```
		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。

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]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]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]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]组成剩下的 0，最终方法数记为 res201。

那么 res1+res2+…+res201 的值就是总的方法数。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]这些面值的钱组成 aim，返回总的方法数。具体实现参见如下代码中的 coins1 方法。

```
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。两种情况下都需要求 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，并且与 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 中查询这个递归过程是否已经计算过，如果已经计算过，就把值拿出来直接用，如果没计算过，需要再进入递归过程。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可以由哪些变量表示，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 $O(N\times \text{aim}^2)$，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。

- 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]。
- 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]]。
- 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]]。
- ……
- 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];
}
```

在最差的情况下，对位置(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 $N\times$aim 个位置，所以总体的时间复杂度为 $O(N\times\text{aim}^2)$。

下面解释之前记忆化搜索方法的时间复杂度为什么也是 $O(N\times\text{aim}^2)$，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，只是单纯地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 $O(N\times aim^2)$。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，这是动态规划方法具有的优势。其实记忆搜索的方法也有优势，本题就很好地体现了。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)、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 $O(N\times aim)$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。

- 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]。
- 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]]。
- 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]]。
- ……
- 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，第 1 种情况的方法数为 dp[i-1][j]，而第 2 种情况一直到第 k 种情况的方法数累加值其实就是 dp[i][j-arr[i]]的值。所以步骤 3 可以简化为 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 $O(N\times aim)$，具体请参看如下代码中的 coins4 方法。

```
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 $O(N\times aim)$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。

```
	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 $O(N\times aim)$、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。首先看写出来的暴力递归函数，找出有哪些参数是不发生变化的，忽略这些变量。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地写出来，因为只是简单的修改，计算完就记录到 map 中，并在下次直接拿来使用，没计算过则依然进行递归计算。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

步学习。

最长递增子序列

【题目】

给定数组 arr，返回 arr 的最长递增子序列。

【举例】

arr=[2,1,5,3,6,4,8,9,7]，返回的最长递增子序列为[1,3,4,8,9]。

【要求】

如果 arr 长度为 N，请实现时间复杂度为 $O(N\log N)$的方法。

【难度】

校 ★★★☆

【解答】

先介绍时间复杂度为 $O(N^2)$的方法，具体过程如下：

1．生成长度为 N 的数组 dp，dp[i]表示在以 arr[i]这个数结尾的情况下，arr[0..i]中的最大递增子序列长度。

2．对第一个数 arr[0]来说，令 dp[0]=1，接下来从左到右依次算出以每个位置的数结尾的情况下，最长递增子序列长度。

3．假设计算到位置 i，求以 arr[i]结尾情况下的最长递增子序列长度，即 dp[i]。如果最长递增子序列以 arr[i]结尾，那么在 arr[0..i-1]中所有比 arr[i]小的数都可以作为倒数第二个数。在这么多倒数第二个数的选择中，以哪个数结尾的最大递增子序列更大，就选那个数作为倒数第二个数，所以 dp[i]=max{dp[j]+1(0<=j<i，arr[j]<arr[i])}。如果 arr[0..i-1]中所有的数都不比 arr[i]小，令 dp[i]=1 即可，说明以 arr[i]结尾情况下的最长递增子序列只包含 arr[i]。

按照步骤 1~3 可以计算出 dp 数组，具体过程请参看如下代码中的 getdp1 方法。

```
public int[] getdp1(int[] arr) {
        int[] dp = new int[arr.length];
        for (int i = 0; i < arr.length; i++) {
```