-2}$$

全部的实现请参看如下代码中的s3方法。

```
	public int s3(int n) {
		if (n < 1) {
			return 0;
```

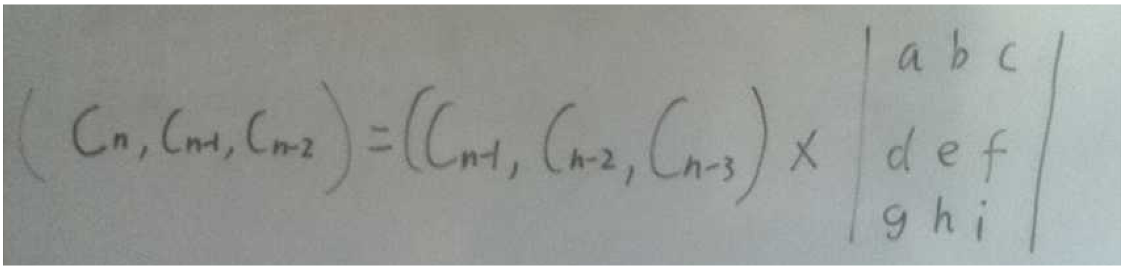
```java
		}
		if (n == 1 || n == 2) {
			return n;
		}
		int[][] base = { { 1, 1 }, { 1, 0 } };
		int[][] res = matrixPower(base, n - 2);
		return 2 * res[0][0] + res[1][0];
	}
```

补充问题2。所有牛都不会死，所以第N-1年的牛会毫无损失的活到第N年。同时所有成熟的牛都会生1头新的牛，那成熟牛的数量如何估计？就是第N-3年的所有牛，到第N年肯定都是成熟的牛，其间出生的牛肯定都没成熟。所以C(n)=C(n-1)+C(n-3)，初始项为C(1)==1，C(2)==2，C(3)==3。这个和斐波那契数列又是十分类似，只不过C(n)依赖C(n-1)和C(n-3)的值，而斐波那契数列F(n)依赖F(n-1)和F(n-2)的值。同样可以很轻易的写出O(2^N)与O(N)的方法，请参看如下代码中的c1和c2方法。

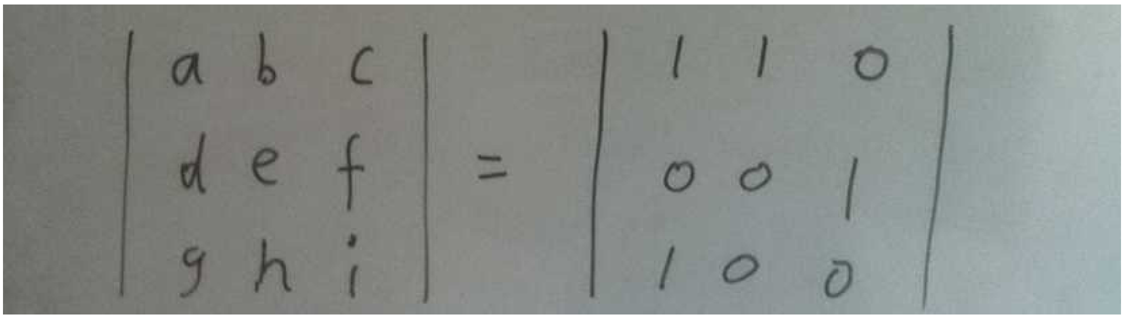
```java
	public int c1(int n) {
		if (n < 1) {
			return 0;
		}
		if (n == 1 || n == 2 || n == 3) {
			return n;
		}
		return c1(n - 1) + c1(n - 3);
	}

	public int c2(int n) {
		if (n < 1) {
			return 0;
		}
		if (n == 1 || n == 2 || n == 3) {
			return n;
		}
		int res = 3;
		int pre = 2;
		int prepre = 1;
		int tmp1 = 0;
		int tmp2 = 0;
		for (int i = 4; i <= n; i++) {
			tmp1 = res;
			tmp2 = pre;
			res = res + prepre;
			pre = tmp1;
			prepre = tmp2;
		}
		return res;
	}
```

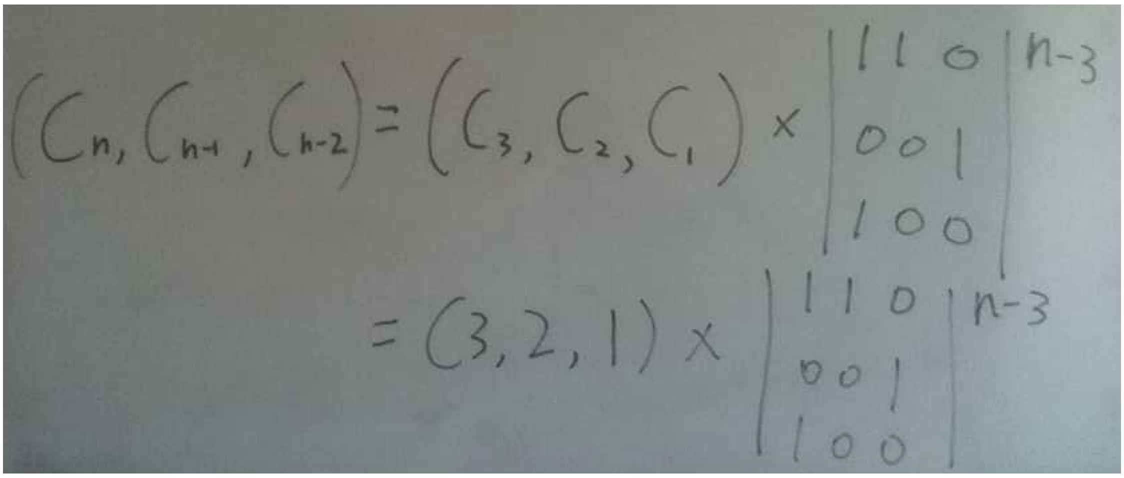
O(logN)的方法。C(n)=C(n-1)+C(n-3)是一个三阶递推数列，一定可以用矩阵乘法的形式表示，且状态矩阵为3*3的矩阵。



把前5项C(1)==1，C(2)==2，C(3)==3，C(4)==4，C(5)==6代入，求出状态矩阵：



求矩阵之后，当n>3时，原来的公式可化简为：



接下来的过程又是利用加速矩阵乘法的方式进行实现，具体请参看如下代码中的c3方法。

```
public int c3(int n) {
        if (n < 1) {
                return 0;
        }
        if (n == 1 || n == 2 || n == 3) {
                return n;
        }
        int[][] base = { { 1, 1, 0 }, { 0, 0, 1 }, { 1, 0, 0 } };
        int[][] res = matrixPower(base, n - 3);
        return 3 * res[0][0] + 2 * res[1][0] + res[2][0];
}
```

如果递归式严格符合F(n)=a*F(n-1)+b*F(n-2)+...+k*F(n-i)，那么它就是一个i阶的递推式，必然有与i*i的状态矩阵有关的矩阵乘法的表达。一律可以用加速矩阵乘法的动态规划将时间复杂度降为O(logN)。

换钱的方法数

【题目】

给定数组arr，arr中所有的值都为正数且不重复。每个值代表一种面值的货币，每种面值的货币可以使用任意张，再给定一个整数aim代表要找的钱数，求换钱有多少种方法。

【举例】

arr=[5,10,25,1]，aim=0。

组成0元的方法有1种，就是所有面值的货币都不用。所以返回1。

arr=[5,10,25,1]，aim=15。

组成15元的方法有6种，分别为3张5元，1张10元+1张5元，1张10元+5张1元，10张1元+1张5元，2张5元+5张1元，15张1元。所以返回6。

arr=[3,5]，aim=2。

任何方法都无法组成2元。所以返回0。

【难度】

尉 ★★☆☆

【解答】

本书将由浅入深的给出所有解法，最后再解释最优解。这道题目的经典之处在于它可以体现暴力递归、记忆搜索和动态规划之间的关系，并可以在动态规划的基础上进行再一次的优化。在面试中出现的大量暴力递归的题目都有相似的优化轨迹，希望引起读者重视。

首先介绍暴力递归的方法。如果arr=[5,10,25,1]，aim=1000，分析过程如下：

1，用0张5元的货币，让[10,25,1]去组成剩下的1000，最终方法数记为res1。
2，用1张5元的货币，让[10,25,1]去组成剩下的995，最终方法数记为res2。
3，用2张5元的货币，让[10,25,1]去组成剩下的990，最终方法数记为res3。
...
201，用200张5元的货币，让[10,25,1]去组成剩下的0，最终方法数记为res201。

那么res1+res2+...+res201的值就是总共的方法数。根据如上的分析过程定义递归函数process1(arr,index,aim)，它的含义是如果用arr[index..N-1]这些面值的钱去组成aim的话，返回总共的方法数。具体实现参见如下代码中的coins1方法。

```java
	public int coins1(int[] arr, int aim) {
		if (arr == null || arr.length == 0 || aim < 0) {
			return 0;
		}
		return process1(arr, 0, aim);
	}

	public int process1(int[] arr, int index, int aim) {
		int res = 0;
		if (index == arr.length) {
			res = aim == 0 ? 1 : 0;
		} else {
			for (int i = 0; arr[index] * i <= aim; i++) {
				res += process1(arr, index + 1, aim - arr[index] * i);
			}
		}
		return res;
	}
```

接下来介绍基于暴力递归的初步优化的方法，也就是记忆搜索的方法。暴力递归之所以暴

力是因为存在大量的重复计算。比如上面的例子，当已经使用0张5元+1张10元的情况下，后续应该求[25,1]组成剩下的990的方法总数。当已经使用2张5元+0张10元的情况下，后续还是求[25,1]组成剩下的990的方法总数。两个情况下都需要求process1(arr,2,990)。类似这样的重复计算在暴力递归的过程中大量发生，所以暴力递归方法的时间复杂度非常高并且和arr中钱的面值有关，最差情况下为O(aim^N)。
记忆化搜索的优化方式。process1(arr,index,aim)中arr是始终不变的，变化的只有index和aim，所以可以用p(index,aim)表示一个递归过程。重复计算之所以大量发生，是因为每一个递归过程的结果都没记下来，所以下次还要重复的去求。所以可以事先准备好一个map，每计算完一个递归过程，都将结果记录到map中。当下次进行同样的递归过程之前，先在map中查询是否这个递归过程已经计算过，如果已经计算过把值拿出来直接用就可以了，如果没计算过再进入递归过程。具体请参看如下代码中的coins2方法，它和coins1方法的区别就是准备好全局变量map，记录已经计算过的递归过程的结果，防止下次重复计算。因为本题的递归过程可由两个变量表示，所以map是一张二维表。map[i][j]表示递归过程p(i,j)的返回值。另外有一些特别值，map[i][j]==0表示递归过程p(i,j)从来没有计算过。map[i][j]==-1表示递归过程p(i,j)计算过但返回值是0。如果map[i][j]的值既不等于0也不等于-1，记为a，则表示递归过程p(i,j)的返回值为a。

```java
public int coins2(int[] arr, int aim) {
        if (arr == null || arr.length == 0 || aim < 0) {
                return 0;
        }
        int[][] map = new int[arr.length + 1][aim + 1];
        return process2(arr, 0, aim, map);
}

public int process2(int[] arr, int index, int aim, int[][] map) {
        int res = 0;
        if (index == arr.length) {
            res = aim == 0 ? 1 : 0;
        } else {
            int mapValue = 0;
            for (int i = 0; arr[index] * i <= aim; i++) {
                mapValue = map[index + 1][aim - arr[index] * i];
                if (mapValue != 0) {
                    res += mapValue == -1 ? 0 : mapValue;
                } else {
                    res += process2(arr, index + 1, aim - arr[index] * i, map);
                }
            }
        }
        map[index][aim] = res == 0 ? -1 : res;
        return res;
}
```

记忆化搜索的方法是针对暴力递归最初级的优化技巧，分析递归函数的状态由哪些变量可以表示，做出相应维度和大小的map即可。记忆化搜索方法的时间复杂度为O(N*(aim^2))，我们在解释完下面的方法后，再来具体解释为什么是这个时间复杂度。

动态规划方法。生成行数为N，列数为aim+1的矩阵dp，dp[i][j]的含义是在使用arr[0..i]货币的情况下，组成钱数j有多少种方法。dp[i][j]的值求法如下：
1，对于矩阵dp第一列的值dp[..][0]，表示组成钱数为0的方法数，很明显是1种，也就是不使用任何货币。所以dp第一列的值统一设置为1。
2，对于矩阵dp第一行的值dp[0][..]，表示只能使用arr[0]这一种货币的情况下，组成钱的方法数，比如arr[0]==5时，能组成的钱数只有0，5，10，15...。所以令dp[0][k*arr[0]]=1(0<=k*arr[0]<=aim，k为非负整数)。
3，除了第一行和第一列的其他位置，记为位置(i,j)。dp[i][j]的值是以下几个值的累加。
　　0）完全不用arr[i]货币，只使用arr[0..i-1]货币时，方法数为dp[i-1][j]。
　　1）用1张arr[i]货币，剩下的钱用arr[0..i-1]货币组成时，方法数为dp[i-1][j-arr[i]]。

　　2）用2张arr[i]货币，剩下的钱用arr[0..i-1]货币组成时，方法数为dp[i-1][j-2*arr[i]]。
　　...
　　k）用k张arr[i]货币，剩下的钱用arr[0..i-1]货币组成时，方法数为dp[i-1][j-k*arr[i]]。j-k*arr[i]>=0，k为非负整数。
4，最终dp[N-1][aim]的值就是最终结果。
具体过程请参看如下代码中的coins3方法。

```
	public int coins3(int[] arr, int aim) {
		if (arr == null || arr.length == 0 || aim < 0) {
			return 0;
		}
		int[][] dp = new int[arr.length][aim + 1];
		for (int i = 0; i < arr.length; i++) {
			dp[i][0] = 1;
		}
		for (int j = 1; arr[0] * j <= aim; j++) {
			dp[0][arr[0] * j] = 1;
		}
		int num = 0;
		for (int i = 1; i < arr.length; i++) {
			for (int j = 1; j <= aim; j++) {
				num = 0;
				for (int k = 0; j - arr[i] * k >= 0; k++) {
					num += dp[i - 1][j - arr[i] * k];
				}
				dp[i][j] = num;
			}
		}
		return dp[arr.length - 1][aim];
	}
```

在最差情况下对于位置(i,j)来说，求解dp[i][j]的计算过程需要枚举dp[i-1][0..j]上的所有值，dp一共有N*aim个位置，所以总体的时间复杂度为O(N*(aim^2))。
下面解释之前记忆化搜索方法的时间复杂度为什么也是O(N*(aim^2))，因为在本质上记忆化搜索方法等价于动态规划方法。记忆化搜索的方法说白了就是不关心到达某一个递归过程的路径，只是单纯的对计算过的递归过程进行记录，避免重复的递归过程，而动态规划的方法则是规定好每一个递归过程的计算顺序，依次进行计算，后计算的过程严格依赖前面计算过的过程。两者都是空间换时间的方法，也都有枚举的过程，区别就在于动态规划规定计算顺序而记忆搜索不用规定。所以记忆化搜索方法的时间复杂度也是O(N*(aim^2))。两者各有优缺点，如果对暴力递归过程简单地优化成记忆搜索的方法，递归函数依然在使用，这在工程上的开销较大。而动态规划方法严格规定了计算顺序，可以将递归计算变成顺序计算，这是动态规划方法很大的优势。其实记忆搜索的方法也有优势，本题就很好的体现了。比如arr=[20000,10000,1000]，aim=2000000000。如果是动态规划的计算方法，要严格计算3*2000000000个位置。而对于记忆搜索来说，因为面值最小的钱为1000，所以百位为(1~9)或十位为(1~9)或各位为(1~9)的钱数是不可能出现的，当然也就不必要计算。通过本例可以知道，记忆化搜索是对必须要计算的递归过程才去计算并记录的。

接下来介绍时间复杂度为O(N*aim)的动态规划方法。我们来看上一个动态规划方法中，求dp[i][j]值的时候的步骤3，这也是最关键的枚举过程：
3，除了第一行和第一列的其他位置，记为位置(i,j)。dp[i][j]的值是以下几个值的累加。
　　0）完全不用arr[i]货币，只使用arr[0..i-1]货币时，方法数为dp[i-1][j]。
　　1）用1张arr[i]货币，剩下的钱用arr[0..i-1]货币组成时，方法数为dp[i-1][j-arr[i]]。

　　2）用2张arr[i]货币，剩下的钱用arr[0..i-1]货币组成时，方法数为dp[i-1][j-2*arr[i]]。
　　...
　　k）用k张arr[i]货币，剩下的钱用arr[0..i-1]货币组成时，方法数为dp[i-1][j-k*arr[i]]。j-k*arr[i]>=0，k为非负整数。
步骤3中，情况0）的方法数为dp[i-1][j]，而情况1）一直到情况k）的方法数累加值其实就是dp[i][j-arr[i]]的值。所以步骤3可以化简为dp[i][j]=dp[i-1][j]+dp[i][j-arr[i]]。一下省去了枚举的过程，时间复杂度也减小至O(N*aim)，具体请参看如下代码中的coins4方法。

```java
public int coins4(int[] arr, int aim) {
	if (arr == null || arr.length == 0 || aim < 0) {
		return 0;
	}
	int[][] dp = new int[arr.length][aim + 1];
	for (int i = 0; i < arr.length; i++) {
		dp[i][0] = 1;
	}
	for (int j = 1; arr[0] * j <= aim; j++) {
		dp[0][arr[0] * j] = 1;
	}
	for (int i = 1; i < arr.length; i++) {
		for (int j = 1; j <= aim; j++) {
			dp[i][j] = dp[i - 1][j];
			dp[i][j] += j - arr[i] >= 0 ? dp[i][j - arr[i]] : 0;
		}
	}
	return dp[arr.length - 1][aim];
}
```

时间复杂度为O(N*aim)的动态规划方法再结合空间压缩的技巧。空间压缩的原理请读者参考本书“矩阵的最小路径和”问题，这里不再详述。请参看如下代码中的coins5方法。

```java
public int coins5(int[] arr, int aim) {
	if (arr == null || arr.length == 0 || aim < 0) {
		return 0;
	}
	int[] dp = new int[aim + 1];
	for (int j = 0; arr[0] * j <= aim; j++) {
		dp[arr[0] * j] = 1;
	}
	for (int i = 1; i < arr.length; i++) {
		for (int j = 1; j <= aim; j++) {
			dp[j] += j - arr[i] >= 0 ? dp[j - arr[i]] : 0;
		}
	}
	return dp[aim];
}
```

至此我们得到了最优解，是时间复杂度为O(N*aim)，额外空间复杂度O(aim)的方法。

【扩展】

通过本题目的优化过程，可以梳理出暴力递归通用的优化过程。对于在面试中遇到的具体题目，面试者一旦想到暴力递归的过程，其实之后的优化过程是水到渠成的。首先看看你写出来的暴力递归函数，找出有哪些参数是不发生变化的，对于这些变量我们忽略。只看那些变化并且可以表示递归过程的参数，找出这些参数之后，记忆搜索的方法其实可以很轻易的写出来，因为只是简单的修改，计算完就记录到map中，下次直接拿来使用，没计算过就依然进行递归计算即可。接下来观察记忆搜索过程中使用的map结构，看看该结构某一个具体的位置的值是通过哪些位置的值求出的，被依赖的位置先求，就能改出动态规划的方法。改出的动态规划方法中，如果有枚举的过程，看看枚举过程是否可以继续优化，常规的方法既有本题所实现的通过表达式来化简枚举状态的方式，也有本书的“丢棋子问题”、“画匠问题”和“邮局选址问题”所涉及的四边形不等式的相关内容，有兴趣的读者可以进一步的学习。