个数组滚动更新的方式无疑节省了大量的空间。没有优化之前，取得某个位置动态规划值的过程是在矩阵中进行两次寻址，优化后，这一过程只需要一次寻址，程序的常数时间也得到了一定程度的加速。但是空间压缩的方法是有局限性的，本题如果改成“打印具有最小路径和的路径”，那么就不能使用空间压缩的方法。如果类似本题这种需要二维表的动态规划题目，最终目的是想求最优解的具体路径，往往需要完整的动态规划表，但如果只是想求最优解的值，则可以使用空间压缩的方法。因为空间压缩的方法是滚动更新的，会覆盖之前求解的值，让求解轨迹变得不可回溯。希望读者好好研究这种空间压缩的实现技巧，本书还有许多动态规划题目会涉及空间压缩方法的实现。

换钱的最少货币数

【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，求组成 aim 的最少货币数。

【举例】

arr=[5,2,3]，aim=20。

4 张 5 元可以组成 20 元，其他的找钱方案都要使用更多张的货币，所以返回 4。

arr=[5,2,3]，aim=0。

不用任何货币就可以组成 0 元，返回 0。

arr=[3,5]，aim=2。

根本无法组成 2 元，钱不能找开的情况下默认返回-1。

【补充题目】

给定数组 arr，arr 中所有的值都为正数。每个值仅代表一张钱的面值，再给定一个整数 aim 代表要找的钱数，求组成 aim 的最少货币数。

【举例】

arr=[5,2,3]，aim=20。

5 元、2 元和 3 元的钱各有 1 张，所以无法组成 20 元，默认返回-1。

arr=[5,2,5,3]，aim=10。

5 元的货币有 2 张，可以组成 10 元，且该方案所需张数最少，返回 2。

arr=[5,2,5,3]，aim=15。

所有的钱加起来才能组成 15 元，返回 4。

arr=[5,2,5,3]，aim=0。

不用任何货币就可以组成 0 元，返回 0。

【难度】

尉 ★★☆☆

【解答】

原问题的经典动态规划方法。如果 arr 的长度为 N，生成行数为 N、列数为 aim+1 的动态规划表的 dp。dp[i][j]的含义是，在可以任意使用 arr[0..i]货币的情况下，组成 j 所需的最小张数。根据这个定义，dp[i][j]的值按如下方式计算：

1. dp[0..N-1][0]的值（即 dp 矩阵中第一列的值）表示找的钱数为 0 时需要的最少张数，钱数为 0 时，完全不需要任何货币，所以全设为 0 即可。

2. dp[0][0..aim]的值（即 dp 矩阵中第一行的值）表示只能使用 arr[0]货币的情况下，找某个钱数的最小张数。比如，arr[0]=2，那么能找开的钱数为 2，4，6，8，...所以令 dp[0][2]=1，dp[0][4]=2，dp[0][6]=3，...第一行其他位置所代表的钱数一律找不开，所以一律设为 32 位整数的最大值，我们把这个值记为 max。

3. 剩下的位置依次从左到右，再从上到下计算。假设计算到位置(i,j)，dp[i][j]的值可能来自下面的情况。

- 完全不使用当前货币 arr[i]情况下的最少张数，即 dp[i-1][j]的值。
- 只使用 1 张当前货币 arr[i]情况下的最少张数，即 dp[i-1][j-arr[i]]+1。
- 只使用 2 张当前货币 arr[i]情况下的最少张数，即 dp[i-1][j-2*arr[i]]+2。
- 只使用 3 张当前货币 arr[i]情况下的最少张数，即 dp[i-1][j-3*arr[i]]+3。

所有的情况中，最终取张数最小的。所以

dp[i][j]=min{dp[i-1][j-k*arr[i]]+k(0<=k)}

=>dp[i][j]=min{dp[i-1][j], min{dp[i-1][j-x*arr[i]]+x(1<=x)}}

=>dp[i][j]=min{dp[i-1][j], min{dp[i-1][j-arr[i]-y*arr[i]]+y+1(0<=y)}}

又有 min{dp[i-1][j-arr[i]-y*arr[i]]+y(0<=y)} => dp[i][j-arr[i]]，所以，最终有：dp[i][j]=min{dp[i-1][j],dp[i][j-arr[i]]+1}。如果 j-arr[i]<0，即发生越界了，说明 arr[i]太大，用一张都会超过钱数 j，令 dp[i][j]=dp[i-1][j]即可。具体过程请参看如下代码中的 minCoins1 方法，整个过程的时间复杂度与额外空间复杂度都为 $O(N\times \text{aim})$，N 为 arr 的长度。

```
public int minCoins1(int[] arr, int aim) {
	if (arr == null || arr.length == 0 || aim < 0) {
		return -1;
	}
	int n = arr.length;
	int max = Integer.MAX_VALUE;
	int[][] dp = new int[n][aim + 1];
	for (int j = 1; j <= aim; j++) {
		dp[0][j] = max;
		if (j - arr[0] >= 0 && dp[0][j - arr[0]] != max) {
			dp[0][j] = dp[0][j - arr[0]] + 1;
		}
	}
	int left = 0;
	for (int i = 1; i < n; i++) {
		for (int j = 1; j <= aim; j++) {
			left = max;
			if (j - arr[i] >= 0 && dp[i][j - arr[i]] != max) {
				left = dp[i][j - arr[i]] + 1;
			}
			dp[i][j] = Math.min(left, dp[i - 1][j]);
		}
	}
	return dp[n - 1][aim] != max ? dp[n - 1][aim] : -1;
}
```

原问题在动态规划基础上的空间压缩方法。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。我们选择生成一个长度为 aim+1 的动态规划一维数组 dp，然后按行来更新 dp 即可。之所以不选按列更新，是因为根据 dp[i][j]=min{dp[i-1][j], dp[i][j-arr[i]]+1}可知，位置(i,j)依赖位置$(i-1,j)$，即往上跳一下的位置，也依赖位置(i,j-arr[i])，即往左跳 arr[i]一下的位置，所以按行更新只需要 1 个一维数组，按列更新需要的一维数组个数就与 arr 中货币的最大值有关，如最大的货币为 a，说明最差情况下要向左侧跳 a 下，相应地，就要准备 a 个一维数组不断地滚动复用，这样实现起来很麻烦，所以不采用按列更新的方式。具体请参看如下代码中的 minCoins2 方法，空间压缩之后时间复杂度为 $O(N\times \text{aim})$，额外空间复杂度为 $O(\text{aim})$。

```
public int minCoins2(int[] arr, int aim) {
	if (arr == null || arr.length == 0 || aim < 0) {
```

```
            return -1;
        }
        int n = arr.length;
        int max = Integer.MAX_VALUE;
        int[] dp = new int[aim + 1];
        for (int j = 1; j <= aim; j++) {
            dp[j] = max;
            if (j - arr[0] >= 0 && dp[j - arr[0]] != max) {
                dp[j] = dp[j - arr[0]] + 1;
            }
        }
        int left = 0;
        for (int i = 1; i < n; i++) {
            for (int j = 1; j <= aim; j++) {
                left = max;
                if (j - arr[i] >= 0 && dp[j - arr[i]] != max) {
                    left = dp[j - arr[i]] + 1;
                }
                dp[j] = Math.min(left, dp[j]);
            }
        }
        return dp[aim] != max ? dp[aim] : -1;
    }
```

补充问题的经典动态规划方法。如果 arr 的长度为 N，生成行数为 N、列数为 aim+1 的动态规划表的 dp。dp[i][j]的含义是，在可以任意使用 arr[0..i]货币的情况下（每个值仅代表一张货币），组成 j 所需的最小张数。根据这个定义，dp[i][j]的值按如下方式计算：

1. dp[0..N-1][0]的值（即 dp 矩阵中第一列的值）表示找的钱数为 0 时需要的最少张数，钱数为 0 时完全不需要任何货币，所以全设为 0 即可。

2．dp[0][0..aim]的值（即 dp 矩阵中第一行的值）表示只能使用一张 arr[0]货币的情况下，找某个钱数的最小张数。比如 arr[0]=2，那么能找开的钱数仅为 2，所以令 dp[0][2]=1。因为只有一张钱，所以其他位置所代表的钱数一律找不开，一律设为 32 位整数的最大值。

3．剩下的位置依次从左到右，再从上到下计算。假设计算到位置(i,j)，dp[i][j]的值可能来自下面两种情况。

1）dp[i-1][j]的值代表在可以任意使用 arr[0..i-1]货币的情况下，组成 j 所需的最小张数。可以任意使用 arr[0..i]货币的情况当然包括不使用这一张面值为 arr[i]的货币，而只任意使用 arr[0..i-1]货币的情况，所以 dp[i][j]的值可能等于 dp[i-1][j]。

2）因为 arr[i]只有一张不能重复使用，所以我们考虑 dp[i-1][j-arr[i]]的值，这个值代表在可以任意使用 arr[0..i-1]货币的情况下，组成 j-arr[i]所需的最小张数。从钱数为 j-arr[i]到钱数 j，只用再加上当前的这张 arr[i]即可。所以 dp[i][j]的值可能等于 dp[i-1][j-arr[i]]+1。

4．如果 dp[i-1][j-arr[i]]中 j-arr[i]<0，也就是位置越界了，说明 arr[i]太大，只用一张都

会超过钱数 j，令 dp[i][j]=dp[i-1][j]即可。否则 dp[i][j]=min{dp[i-1][j]，dp[i-1][j-arr[i]]+1}。

具体过程请参看如下代码中的 minCoins3 方法，整个过程的时间复杂度与额外空间复杂度都为 $O(N\times \text{aim})$，N 为 arr 的长度。

```
public int minCoins3(int[] arr, int aim) {
	if (arr == null || arr.length == 0 || aim < 0) {
		return -1;
	}
	int n = arr.length;
	int max = Integer.MAX_VALUE;
	int[][] dp = new int[n][aim + 1];
	for (int j = 1; j <= aim; j++) {
		dp[0][j] = max;
	}
	if (arr[0] <= aim) {
		dp[0][arr[0]] = 1;
	}
	int leftup = 0; // 左上角某个位置的值
	for (int i = 1; i < n; i++) {
		for (int j = 1; j <= aim; j++) {
			leftup = max;
			if (j - arr[i] >= 0 && dp[i - 1][j - arr[i]] != max) {
				leftup = dp[i - 1][j - arr[i]] + 1;
			}
			dp[i][j] = Math.min(leftup, dp[i - 1][j]);
		}
	}
	return dp[n - 1][aim] != max ? dp[n - 1][aim] : -1;
}
```

进阶问题在动态规划基础上的空间压缩方法。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。我们选择生成一个长度为 aim+1 的动态规划一维数组 dp，然后按行来更新 dp 即可，不选按列更新的方式与原问题同理。具体请参看如下代码中的 minCoins4 方法，空间压缩之后时间复杂度为 $O(N\times \text{aim})$，额外空间复杂度为 $O(\text{aim})$。

```
public int minCoins4(int[] arr, int aim) {
	if (arr == null || arr.length == 0 || aim < 0) {
		return -1;
	}
	int n = arr.length;
	int max = Integer.MAX_VALUE;
	int[] dp = new int[aim + 1];
	for (int j = 1; j <= aim; j++) {
		dp[j] = max;
	}
	if (arr[0] <= aim) {
		dp[arr[0]] = 1;
	}
```

```
        int leftup = 0; // 左上角某个位置的值
        for (int i = 1; i < n; i++) {
                for (int j = aim; j > 0; j--) {
                        leftup = max;
                        if (j - arr[i] >= 0 && dp[j - arr[i]] != max) {
                                leftup = dp[j - arr[i]] + 1;
                        }
                        dp[j] = Math.min(leftup, dp[j]);
                }
        }
        return dp[aim] != max ? dp[aim] : -1;
}
```

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 元和 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】

本书将由浅入深地给出所有的解法，最后解释最优解。这道题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。