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，母牛开始生小母牛。每只小母牛 3 年之后成熟又可以生小母牛。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】

对以上所有的问题，请实现时间复杂度 $O(\log N)$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 $F(N)=F(N-1)+F(N-2)$，于是很轻松地写出暴力递归的代码。请参看如下代码中的 f1 方法。

```
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。

```
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;
        }
```

$O(\log N)$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 $O(\log N)$。$F(n)=F(n-1)+F(n-2)$，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：

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

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：

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

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

$$\vdots$$

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

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能够在 $O(\log N)$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 $O(\log N)$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，如何最快地求解 10 的 75 次方。

1．75 的二进制数形式为 1001011。

2．10 的 75 次方=$10^{64}\times10^{8}\times10^{2}\times10^{1}$。

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}，即 75 的二进制数形式总共有多少位，我们就使用了几次乘法。

3．在步骤 2 进行的过程中，把应该累乘的值相乘即可，比如 10^{64}、10^8、10^2、10^1 应该累乘，因为 64、8、2、1 对应到 75 的二进制数中，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 的二进制数中，相应的位上是 0。

对矩阵来说同理，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。

```
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，相当于整数中的 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。

```
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 级，方法只有 1 种。如果台阶有 2 级，方法有 2 种。如果台阶有 *N* 级，最后跳上第 *N* 级的情况，要么是从 *N*-2 级台阶直接跨 2 级台阶，要么是从 *N*-1 级台阶跨 1 级台阶，所以台阶有 *N* 级的方法数为跨到 *N*-2 级台阶的方法数加上跨到 *N*-1

级台阶的方法数，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。可以很轻易地写出 $O(2^N)$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;
}
```

$O(\log N)$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：

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

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;
        }
        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。所有的牛都不会死，所以第 N-1 年的牛会毫无损失地活到第 N 年。同时所有成熟的牛都会生 1 头新的牛，那么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有成熟。所以 $C(n)=C(n\text{-}1)+C(n\text{-}3)$，初始项为 C(1)==1，C(2)==2，C(3)==3。这个和斐波那契数列又十分类似，只不过 $C(n)$依赖 $C(n\text{-}1)$和 $C(n\text{-}3)$的值，而斐波那契数列 $F(n)$依赖 $F(n\text{-}1)$和 $F(n\text{-}2)$的值。同样可以很轻易地写出 $O(2^N)$与 $O(N)$的方法，请参看如下代码中的 c1 和 c2 方法。

```
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;
}
```

$O(\log N)$的方法。$C(n)=C(n\text{-}1)+C(n\text{-}3)$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。

$$(C_n, C_{n-1}, C_{n-2}) = (C_{n-1}, C_{n-2}, C_{n-3}) \times \begin{vmatrix} a & b & c \\ d & e & f \\ g & h & i \end{vmatrix}$$

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：

$$\begin{vmatrix} a & b & c \\ d & e & f \\ g & h & i \end{vmatrix} = \begin{vmatrix} 1 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{vmatrix}$$

求矩阵之后，当 n>3 时，原来的公式可化简为：

$$(C_n, C_{n-1}, C_{n-2}) = (C_3, C_2, C_1) \times \begin{vmatrix} 1 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{vmatrix}^{n-3} = (3,2,1) \times \begin{vmatrix} 1 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{vmatrix}^{n-3}$$

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 $F(n)=a\times F(n\text{-}1)+b\times F(n\text{-}2)+\ldots+k\times F(n\text{-}i)$，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 $O(\log N)$。

矩阵的最小路径和

【题目】

给定一个矩阵 m，从左上角开始每次只能向右或者向下走，最后到达右下角的位置，路径上所有的数字累加起来就是路径和，返回所有的路径中最小的路径和。

【举例】

如果给定的 m 如下：

1　3　5　9

8 1 3 4
5 0 6 1
8 8 4 0

路径 1，3，1，0，6，1，0 是所有路径中路径和最小的，所以返回 12。

【难度】

尉 ★★☆☆

【解答】

经典动态规划方法。假设矩阵 *m* 的大小为 *M*×*N*，行数为 *M*，列数为 *N*。先生成大小和 *m* 一样的矩阵 dp，dp[i][j]的值表示从左上角（即(0,0)）位置走到(*i*,*j*)位置的最小路径和。对 *m* 的第一行的所有位置来说，即(0,*j*)(0≤*j*<*N*)，从(0,0)位置走到(0,*j*)位置只能向右走，所以(0,0)位置到(0,*j*)位置的路径和就是 m[0][0..j]这些值的累加结果。同理，对 *m* 的第一列的所有位置来说，即(*i*,0)(0≤*i*<*M*)，从(0,0)位置走到(*i*,0)位置只能向下走，所以(0,0)位置到(*i*,0)位置的路径和就是 m[0..i][0]这些值的累加结果。以题目中的例子来说，dp 第一行和第一列的值如下：

1 4 9 1 8
9
14
22

除第一行和第一列的其他位置(*i*,*j*)外，都有左边位置(*i*-1,*j*)和上边位置(*i*,*j*-1)。从(0,0)到(*i*,*j*)的路径必然经过位置(*i*-1,*j*)或位置(*i*,*j*-1)，所以，dp[i][j]=min{dp[i-1][j],dp[i][j-1]}+m[i][j]，含义是比较从(0,0)位置开始，经过(*i*-1,*j*)位置最终到达(*i*,*j*)的最小路径和经过(*i*,*j*-1)位置最终到达(*i*,*j*)的最小路径之间，哪条路径的路径和更小。那么更小的路径和就是 dp[i][j]的值。以题目的例子来说，最终生成的 dp 矩阵如下：

1 4 9 18
9 5 8 12
14 5 11 12
22 13 15 12

除第一行和第一列之外，每一个位置都考虑从左边到达自己的路径和更小还是从上边达到自己的路径和更小。最右下角的值就是整个问题的答案。具体过程请参看如下代码中

的 minPathSum1 方法。

```
public int minPathSum1(int[][] m) {
        if (m == null || m.length == 0 || m[0] == null || m[0].length == 0) {
                return 0;
        }
        int row = m.length;
        int col = m[0].length;
        int[][] dp = new int[row][col];
        dp[0][0] = m[0][0];
        for (int i = 1; i < row; i++) {
                dp[i][0] = dp[i - 1][0] + m[i][0];
        }
        for (int j = 1; j < col; j++) {
                dp[0][j] = dp[0][j - 1] + m[0][j];
        }
        for (int i = 1; i < row; i++) {
                for (int j = 1; j < col; j++) {
                   dp[i][j] = Math.min(dp[i - 1][j], dp[i][j - 1]) + m[i][j];
                }
        }
        return dp[row - 1][col - 1];
}
```

矩阵中一共有 $M\times N$ 个位置，每个位置都计算一次从(0,0)位置达到自己的最小路径和，计算的时候只是比较上边位置的最小路径和与左边位置的最小路径和哪个更小，所以时间复杂度为 $O(M\times N)$，dp 矩阵的大小为 $M\times N$，所以额外空间复杂度为 $O(M\times N)$。

动态规划经过空间压缩后的方法。这道题的经典动态规划方法在经过空间压缩之后，时间复杂度依然是 $O(M\times N)$，但是额外空间复杂度可以从 $O(M\times N)$减小至 $O(\min\{M,N\})$，也就是不使用大小为 $M\times N$ 的 dp 矩阵，而仅仅使用大小为 $\min\{M,N\}$的 arr 数组。具体过程如下（以题目的例子来举例说明）：

1．生成长度为 4 的数组 arr，初始时 arr=[0,0,0,0]，我们知道从(0,0)位置到达 m 中第一行的每个位置，最小路径和就是从(0,0)位置的值开始依次累加的结果，所以依次把 arr 设置为 arr=[1,4,9,18]，此时 arr[j]的值代表从(0,0)位置达到(0,j)位置的最小路径和。

2．步骤 1 中 arr[j]的值代表从(0,0)位置达到(0,j)位置的最小路径和，在这一步中想把 arr[j]的值更新成从(0,0)位置达到(1,j)位置的最小路径和。首先来看 arr[0]，更新之前 arr[0]的值代表(0,0)位置到达(0,0)位置的最小路径和(dp[0][0])，如果想把 arr[0]更新成从(0,0)位置达到(1,0)位置的最小路径和(dp[1][0])，令 arr[0]=arr[0]+m[1][0]=9 即可。然后来看 arr[1]，更新之前 arr[1]的值代表(0,0)位置到达(0,1)位置的最小路径和(dp[0][1])，更新之后想让 arr[1]代表(0,0)位置到达(1,1)位置的最小路径和(dp[1][1])。根据动态规划的求解过程，到达(1,1)位

置有两种选择，一种是从(1,0)位置到达(1,1)位置(dp[1][0]+m[1][1])，另一种是从(0,1)位置到达(1,1)位置(dp[0][1]+m[1][1])，应该选择路径和最小的那个。此时 arr[0]的值已经更新成 dp[1][0]，arr[1]目前还没有更新，所以，arr[1]还是 dp[0][1]，arr[1]=min{arr[0],arr[1]}+m[1][1]=5。更新之后，arr[1]的值变为 dp[1][1]的值。同理，arr[2]=min{arr[1],arr[2]}+m[1][2]，...。最终 arr 可以更新成[9,5,8,12]。

3．重复步骤 2 的更新过程，一直到 arr 彻底变成 dp 矩阵的最后一行。整个过程其实就是不断滚动更新 arr 数组，让 arr 依次变成 dp 矩阵每一行的值，最终变成 dp 矩阵最后一行的值。

本题的例子是矩阵 *m* 的行数等于列数，如果给定的矩阵列数小于行数（*N*<*M*），依然可以用上面的方法令 arr 更新成 dp 矩阵每一行的值。但如果给定的矩阵行数小于列数（*M*<*N*），那么就生成长度为 *M* 的 arr，然后令 arr 更新成 dp 矩阵每一列的值，从左向右滚动过去。以本例来说，如果按列来更新，arr 首先更新成[1,9,14,22]，然后向右滚动更新成[4,5,5,13]，继续向右滚动更新成[9,8,11,15]，最后是[18,12,12,12]。总之，是根据给定矩阵行和列的大小关系决定滚动的方式，始终生成最小长度(min{*M*,*N*})的 arr 数组。具体过程请参看如下代码中的 minPathSum2 方法。

```
public int minPathSum2(int[][] m) {
        if (m == null || m.length == 0 || m[0] == null || m[0].length == 0) {
                return 0;
        }
        int more = Math.max(m.length, m[0].length); // 行数与列数较大的那个为 more
        int less = Math.min(m.length, m[0].length); // 行数与列数较小的那个为 less
        boolean rowmore = more == m.length; // 行数是不是大于等于列数
        int[] arr = new int[less]; // 辅助数组的长度仅为行数与列数中的最小值
        arr[0] = m[0][0];
        for (int i = 1; i < less; i++) {
                arr[i] = arr[i - 1] + (rowmore ? m[0][i] : m[i][0]);
        }
        for (int i = 1; i < more; i++) {
                arr[0] = arr[0] + (rowmore ? m[i][0] : m[0][i]);
                for (int j = 1; j < less; j++) {
                        arr[j] = Math.min(arr[j - 1], arr[j])
                                        + (rowmore ? m[i][j] : m[j][i]);
                }
        }
        return arr[less - 1];
}
```

【扩展】

本题压缩空间的方法几乎可以应用到所有需要二维动态规划表的面试题目中，通过一

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*×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*×aim)，额外空间复杂度为 *O*(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 $O(N\times aim)$，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 $O(N\times aim)$，额外空间复杂度为 $O(aim)$。

```
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。

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 $O(\text{aim}^N)$。

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。

```
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 $O(N\times aim^2)$，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*×aim 个位置，所以总体的时间复杂度为 $O(N\times aim^2)$。

下面解释之前记忆化搜索方法的时间复杂度为什么也是 $O(N\times aim^2)$，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++) {
```

```
                dp[i] = 1;
                for (int j = 0; j < i; j++) {
                        if (arr[i] > arr[j]) {
                                dp[i] = Math.max(dp[i], dp[j] + 1);
                        }
                }
        }
        return dp;
}
```

接下来解释如何根据求出的 dp 数组得到最长递增子序列。以题目的例子来说明，arr=[2,1,5,3,6,4,8,9,7]，求出的数组 dp=[1,1,2,2,3,3,4,5,4]。

1．遍历 dp 数组，找到最大值以及位置。在本例中最大值为 5，位置为 7，说明最终的最长递增子序列的长度为 5，并且应该以 arr[7]这个数(arr[7]==9)结尾。

2．从 arr 数组的位置 7 开始从右向左遍历。如果对某一个位置 i，既有 arr[i]<arr[7]，又有 dp[i]==dp[7]-1，说明 arr[i]可以作为最长递增子序列的倒数第二个数。在本例中，arr[6]<arr[7]，并且 dp[6]==dp[7]-1，所以 8 应该作为最长递增子序列的倒数第二个数。

3．从 arr 数组的位置 6 开始继续向左遍历，按照同样的过程找到倒数第三个数。在本例中，位置 5 满足 arr[5]<arr[6]，并且 dp[5]==dp[6]-1，同时位置 4 也满足。选 arr[5]或者 arr[4]作为倒数第三个数都可以。

4．重复这样的过程，直到所有的数都找出来。

dp 数组包含每一步决策的信息，其实根据 dp 数组找出最长递增子序列的过程就是从某一个位置开始逆序还原出决策路径的过程。具体过程请参看如下代码中的 generateLIS 方法。

```
public int[] generateLIS(int[] arr, int[] dp) {
        int len = 0;
        int index = 0;
        for (int i = 0; i < dp.length; i++) {
                if (dp[i] > len) {
                        len = dp[i];
                        index = i;
                }
        }
        int[] lis = new int[len];
        lis[--len] = arr[index];
        for (int i = index; i >= 0; i--) {
                if (arr[i] < arr[index] && dp[i] == dp[index] - 1) {
                        lis[--len] = arr[i];
                        index = i;
                }
        }
        return lis;
```

```
    }
```

整个过程的主方法参看如下代码中的 lis1 方法。

```
public int[] lis1(int[] arr) {
        if (arr == null || arr.length == 0) {
                return null;
        }
        int[] dp = getdp1(arr);
        return generateLIS(arr, dp);
}
```

很明显，计算 dp 数组过程的时间复杂度为 $O(N^2)$，根据 dp 数组得到最长递增子序列过程的时间复杂度为 $O(N)$，所以整个过程的时间复杂度为 $O(N^2)$。如果让时间复杂度达到 $O(N\log N)$，只要让计算 dp 数组的过程达到时间复杂度 $O(N\log N)$即可，之后根据 dp 数组生成最长递增子序列的过程是一样的。

时间复杂度 $O(N\log N)$生成 dp 数组的过程是利用二分查找来进行的优化。先生成一个长度为 N 的数组 ends，初始时 ends[0]=arr[0]，其他位置上的值为 0。生成整型变量 right，初始时 right=0。在从左到右遍历 arr 数组的过程中，求解 dp[i]的过程需要使用 ends 数组和 right 变量，所以这里解释一下其含义。遍历的过程中，ends[0..right]为有效区，ends[right+1..N-1]为无效区。对有效区上的位置 b，如果有 ends[b]==c，则表示遍历到目前为止，在所有长度为 b+1 的递增序列中，最小的结尾数是 c。无效区的位置则没有意义。

比如，arr=[2,1,5,3,6,4,8,9,7]，初始时 dp[0]=1，ends[0]=2，right=0。ends[0..0]为有效区，ends[0]==2 的含义是，在遍历过 arr[0]之后，所有长度为 1 的递增序列中(此时只有[2])，最小的结尾数是 2。之后的遍历继续用这个例子来说明求解过程。

1．遍历到 arr[1]==1。ends 有效区=ends[0..0]=[2]，在有效区中找到最左边的大于或等于 arr[1]的数。发现是 ends[0]，表示以 arr[1]结尾的最长递增序列只有 arr[1]，所以令 dp[1]=1。然后令 ends[0]=1，因为遍历到目前为止，在所有长度为 1 的递增序列中，最小的结尾数是 1，而不再是 2。

2．遍历到 arr[2]==5。ends 有效区=ends[0..0]=[1]，在有效区中找到最左边大于或等于 arr[2]的数。发现没有这样的数，表示以 arr[2]结尾的最长递增序列长度=ends 有效区长度+1，所以令 dp[2]=2。ends 整个有效区都没有比 arr[2]更大的数，说明发现了比 ends 有效区长度更长的递增序列，于是把有效区扩大，ends 有效区=ends[0..1]=[1,5]。

3．遍历到 arr[3]==3。ends 有效区=ends[0..1]=[1,5]，在有效区中用二分法找到最左边大于或等于 arr[3]的数。发现是 ends[1]，表示以 arr[3]结尾的最长递增序列长度为 2，所以

令 dp[3]=2。然后令 ends[1]=3，因为遍历到目前为止，在所有长度为 2 的递增序列中，最小的结尾数是 3，而不再是 5。

4．遍历到 arr[4]==6。ends 有效区=ends[0..1]=[1,3]，在有效区中用二分法找到最左边大于或等于 arr[4]的数。发现没有这样的数，表示以 arr[4]结尾的最长递增序列长度=ends 有效区长度+1，所以令 dp[4]=3。ends 整个有效区都没有比 arr[4]更大的数，说明发现了比 ends 有效区长度更长的递增序列，于是把有效区扩大，ends 有效区=ends[0..2]=[1,3,6]。

5．遍历到 arr[5]==4。ends 有效区=ends[0..2]=[1,3,6]，在有效区中用二分法找到最左边大于或等于 arr[5]的数。发现是 ends[2]，表示以 arr[5]结尾的最长递增序列长度为 3，所以令 dp[5]=3。然后令 ends[2]=4，表示在所有长度为 3 的递增序列中，最小的结尾数变为 4。

6．遍历到 arr[6]==8。ends 有效区=ends[0..2]=[1,3,4]，在有效区中用二分法找到最左边大于或等于 arr[6]的数。发现没有这样的数，表示以 arr[6]结尾的最长递增序列长度=ends 有效区长度+1，所以令 dp[6]=4。ends 整个有效区都没有比 arr[6]更大的数，说明发现了比 ends 有效区长度更长的递增序列，于是把有效区扩大，ends 有效区=ends[0..3]=[1,3,4,8]。

7．遍历到 arr[7]==9。ends 有效区=ends[0..3]=[1,3,4,8]，在有效区中用二分法找到最左边大于或等于 arr[7]的数。发现没有这样的数，表示以 arr[7]结尾的最长递增序列长度=ends 有效区长度+1，所以令 dp[7]=5。ends 整个有效区都没有比 arr[7]更大的数，于是把有效区扩大，ends 有效区=ends[0..5]=[1,3,4,8,9]。

8．遍历到 arr[8]==7。ends 有效区=ends[0..5]=[1,3,4,8,9]，在有效区中用二分法找到最左边大于或等于 arr[8]的数。发现是 ends[3]，表示以 arr[8]结尾的最长递增序列长度为 4，所以令 dp[8]=4。然后令 ends[3]=7，表示在所有长度为 4 的递增序列中，最小的结尾数变为 7。

具体过程请参看如下代码中的 getdp2 方法。

```
public int[] getdp2(int[] arr) {
        int[] dp = new int[arr.length];
        int[] ends = new int[arr.length];
        ends[0] = arr[0];
        dp[0] = 1;
        int right = 0;
        int l = 0;
        int r = 0;
        int m = 0;
        for (int i = 1; i < arr.length; i++) {
                l = 0;
                r = right;
                while (l <= r) {
                        m = (l + r) / 2;
```

```
                        if (arr[i] > ends[m]) {
                                l = m + 1;
                        } else {
                                r = m - 1;
                        }
                }
                right = Math.max(right, l);
                ends[l] = arr[i];
                dp[i] = l + 1;
        }
        return dp;
}
```

时间复杂度 $O(N\log N)$方法的整个过程请参看如下代码中的 lis2 方法。

```
public int[] lis2(int[] arr) {
        if (arr == null || arr.length == 0) {
                return null;
        }
        int[] dp = getdp2(arr);
        return generateLIS(arr, dp);
}
```

汉诺塔问题

【题目】

给定一个整数 n，代表汉诺塔游戏中从小到大放置的 n 个圆盘，假设开始时所有的圆盘都放在左边的柱子上，想按照汉诺塔游戏的要求把所有的圆盘都移到右边的柱子上。实现函数打印最优移动轨迹。

【举例】

n=1 时，打印：

move from left to right

n=2 时，打印：

move from left to mid

move from left to right

move from mid to right

【进阶题目】

给定一个整型数组 arr，其中只含有 1、2 和 3，代表所有圆盘目前的状态，1 代表左柱，2 代表中柱，3 代表右柱，arr[i]的值代表第 *i*+1 个圆盘的位置。比如，arr=[3,3,2,1]，代表第 1 个圆盘在右柱上、第 2 个圆盘在右柱上、第 3 个圆盘在中柱上、第 4 个圆盘在左柱上。如果 arr 代表的状态是最优移动轨迹过程中出现的状态，返回 arr 这种状态是最优移动轨迹中的第几个状态。如果 arr 代表的状态不是最优移动轨迹过程中出现的状态，则返回-1。

【举例】

arr=[1,1]。两个圆盘目前都在左柱上，也就是初始状态，所以返回 0。

arr=[2,1]。第一个圆盘在中柱上、第二个圆盘在左柱上，这个状态是 2 个圆盘的汉诺塔游戏中最优移动轨迹的第 1 步，所以返回 1。

arr=[3,3]。第一个圆盘在右柱上、第二个圆盘在右柱上，这个状态是 2 个圆盘的汉诺塔游戏中最优移动轨迹的第 3 步，所以返回 3。

arr=[2,2]。第一个圆盘在中柱上、第二个圆盘在中柱上，这个状态是 2 个圆盘的汉诺塔游戏中最优移动轨迹从来不会出现的状态，所以返回-1。

【进阶题目要求】

如果 arr 长度为 *N*，请实现时间复杂度为 *O*(*N*)、额外空间复杂度为 *O*(1)的方法。

【难度】

校 ★★★☆

【解答】

原问题。假设有 from 柱子、mid 柱子和 to 柱子，都在 from 的圆盘 *1*~i 完全移动到 to，最优过程为：

步骤 1 为圆盘 *1*~*i*-1 从 from 移动到 mid。

步骤 2 为单独把圆盘 *i* 从 from 移动到 to。

步骤 3 为把圆盘 *1*~*i*-1 从 mid 移动到 to。如果圆盘只有 1 个，直接把这个圆盘从 from 移动到 to 即可。

打印最优移动轨迹的方法参见如下代码中的 hanoi 方法。

```
public void hanoi(int n) {
        if (n > 0) {
```

```
                func(n, "left", "mid", "right");
        }
    }

    public void func(int n, String from, String mid, String to) {
        if (n == 1) {
                System.out.println("move from " + from + " to " + to);
        } else {
                func(n - 1, from, to, mid);
                func(1, from, mid, to);
                func(n - 1, mid, from, to);
        }
    }
```

进阶题目。首先求都在 from 柱子上的圆盘 1~i，如果都移动到 to 上的最少步骤数，假设为 $S(i)$。根据上面的步骤，$S(i)$=步骤 1 的步骤总数+1+步骤 3 的步骤总数=$S(i\text{-}1)$+1+$S(i\text{-}1)$，$S(1)$=1。所以 $S(i)$+1=2($S(i\text{-}1)$+1)，$S(1)$+1==2。根据等比数列求和公式得到 $S(i)+1=2^i$，所以 $S(i)=2^{i-1}$。

对于数组 arr 来说，arr[N-1]表示最大圆盘 N 在哪个柱子上，情况有以下三种：

- 圆盘 N 在左柱上，说明步骤 1 或者没有完成，或者已经完成，需要考查圆盘 1~N-1 的状况。
- 圆盘 N 在右柱上，说明步骤 1 已经完成，起码走完了 2^{N-1} 步。步骤 2 也已经完成，起码又走完了 1 步，所以当前状况起码是最优步骤的 2^{N-1} 步，剩下的步骤怎么确定还得继续考查圆盘 1~N-1 的状况。
- 圆盘 N 在中柱上，这是不可能的，最优步骤中不可能让圆盘 N 处在中柱上，直接返回-1。

所以整个过程可以总结为：对圆盘 1~i 来说，如果目标为从 from 到 to，那么情况有三种：

- 圆盘 i 在 from 上，需要继续考查圆盘 1~i-1 的状况，圆盘 1~i-1 的目标为从 from 到 mid。
- 圆盘 i 在 to 上，说明起码走完了 2^{i-1} 步，剩下的步骤怎么确定还得继续考查圆盘 1~i-1 的状况，圆盘 1~i-1 的目标为从 mid 到 to。
- 圆盘 i 在 mid 上，直接返回-1。

整个过程参看如下代码中的 step1 方法。

```
public int step1(int[] arr) {
        if (arr == null || arr.length == 0) {
                return -1;
        }
```

```
        return process(arr, arr.length - 1, 1, 2, 3);
}

public int process(int[] arr, int i, int from, int mid, int to) {
        if (i == -1) {
                return 0;
        }
        if (arr[i] != from && arr[i] != to) {
                return -1;
        }
        if (arr[i] == from) {
                return process(arr, i - 1, from, to, mid);
        } else {
                int rest = process(arr, i - 1, mid, from, to);
                if (rest == -1) {
                        return -1;
                }
                return (1 << i) + rest;
        }
}
```

step1 方法是递归函数，递归最多调用 N 次，并且每步的递归函数再调用递归函数的次数最多一次。在每个递归过程中，除去递归调用的部分，剩下过程的时间复杂度为 $O(1)$，所以 step1 方法的时间复杂度为 $O(N)$。但是因为递归函数需要函数栈的关系，step1 方法的额外空间复杂度为 $O(N)$，所以为了达到题目的要求，需要将整个过程改成非递归的方法，具体请参看如下代码中的 step2 方法。

```
public int step2(int[] arr) {
        if (arr == null || arr.length == 0) {
                return -1;
        }
        int from = 1;
        int mid = 2;
        int to = 3;
        int i = arr.length - 1;
        int res = 0;
        int tmp = 0;
        while (i >= 0) {
                if (arr[i] != from && arr[i] != to) {
                        return -1;
                }
                if (arr[i] == to) {
                        res += 1 << i;
                        tmp = from;
                        from = mid;
                } else {
                        tmp = to;
                        to = mid;
                }
```

```
                mid = tmp;
                i--;
            }
            return res;
        }
```

最长公共子序列问题

【题目】

给定两个字符串 str1 和 str2，返回两个字符串的最长公共子序列。

【举例】

str1="1A2C3D4B56"，str2="B1D23CA45B6A"。

"123456"或者"12C4B6"都是最长公共子序列，返回哪一个都行。

【难度】

尉 ★★☆☆

【解答】

本题是非常经典的动态规划问题，先来介绍求解动态规划表的过程。如果 str1 的长度为 *M*，str2 的长度为 *N*，生成大小为 *M*×*N* 的矩阵 dp，行数为 *M*，列数为 *N*。dp[i][j]的含义是 str1[0..i]与 str2[0..j]的最长公共子序列的长度。从左到右，再从上到下计算矩阵 dp。

1．矩阵 dp 第一列即 dp[0..M-1][0]，dp[i][0]的含义是 str1[0..i]与 str2[0]的最长公共子序列长度。str2[0]只有一个字符，所以 dp[i][0]最大为 1。如果 str1[i]==str2[0]，令 dp[i][0]=1，一旦 dp[i][0]被设置为 1，之后的 dp[i+1..M-1][0]也都为 1。比如，str1[0..M-1]="ABCDE"，str2[0]="B"。str1[0]为"A"，与 str2[0]不相等，所以 dp[0][0]=0。str1[1]为"B"，与 str2[0]相等，所以 str1[0..1]与 str2[0]的最长公共子序列为"B"，令 dp[1][0]=1。之后的 dp[2..4][0]肯定都是 1，因为 str[0..2]、str[0..3]和 str[0..4]与 str2[0]的最长公共子序列肯定有"B"。

2. 矩阵 dp 第一行即 dp[0][0..N-1]与步骤 1 同理，如果 str1[0]==str2[j]，则令 dp[0][j]=1，一旦 dp[0][j]被设置为 1，之后的 dp[0][j+1..N-1]也都为 1。

3．对其他位置(i,j)，dp[i][j]的值只可能来自以下三种情况：

- 可能是 dp[i-1][j]，代表 str1[0..i-1]与 str2[0..j]的最长公共子序列长度。比如，

str1="A1BC2"，str2="AB34C"。str1[0..3]（即"A1BC"）与 str2[0..4]（即"AB34C"）的最长公共子序列为"ABC"，即 dp[3][4]为 3。str1[0..4]（即"A1BC2"）与 str2[0..4]（即"AB34C"）的最长公共子序列也是"ABC"，所以 dp[4][4]也为 3。

- 可能是 dp[i][j-1]，代表 str1[0..i]与 str2[0..j-1]的最长公共子序列长度。比如，str1="A1B2C"，str2="AB3C4"。str1[0..4]（即"A1B2C"）与 str2[0..3]（即"AB3C"）的最长公共子序列为"ABC"，即 dp[4][3]为 3。str1[0..4]（即"A1B2C"）与 str2[0..4]（即"AB3C4"）的最长公共子序列也是"ABC"，所以 dp[4][4]也为 3。
- 如果 str1[i]==str2[j]，还可能是 dp[i-1][j-1]+1。比如 str1="ABCD"，str2="ABCD"。str1[0..2]（即"ABC"）与 str2[0..2]（即"ABC"）的最长公共子序列为"ABC"，即 dp[2][2]为 3。因为 str1[3]==str2[3]=="D"，所以 str1[0..3]与 str2[0..3]的最长公共子序列是"ABCD"。

这三个可能的值中，选最大的作为 dp[i][j]的值。具体过程请参看如下代码中的 getdp 方法。

```
public int[][] getdp(char[] str1, char[] str2) {
        int[][] dp = new int[str1.length][str2.length];
        dp[0][0] = str1[0] == str2[0] ? 1 : 0;
        for (int i = 1; i < str1.length; i++) {
            dp[i][0] = Math.max(dp[i - 1][0], str1[i] == str2[0] ? 1 : 0);
        }
        for (int j = 1; j < str2.length; j++) {
            dp[0][j] = Math.max(dp[0][j - 1], str1[0] == str2[j] ? 1 : 0);
        }
        for (int i = 1; i < str1.length; i++) {
            for (int j = 1; j < str2.length; j++) {
                dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]);
                if (str1[i] == str2[j]) {
                    dp[i][j] = Math.max(dp[i][j], dp[i - 1][j - 1] + 1);
                }
            }
        }
        return dp;
}
```

dp 矩阵中最右下角的值代表 str1 整体和 str2 整体的最长公共子序列的长度。通过整个 dp 矩阵的状态，可以得到最长公共子序列。具体方法如下：

1．从矩阵的右下角开始，有三种移动方式：向上、向左、向左上。假设移动的过程中，*i* 表示此时的行数，*j* 表示此时的列数，同时用一个变量 res 来表示最长公共子序列。

2．如果 dp[i][j]大于 dp[i-1][j]和 dp[i][j-1]，说明之前在计算 dp[i][j]的时候，一定是选

择了决策 dp[i-1][j-1]+1，可以确定 str1[i]等于 str2[j]，并且这个字符一定属于最长公共子序列，把这个字符放进 res，然后向左上方移动。

3．如果 dp[i][j]等于 dp[i-1][j]，说明之前在计算 dp[i][j]的时候，dp[i-1][j-1]+1 这个决策不是必须选择的决策，向上方移动即可。

4．如果 dp[i][j]等于 dp[i][j-1]，与步骤 3 同理，向左方移动。

5．如果 dp[i][j]同时等于 dp[i-1][j]和 dp[i][j-1]，向上还是向下无所谓，选择其中一个即可，反正不会错过必须选择的字符。

也就是说，通过 dp 求解最长公共子序列的过程就是还原出当时如何求解 dp 的过程，来自哪个策略就朝哪个方向移动。全部过程请参看如下代码中的 lcse 方法。

```
public String lcse(String str1, String str2) {
    if (str1 == null || str2 == null || str1.equals("") || str2.equals("")) {
        return "";
    }
    char[] chs1 = str1.toCharArray();
    char[] chs2 = str2.toCharArray();
    int[][] dp = getdp(chs1, chs2);
    int m = chs1.length - 1;
    int n = chs2.length - 1;
    char[] res = new char[dp[m][n]];
    int index = res.length - 1;
    while (index >= 0) {
        if (n > 0 && dp[m][n] == dp[m][n - 1]) {
            n--;
        } else if (m > 0 && dp[m][n] == dp[m - 1][n]) {
            m--;
        } else {
            res[index--] = chs1[m];
            m--;
            n--;
        }
    }
    return String.valueOf(res);
}
```

计算 dp 矩阵中的某个位置就是简单比较相关的 3 个位置的值而已，所以时间复杂度为 $O(1)$，动态规划表 dp 的大小为 $M \times N$，所以计算 dp 矩阵的时间复杂度为 $O(M \times N)$。通过 dp 得到最长公共子序列的过程为 $O(M+N)$，因为向左最多移动 N 个位置，向上最多移动 M 个位置，所以总的时间复杂度为 $O(M \times N)$，额外空间复杂度为 $O(M \times N)$。如果题目不要求返回最长公共子序列，只想求最长公共子序列的长度，那么可以用空间压缩的方法将额外空间复杂度减小为 $O(\min\{M,N\})$，有兴趣的读者请阅读本书“矩阵的最小路径和”问题，这里

不再详述。

最长公共子串问题

【题目】

给定两个字符串 str1 和 str2，返回两个字符串的最长公共子串。

【举例】

str1="1AB2345CD"，str2="12345EF"，返回"2345"。

【要求】

如果 str1 长度为 M，str2 长度为 N，实现时间复杂度为 $O(M\times N)$，额外空间复杂度为 $O(1)$的方法。

【难度】

校 ★★★☆

【解答】

经典动态规划的方法可以做到时间复杂度为 $O(M\times N)$，额外空间复杂度为 $O(M\times N)$，经过优化之后的实现可以把额外空间复杂度从 $O(M\times N)$降至 $O(1)$，我们先来介绍经典方法。

首先需要生成动态规划表。生成大小为 $M\times N$ 的矩阵 dp，行数为 M，列数为 N。dp[i][j]的含义是，在必须把 str1[i]和 str2[j]当作公共子串最后一个字符的情况下，公共子串最长能有多长。比如，str1="A1234B"，str2="CD1234"，dp[3][4]的含义是在必须把 str1[3]（即'3'）和 str2[4]（即'3'）当作公共子串最后一个字符的情况下，公共子串最长能有多长。这种情况下的最长公共子串为"123"，所以 dp[3][4]为 3。再如，str1="A12E4B"，str2="CD12F4"，dp[3][4]的含义是在必须把 str1[3]（即'E'）和 str2[4]（即'F'）当作公共子串最后一个字符的情况下，公共子串最长能有多长。这种情况下根本不能构成公共子串，所以 dp[3][4]为 0。介绍了 dp[i][j]的意义后，接下来介绍 dp[i][j]怎么求。具体过程如下：

1．矩阵 dp 第一列即 dp[0..M-1][0]。对某一个位置(i,0)来说，如果 str1[i]==str2[0]，令 dp[i][0]=1，否则令 dp[i][0]=0。比如 str1="ABAC"，str2[0]="A"。dp 矩阵第一列上的值依次

为 dp[0][0]=1，dp[1][0]=0，dp[2][0]=1，dp[3][0]=0。

2．矩阵 dp 第一行即 dp[0][0..N-1]与步骤 1 同理。对某一个位置(0,*j*)来说，如果 str1[0]==str2[j]，令 dp[0][j]=1，否则令 dp[0][j]=0。

3．其他位置按照从左到右，再从上到下来计算，dp[i][j]的值只可能有两种情况。

- 如果 str1[i]!=str2[j]，说明在必须把 str1[i]和 str2[j]当作公共子串最后一个字符是不可能的，令 dp[i][j]=0。
- 如果 str1[i]==str2[j]，说明 str1[i]和 str2[j]可以作为公共子串的最后一个字符，从最后一个字符向左能扩多大的长度呢？就是 dp[i-1][j-1]的值，所以令 dp[i][j]=dp[i-1][j-1]+1。

如果 str1="abcde"，str2="bebcd"。计算的 dp 矩阵如下：

	b	e	b	c	d
a	0	0	0	0	0
b	1	0	1	0	0
c	0	0	0	2	0
d	0	0	0	0	3
e	0	1	0	0	0

计算 dp 矩阵的具体过程请参看如下代码中的 getdp 方法。

```
public int[][] getdp(char[] str1, char[] str2) {
        int[][] dp = new int[str1.length][str2.length];
        for (int i = 0; i < str1.length; i++) {
                if (str1[i] == str2[0]) {
                        dp[i][0] = 1;
                }
        }
        for (int j = 1; j < str2.length; j++) {
                if (str1[0] == str2[j]) {
                        dp[0][j] = 1;
                }
        }
        for (int i = 1; i < str1.length; i++) {
                for (int j = 1; j < str2.length; j++) {
                        if (str1[i] == str2[j]) {
                                dp[i][j] = dp[i - 1][j - 1] + 1;
                        }
                }
        }
        return dp;
}
```

生成动态规划表 dp 之后，得到最长公共子串是非常容易的。比如，上边生成的 dp 中，最大值是 dp[3][4]==3，说明最长公共子串的长度为 3。最长公共子串的最后一个字符是 str1[3]，当然也是 str2[4]，因为两个字符一样。那么最长公共子串为从 str1[3]开始向左一共 3 字节的子串，即 str1[1..3]，当然也是 str2[2..4]。总之，遍历 dp 找到最大值及其位置，最长公共子串自然可以得到。具体过程请参看如下代码中的 lcst1 方法，也是整个过程的主方法。

```
public String lcst1(String str1, String str2) {
    if (str1 == null || str2 == null || str1.equals("") || str2.equals("")) {
        return "";
    }
    char[] chs1 = str1.toCharArray();
    char[] chs2 = str2.toCharArray();
    int[][] dp = getdp(chs1, chs2);
    int end = 0;
    int max = 0;
    for (int i = 0; i < chs1.length; i++) {
        for (int j = 0; j < chs2.length; j++) {
            if (dp[i][j] > max) {
                end = i;
                max = dp[i][j];
            }
        }
    }
    return str1.substring(end - max + 1, end + 1);
}
```

经典动态规划的方法需要大小为 $M \times N$ 的 dp 矩阵，但实际上是可以减小至 $O(1)$的，因为我们注意到计算每一个 dp[i][j]的时候，最多只需要其左上方 dp[i-1][j-1]的值，所以按照斜线方向来计算所有的值，只需要一个变量就可以计算出所有位置的值，如图 4-1 所示。

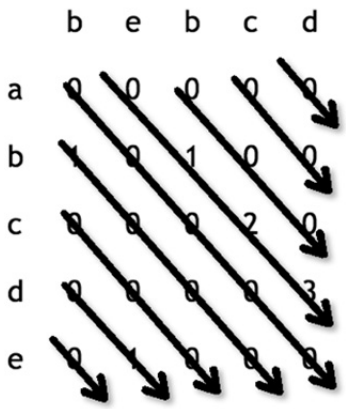

图 4-1

每一条斜线在计算之前生成整型变量 len，len 表示左上方位置的值，初始时 len=0。从斜线最左上的位置开始向右下方依次计算每个位置的值，假设计算到位置(i,j)，此时 len 表示位置(i-1,j-1)的值。如果 str1[i]==str2[j]，那么位置(i,j)的值为 len+1，如果 str1[i]!=str2[j]，那么位置(i,j)的值为 0。计算后将 len 更新成位置(i,j)的值，然后计算下一个位置，即(i+1,j+1)位置的值。依次计算下去就可以得到斜线上每个位置的值，然后算下一条斜线。用全局变量 max 记录所有位置的值中的最大值。最大值出现时，用全局变量 end 记录其位置即可。具体过程请参看如下代码中的 lcst2 方法。

```
public String lcst2(String str1, String str2) {
    if (str1 == null || str2 == null || str1.equals("") || str2.equals("")) {
        return "";
    }
    char[] chs1 = str1.toCharArray();
    char[] chs2 = str2.toCharArray();
    int row = 0; // 斜线开始位置的行
    int col = chs2.length - 1; // 斜线开始位置的列
    int max = 0; // 记录最大长度
    int end = 0; // 最大长度更新时，记录子串的结尾位置
    while (row < chs1.length) {
        int i = row;
        int j = col;
        int len = 0;
        // 从(i,j)开始向右下方遍历
        while (i < chs1.length && j < chs2.length) {
                if (chs1[i] != chs2[j]) {
                        len = 0;
                } else {
                        len++;
                }
                // 记录最大值，以及结束字符的位置
                if (len > max) {
                        end = i;
                        max = len;
                }
                i++;
                j++;
        }
        if (col > 0) { // 斜线开始位置的列先向左移动
                col--;
        } else { // 列移动到最左之后，行向下移动
                row++;
        }
    }
    return str1.substring(end - max + 1, end + 1);
}
```

最小编辑代价

【题目】

给定两个字符串 str1 和 str2，再给定三个整数 ic、dc 和 rc，分别代表插入、删除和替换一个字符的代价，返回将 str1 编辑成 str2 的最小代价。

【举例】

str1="abc"，str2="adc"，ic=5，dc=3，rc=2。
从"abc"编辑成"adc"，把'b'替换成'd'是代价最小的，所以返回 2。
str1="abc"，str2="adc"，ic=5，dc=3，rc=100。
从"abc"编辑成"adc"，先删除'b'，然后插入'd'是代价最小的，所以返回 8。
str1="abc"，str2="abc"，ic=5，dc=3，rc=2。
不用编辑了，本来就是一样的字符串，所以返回 0。

【难度】

校 ★★★☆

【解答】

如果 str1 的长度为 M，str2 的长度为 N，经典动态规划的方法可以达到时间复杂度为 $O(M\times N)$，额外空间复杂度为 $O(M\times N)$。如果结合空间压缩的技巧，可以把额外空间复杂度减至 $O(\min\{M,N\})$。

先来介绍经典动态规划的方法。首先生成大小为$(M+1)\times(N+1)$的矩阵 dp，dp[i][j]的值代表 str1[0..i-1]编辑成 str2[0..j-1]的最小代价。举个例子，str1="ab12cd3"，str2="abcdf"，ic=5，dc=3，rc=2。dp 是一个 8×6 的矩阵，最终计算结果如下。

	‘’	‘a’	‘b’	‘c’	‘d’	‘f’
‘’	0	5	10	15	20	25
‘a’	3	0	5	10	15	20
‘b’	6	3	0	5	10	15
‘1’	9	6	3	2	7	12

‘2’	12	9	6	5	4	9
‘c’	15	12	9	6	7	6
‘d’	18	15	12	9	6	9
‘3’	21	18	15	12	9	8

下面具体说明 dp 矩阵每个位置的值是如何计算的。

1．dp[0][0]=0，表示 str1 空的子串编辑成 str2 空的子串的代价为 0。

2．矩阵 dp 第一列即 dp[0..M-1][0]。dp[i][0]表示 str1[0..i-1]编辑成空串的最小代价，毫无疑问，是把 str1[0..i-1]所有的字符删掉的代价，所以 dp[i][0]=dc*i。

3．矩阵 dp 第一行即 dp[0][0..N-1]。dp[0][j]表示空串编辑成 str2[0..j-1]的最小代价，毫无疑问，是在空串里插入 str2[0..j-1]所有字符的代价，所以 dp[0][j]=ic*j。

4. 其他位置按照从左到右，再从上到下来计算，dp[i][j]的值只可能来自以下四种情况。

- str1[0..i-1]可以先编辑成 str1[0..i-2]，也就是删除字符 str1[i-1]，然后由 str1[0..i-2]编辑成 str2[0..j-1]，dp[i-1][j]表示 str1[0..i-2]编辑成 str2[0..j-1]的最小代价，那么 dp[i][j]可能等于 dc+dp[i-1][j]。
- str1[0..i-1]可以先编辑成 str2[0..j-2]，然后将 str2[0..j-2]插入字符 str2[j-1]，编辑成 str2[0..j-1]，dp[i][j-1]表示 str1[0..i-1]编辑成 str2[0..j-2]的最小代价，那么 dp[i][j]可能等于 dp[i][j-1]+ic。
- 如果 str1[i-1]!=str2[j-1]。先把 str1[0..i-1]中 str1[0..i-2]的部分变成 str2[0..j-2]，然后把字符 str1[i-1]替换成 str2[j-1]，这样 str1[0..i-1]就编辑成 str2[0..j-1]了。dp[i-1][j-1]表示 str1[0..i-2]编辑成 str2[0..i-2]的最小代价，那么 dp[i][j]可能等于 dp[i-1][j-1]+rc。
- 如果 str1[i-1]==str2[j-1]。先把 str1[0..i-1]中 str1[0..i-2]的部分变成 str2[0..j-2]，因为此时字符 str1[i-1]等于 str2[j-1]，所以 str1[0..i-1]已经编辑成 str2[0..j-1]了。dp[i-1][j-1]表示 str1[0..i-2]编辑成 str2[0..i-2]的最小代价，那么 dp[i][j]可能等于 dp[i-1][j-1]。

5．以上四种可能的值中，选最小值作为 dp[i][j]的值。dp 最右下角的值就是最终结果。具体过程请参看如下代码中的 minCost1 方法。

```
public int minCost1(String str1, String str2, int ic, int dc, int rc) {
        if (str1 == null || str2 == null) {
                return 0;
        }
        char[] chs1 = str1.toCharArray();
        char[] chs2 = str2.toCharArray();
        int row = chs1.length + 1;
```

```
        int col = chs2.length + 1;
        int[][] dp = new int[row][col];
        for (int i = 1; i < row; i++) {
                dp[i][0] = dc * i;
        }
        for (int j = 1; j < col; j++) {
                dp[0][j] = ic * j;
        }
        for (int i = 1; i < row; i++) {
                for (int j = 1; j < col; j++) {
                        if (chs1[i - 1] == chs2[j - 1]) {
                                dp[i][j] = dp[i - 1][j - 1];
                        } else {
                                dp[i][j] = dp[i - 1][j - 1] + rc;
                        }
                        dp[i][j] = Math.min(dp[i][j], dp[i][j - 1] + ic);
                        dp[i][j] = Math.min(dp[i][j], dp[i - 1][j] + dc);
                }
        }
        return dp[row - 1][col - 1];
}
```

经典动态规划方法结合空间压缩的方法。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。但是本题空间压缩的方法有一点特殊。在“矩阵的最小路径和”问题中，dp[i][j]依赖两个位置的值 dp[i-1][j]和 dp[i][j-1]，滚动数组从左到右更新是没有问题的，因为在求 dp[j]的时候，dp[j]没有更新之前相当于 dp[i-1][j]的值，dp[j-1]的值又已经更新过相当于 dp[i][j-1]的值。而本题 dp[i][j]依赖 dp[i-1][j]、dp[i][j-1]和 dp[i-1][j-1]的值，所以滚动数组从左到右更新时，还需要一个变量来保存 dp[j-1]没更新之前的值，也就是左上角的 dp[i-1][j-1]。

理解了上述过程后，就不难发现该过程确实只用了一个 dp 数组，但 dp 长度等于 str2 的长度加 1（即 N+1），而不是 $O(\text{Min}\{M,N\})$。所以还要把 str1 和 str2 中长度较短的一个作为列对应的字符串，长度较长的作为行对应的字符串。上面介绍的动态规划方法都是把 str2 作为列对应的字符串，如果 str1 做了列对应的字符串，把插入代价 ic 和删除代价 dc 交换一下即可。

具体过程请参看如下代码中的 minCost2 方法。

```
public int minCost2(String str1, String str2, int ic, int dc, int rc) {
        if (str1 == null || str2 == null) {
                return 0;
        }
        char[] chs1 = str1.toCharArray();
        char[] chs2 = str2.toCharArray();
        char[] longs = chs1.length >= chs2.length ? chs1 : chs2;
```

```
        char[] shorts = chs1.length < chs2.length ? chs1 : chs2;
        if (chs1.length < chs2.length) { // str2 较长就交换 ic 和 dc 的值
                int tmp = ic;
                ic = dc;
                dc = tmp;
        }
        int[] dp = new int[shorts.length + 1];
        for (int i = 1; i <= shorts.length; i++) {
                dp[i] = ic * i;
        }
        for (int i = 1; i <= longs.length; i++) {
                int pre = dp[0]; // pre 表示左上角的值
                dp[0] = dc * i;
                for (int j = 1; j <= shorts.length; j++) {
                        int tmp = dp[j]; // dp[j]没更新前先保存下来
                        if (longs[i - 1] == shorts[j - 1]) {
                                dp[j] = pre;
                        } else {
                                dp[j] = pre + rc;
                        }
                        dp[j] = Math.min(dp[j], dp[j - 1] + ic);
                        dp[j] = Math.min(dp[j], tmp + dc);
                        pre = tmp; // pre 变成 dp[j]没更新前的值
                }
        }
        return dp[shorts.length];
}
```

字符串的交错组成

【题目】

给定三个字符串 str1、str2 和 aim，如果 aim 包含且仅包含来自 str1 和 str2 的所有字符，而且在 aim 中属于 str1 的字符之间保持原来在 str1 中的顺序，属于 str2 的字符之间保持原来在 str2 中的顺序，那么称 aim 是 str1 和 str2 的交错组成。实现一个函数，判断 aim 是否是 str1 和 str2 交错组成。

【举例】

str1="AB"，str2="12"。那么"AB12"、"A1B2"、"A12B"、"1A2B"和"1AB2"等都是 str1 和 str2 的交错组成。

【难度】

校 ★★★☆

【解答】

如果 str1 的长度为 M，str2 的长度为 N，经典动态规划的方法可以达到时间复杂度为 $O(M\times N)$，额外空间复杂度为 $O(M\times N)$。如果结合空间压缩的技巧，可以把额外空间复杂度减至 $O(\min\{M,N\})$。

先来介绍经典动态规划的方法。首先 aim 如果是 str1 和 str2 的交错组成，aim 的长度一定是 $M+N$，否则直接返回 false。然后生成大小为$(M+1)\times(N+1)$布尔类型的矩阵 dp，dp[i][j]的值代表 aim[0..i+j-1]能否被 str1[0..i-1]和 str2[0..j-1]交错组成。计算 dp 矩阵的时候，是从左到右，再从上到下计算的，dp[M][N]也就是 dp 矩阵中最右下角的值，表示 aim 整体能否被 str1 整体和 str2 整体交错组成，也就是最终结果。下面具体说明 dp 矩阵每个位置的值是如何计算的。

1．dp[0][0]=true。aim 为空串时，当然可以被 str1 为空串和 str2 为空串交错组成。

2．矩阵 dp 第一列即 dp[0..M-1][0]。dp[i][0]表示 aim[0..i-1]能否只被 str1[0..i-1]交错组成。如果 aim[0..i-1]等于 str1[0..i-1]，则令 dp[i][0]=true，否则令 dp[i][0]=false。

3．矩阵 dp 第一行即 dp[0][0..N-1]。dp[0][j]表示 aim[0..j-1]能否只被 str2[0..j-1]交错组成。如果 aim[0..j-1]等于 str1[0..j-1]，则令 dp[i][0]=true，否则令 dp[i][0]=false。

4．对其他位置(i,j)，dp[i][j]的值由下面的情况决定。

- dp[i-1][j]代表 aim[0..i+j-2]能否被 str1[0..i-2]和 str2[0..j-1]交错组成，如果可以，那么如果再有 str1[i-1]等于 aim[i+j-1],说明 str1[i-1]又可以作为交错组成 aim[0..i+j-1]的最后一个字符。令 dp[i][j]=true。
- dp[i][j-1]代表 aim[0..i+j-2]能否被 str1[0..i-1]和 str2[0..j-2]交错组成，如果可以，那么如果再有 str2[j-1]等于 aim[i+j-1],说明 str1[j-1]又可以作为交错组成 aim[0..i+j-1]的最后一个字符。令 dp[i][j]=true。
- 如果第 1 种情况和第 2 种情况都不满足，令 dp[i][j]=false。

具体过程请参看如下代码中的 isCross1 方法。

```
public boolean isCross1(String str1, String str2, String aim) {
        if (str1 == null || str2 == null || aim == null) {
                return false;
        }
        char[] ch1 = str1.toCharArray();
```

```
        char[] ch2 = str2.toCharArray();
        char[] chaim = aim.toCharArray();
        if (chaim.length != ch1.length + ch2.length) {
                return false;
        }
        boolean[][] dp = new boolean[ch1.length + 1][ch2.length + 1];
        dp[0][0] = true;
        for (int i = 1; i <= ch1.length; i++) {
                if (ch1[i - 1] != chaim[i - 1]) {
                        break;
                }
                dp[i][0] = true;
        }
        for (int j = 1; j <= ch2.length; j++) {
                if (ch2[j - 1] != chaim[j - 1]) {
                        break;
                }
                dp[0][j] = true;
        }
        for (int i = 1; i <= ch1.length; i++) {
                for (int j = 1; j <= ch2.length; j++) {
                    if ((ch1[i - 1] == chaim[i + j - 1] && dp[i - 1][j])
                        || (ch2[j - 1] == chaim[i + j - 1] && dp[i][j - 1])) {
                            dp[i][j] = true;
                    }
                }
        }
        return dp[ch1.length][ch2.length];
}
```

经典动态规划方法结合空间压缩的方法。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。实际进行空间压缩的时候，比较 str1 和 str2 中哪个长度较小，长度较小的那个作为列对应的字符串，然后生成和较短字符串长度一样的一维数组 dp，滚动更新即可。

具体请参看如下代码中的 isCross2 方法。

```
public boolean isCross2(String str1, String str2, String aim) {
        if (str1 == null || str2 == null || aim == null) {
                return false;
        }
        char[] ch1 = str1.toCharArray();
        char[] ch2 = str2.toCharArray();
        char[] chaim = aim.toCharArray();
        if (chaim.length != ch1.length + ch2.length) {
                return false;
        }
        char[] longs = ch1.length >= ch2.length ? ch1 : ch2;
        char[] shorts = ch1.length < ch2.length ? ch1 : ch2;
        boolean[] dp = new boolean[shorts.length + 1];
```

```
        dp[0] = true;
        for (int i = 1; i <= shorts.length; i++) {
            if (shorts[i - 1] != chaim[i - 1]) {
                break;
            }
            dp[i] = true;
        }
        for (int i = 1; i <= longs.length; i++) {
            dp[0] = dp[0] && longs[i - 1] == chaim[i - 1];
            for (int j = 1; j <= shorts.length; j++) {
                if ((longs[i - 1] == chaim[i + j - 1] && dp[j])
                    || (shorts[j - 1] == chaim[i + j - 1] && dp[j - 1])) {
                    dp[j] = true;
                } else {
                    dp[j] = false;
                }
            }
        }
        return dp[shorts.length];
    }
```

龙与地下城游戏问题

【题目】

给定一个二维数组 map，含义是一张地图，例如，如下矩阵：

-2　-3　3

-5　-10　1

0　30　-5

游戏的规则如下：

- 骑士从左上角出发，每次只能向右或向下走，最后到达右下角见到公主。
- 地图中每个位置的值代表骑士要遭遇的事情。如果是负数，说明此处有怪兽，要让骑士损失血量。如果是非负数，代表此处有血瓶，能让骑士回血。
- 骑士从左上角到右下角的过程中，走到任何一个位置时，血量都不能少于 1。

为了保证骑士能见到公主，初始血量至少是多少？根据 map，返回初始血量。

【难度】

尉 ★★☆☆

【解答】

先介绍经典动态规划的方法，定义和地图大小一样的矩阵，记为 dp，dp[i][j]的含义是如果骑士要走上位置(*i*,*j*)，并且从该位置选一条最优的路径，最后走到右下角，骑士起码应该具备的血量。根据 dp 的定义，我们最终需要的是 dp[0][0]的结果。以题目的例子来说，map[2][2]的值为-5，所以骑士若要走上这个位置，需要 6 点血才能让自己不死。同时位置(2,2)已经是最右下角的位置，即没有后续的路径，所以 dp[2][2]==6。

那么 dp[i][j]的值应该怎么计算呢？

骑士还要面临向下还是向右的选择，dp[i][j+1]是骑士选择当前向右走并最终达到右下角的血量要求。同理，dp[i+1][j]是向下走的要求。如果骑士决定向右走，那么骑士在当前位置加完血或者扣完血之后的血量只要等于 dp[i][j+1]即可。那么骑士在加血或扣血之前的血量要求（也就是在没有踏上(*i*,*j*)位置之前的血量要求），就是 dp[i][j+1]-map[i][j]。同时，骑士血量要随时不少于 1，所以向右的要求为 max{dp[i][j+1]-map[i][j],1}。如果骑士决定向下走，分析方式相同，向下的要求为 max{dp[i+1][j]-map[i][j],1}。

骑士可以有两种选择，当然要选最优的一条，所以 dp[i][j]=Min{向右的要求,向下的要求}。计算 dp 矩阵时从右下角开始计算，选择依次从右至左、再从下到上的计算方式即可。

具体请参看如下代码中的 minHP1 方法。

```
public int minHP1(int[][] m) {
        if (m == null || m.length == 0 || m[0] == null || m[0].length == 0) {
                return 1;
        }
        int row = m.length;
        int col = m[0].length;
        int[][] dp = new int[row--][col--];
        dp[row][col] = m[row][col] > 0 ? 1 : -m[row][col] + 1;
        for (int j = col - 1; j >= 0; j--) {
                dp[row][j] = Math.max(dp[row][j + 1] - m[row][j], 1);
        }
        int right = 0;
        int down = 0;
        for (int i = row - 1; i >= 0; i--) {
                dp[i][col] = Math.max(dp[i + 1][col] - m[i][col], 1);
                for (int j = col - 1; j >= 0; j--) {
                        right = Math.max(dp[i][j + 1] - m[i][j], 1);
                        down = Math.max(dp[i + 1][j] - m[i][j], 1);
                        dp[i][j] = Math.min(right, down);
                }
        }
        return dp[0][0];
}
```

如果 map 大小为 $M \times N$，经典动态规划方法的时间复杂度为 $O(M \times N)$，额外空间复杂度为 $O(M \times N)$。结合空间压缩之后可以将额外空间复杂度降至 $O(\min\{M,N\})$。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 minHP2 方法。

```
public static int minHP2(int[][] m) {
        if (m == null || m.length == 0 || m[0] == null || m[0].length == 0) {
                return 1;
        }
        int more = Math.max(m.length, m[0].length);
        int less = Math.min(m.length, m[0].length);
        boolean rowmore = more == m.length;
        int[] dp = new int[less];
        int tmp = m[m.length - 1][m[0].length - 1];
        dp[less - 1] = tmp > 0 ? 1 : -tmp + 1;
        int row = 0;
        int col = 0;
        for (int j = less - 2; j >= 0; j--) {
                row = rowmore ? more - 1 : j;
                col = rowmore ? j : more - 1;
                dp[j] = Math.max(dp[j + 1] - m[row][col], 1);
        }
        int choosen1 = 0;
        int choosen2 = 0;
        for (int i = more - 2; i >= 0; i--) {
                row = rowmore ? i : less - 1;
                col = rowmore ? less - 1 : i;
                dp[less - 1] = Math.max(dp[less - 1] - m[row][col], 1);
                for (int j = less - 2; j >= 0; j--) {
                        row = rowmore ? i : j;
                        col = rowmore ? j : i;
                        choosen1 = Math.max(dp[j] - m[row][col], 1);
                        choosen2 = Math.max(dp[j + 1] - m[row][col], 1);
                        dp[j] = Math.min(choosen1, choosen2);
                }
        }
        return dp[0];
}
```

数字字符串转换为字母组合的种数

【题目】

给定一个字符串 str，str 全部由数字字符组成，如果 str 中某一个或某相邻两个字符组成的子串值在 1~26 之间，则这个子串可以转换为一个字母。规定"1"转换为"A"，"2"转换

为"B"，"3"转换为"C"..."26"转换为"Z"。写一个函数，求 str 有多少种不同的转换结果，并返回种数。

【举例】

str="1111"。

能转换出的结果有"AAAA"、"LAA"、"ALA"、"AAL"和"LL"，返回 5。

str="01"。

"0"没有对应的字母，而"01"根据规定不可转换，返回 0。

str="10"。

能转换出的结果是"J"，返回 1。

【难度】

尉 ★★☆☆

【解答】

暴力递归的方法。假设 str 的长度为 N，先定义递归函数 $p(i)(0 \leqslant i \leqslant N)$。$p(i)$的含义是 str[0..i-1]已经转换完毕，而 str[i..N-1]还没转换的情况下，最终合法的转换种数有多少并返回。特别指出，$p(N)$表示 str[0..N-1]（也就是 str 的整体）都已经转换完，没有后续的字符了，那么合法的转换种数为 1，即 $p(N)=1$。比如，str="111123"，p(4)表示 str[0..3]（即"1111"）已经转换完毕，具体结果是什么不重要，反正已经转换完毕并且不可变，没转换的部分是 str[4..5]（即"23"），可转换的为"BC"或"W"只有两种，所以 $p(4)=2$。$p(6)$表示 str 整体已经转换完毕，所以 $p(6)=1$。那么 $p(i)$如何计算呢？只有以下 4 种情况。

- 如果 i==N。根据上文对 $p(N)=1$ 的解释，直接返回 1。
- 如果不满足情况 1，又有 str[i]=='0'。str[0..i-1]已经转换完毕，而 str[i..N-1]此时又以'0'开头，str[i..N-1]无论怎样都不可能合法转换，所以直接返回 0。
- 如果不满足情况 1 和情况 2，说明 str[i]属于'1'~'9'，str[i]可以转换为'A'~'I'，那么 $p(i)$的值一定包含 $p(i+1)$的值，即 $p(i)=p(i+1)$。
- 如果不满足情况 1 和情况 2，说明 str[i]属于'1'~'9'，如果又有 str[i..i+1]在"10”"~"26"之间，str[i..i+1]可以转换为'J'~'Z'，那么 $p(i)$的值一定也包含 $p(i+2)$的值，即 $p(i)+=p(i+2)$。

具体过程请参看如下代码中的 num1 方法。

```
public int num1(String str) {
        if (str == null || str.equals("")) {
                return 0;
        }
        char[] chs = str.toCharArray();
        return process(chs, 0);
}

public int process(char[] chs, int i) {
        if (i == chs.length) {
                return 1;
        }
        if (chs[i] == '0') {
                return 0;
        }
        int res = process(chs, i + 1);
        if (i + 1 < chs.length && (chs[i] - '0') * 10 + chs[i + 1] - '0' < 27) {
                res += process(chs, i + 2);
        }
        return res;
}
```

以上过程中，$p(i)$最多可能会有两个递归分支 $p(i+1)$和 $p(i+2)$，一共有 N 层递归，所以时间复杂度为 $O(2^N)$，额外空间复杂度就是递归使用的函数栈的大小为 $O(N)$。但是研究一下递归函数 p 就会发现，$p(i)$最多依赖 $p(i+1)$和 $p(i+2)$的值，这是可以从后往前进行顺序计算的，也就是先计算 $p(N)$和 $p(N-1)$，然后根据这两个值计算 $p(N-2)$，再根据 $p(N-1)$和 $p(N-2)$计算 $p(N-3)$，最后根据 $p(1)$和 $p(2)$计算出 $p(0)$即可，类似斐波那契数列的求解过程，只不过斐波那契数列是从前往后计算的，这里是从后往前计算而已。具体过程请参看如下代码中的 num2 方法。

```
public int num2(String str) {
        if (str == null || str.equals("")) {
                return 0;
        }
        char[] chs = str.toCharArray();
        int cur = chs[chs.length - 1] == '0' ? 0 : 1;
        int next = 1;
        int tmp = 0;
        for (int i = chs.length - 2; i >= 0; i--) {
                if (chs[i] == '0') {
                        next = cur;
                        cur = 0;
                } else {
                        tmp = cur;
                        if ((chs[i] - '0') * 10 + chs[i + 1] - '0' < 27) {
                                cur += next;
                        }
```

```
                    next = tmp;
                }
            }
            return cur;
        }
```

因为是顺序计算，所以 num2 方法的时间复杂度为 $O(N)$，同时只用了 cur、next 和 tmp 进行滚动更新，所以额外空间复杂度为 $O(1)$。但是本题并不能像斐波那契数列问题那样用矩阵乘法的优化方法将时间复杂度优化到 $O(\log N)$，这是因为斐波那契数列是严格的 $f(i)=f(i-1)+f(i-2)$，但是本题并不严格，str[i]的具体情况决定了 $p(i)$是等于 0 还是等于 $p(i+1)$，还是等于 $p(i+1)+p(i+2)$。有状态转移的表达式不可以用矩阵乘法将时间复杂度优化到 $O(\log N)$。但如果 str 只由字符'1'和字符'2'组成，比如"12121121212122"，那么就可以使用矩阵乘法的方法将时间复杂度优化为 $O(\log N)$。因为 str[i]都可以单独转换成字母，str[i..i+1]也都可以一起转换成字母，此时一定有 $p(i)=p(i+1)+p(i+2)$。总之，可以使用矩阵乘法的前提是递归表达式不会发生转移。

表达式得到期望结果的组成种数

【题目】

给定一个只由 0（假）、1(真)、&(逻辑与)、|（逻辑或）和^(异或)五种字符组成的字符串 express，再给定一个布尔值 desired。返回 express 能有多少种组合方式，可以达到 desired 的结果。

【举例】

express="1^0|0|1"，desired=false。
只有 1^((0|0)|1)和 1^(0|(0|1))的组合可以得到 false，返回 2。
express="1"，desired=false。
无组合则可以得到 false，返回 0。

【难度】

校 ★★★☆

【解答】

应该首先判断 express 是否合乎题目要求，比如"1^"和"10"，都不是有效的表达式。总结起来有以下三个判断标准：

- 表达式的长度必须是奇数。
- 表达式下标为偶数位置的字符一定是'0'或者'1'。
- 表达式下标为奇数位置的字符一定是'&'或'|'或'^'。

只要符合上述三个标准，表达式必然是有效的。具体参看如下代码中的 isValid 方法。

```
public boolean isValid(char[] exp) {
        if ((exp.length & 1) == 0) {
                return false;
        }
        for (int i = 0; i < exp.length; i = i + 2) {
                if ((exp[i] != '1') && (exp[i] != '0')) {
                        return false;
                }
        }
        for (int i = 1; i < exp.length; i = i + 2) {
                if ((exp[i] != '&') && (exp[i] != '|') && (exp[i] != '^')) {
                        return false;
                }
        }
        return true;
}
```

暴力递归方法。在判断 express 符合标准之后，将 express 划分成左右两个部分，求出各种划分的情况下，能得到 desired 的种数是多少。以本题的例子进行举例说明，express 为"1^0|0|1"，desired 为 false，总的种数求法如下：

- 第 1 个划分为'^'，左部分为"1"，右部分为"0|0|1"，因为当前划分的逻辑符号为^，所以要想在此划分下得到 false，包含的可能性有两种：左部分为真，右部分为真；左部分为假，右部分为假。

结果 1 = 左部分为真的种数 × 右部分为真的种数 + 左部分为假的种数 × 右部分为假的种数。

- 第 2 个划分为'|'，左部分为"1^0"，右部分为"0|1"，因为当前划分的逻辑符号为|，所以要想在此划分下得到 false，包含的可能性只有一种，即左部分为假，右部分为假。

结果 2 = 左部分为假的种数 × 右部分为假的种数。

- 第 3 个划分为'|'，左部分为"1^0|0"，右部分为"1"，因为当前划分的逻辑符号为|，

所以结果 3 = 左部分为假的种数 × 右部分为假的种数。

- 结果 1+结果 2+结果 3 就是总的种数，也就是说，一个字符串中有几个逻辑符号，就有多少种划分，把每种划分能够得到最终 desired 值的种数全加起来，就是总的种数。

现在来系统地总结一下划分符号和 desired 的情况。

① 划分符号为^、desired 为 true 的情况下：

种数 = 左部分为真的种数 × 右部分为假的种数 + 左部分为假的种数 × 右部分为真的种数。

② 划分符号为^、desired 为 false 的情况下：

种数 = 左部分为真的种数 × 右部分为真的种数 + 左部分为假的种数 × 右部分为假的种数。

③ 划分符号为&、desired 为 true 的情况下：

种数 = 左部分为真的种数 × 右部分为真的种数。

④ 划分符号为&、desired 为 false 的情况下：

种数 = 左部分为真的种数 × 右部分为假的种数 + 左部分为假的种数 × 右部分为真的种数 + 左部分为假的种数 × 右部分为假的种数。

⑤ 划分符号为|、desired 为 true 的情况下：

种数 = 左部分为真的种数 × 右部分为假的种数 + 左部分为假的种数 × 右部分为真的种数 + 左部分为真的种数 × 右部分为真的种数。

⑥ 划分符号为|、desired 为 false 的情况下：

种数 = 左部分为假的种数 × 右部分为假的种数。

根据如上总结，以 express 中的每一个逻辑符号来划分 express，每种划分都求出各自的种数，再把种数累加起来，就是 express 达到 desired 总的种数。每次划分出的左右两部分递归求解即可。具体过程请参看如下代码中的 num1 方法。

```
public int num1(String express, boolean desired) {
    if (express == null || express.equals("")) {
        return 0;
    }
    char[] exp = express.toCharArray();
    if (!isValid(exp)) {
        return 0;
    }
    return p(exp, desired, 0, exp.length - 1);
}
```

```
public int p(char[] exp, boolean desired, int l, int r) {
    if (l == r) {
        if (exp[l] == '1') {
            return desired ? 1 : 0;
        } else {
            return desired ? 0 : 1;
        }
    }
    int res = 0;
    if (desired) {
        for (int i = l + 1; i < r; i += 2) {
            switch (exp[i]) {
                case '&':
                    res += p(exp, true, l, i - 1) * p(exp, true, i + 1, r);
                    break;
                case '|':
                    res += p(exp, true, l, i - 1) * p(exp, false, i + 1, r);
                    res += p(exp, false, l, i - 1) * p(exp, true, i + 1, r);
                    res += p(exp, true, l, i - 1) * p(exp, true, i + 1, r);
                    break;
                case '^':
                    res += p(exp, true, l, i - 1) * p(exp, false, i + 1, r);
                    res += p(exp, false, l, i - 1) * p(exp, true, i + 1, r);
                    break;
            }
        }
    } else {
        for (int i = l + 1; i < r; i += 2) {
            switch (exp[i]) {
                case '&':
                    res += p(exp, false, l, i - 1) * p(exp, true, i + 1, r);
                    res += p(exp, true, l, i - 1) * p(exp, false, i + 1, r);
                    res += p(exp, false, l, i - 1) * p(exp, false, i + 1, r);
                    break;
                case '|':
                    res += p(exp, false, l, i - 1) * p(exp, false, i + 1, r);
                    break;
                case '^':
                    res += p(exp, true, l, i - 1) * p(exp, true, i + 1, r);
                    res += p(exp, false, l, i - 1) * p(exp, false, i + 1, r);
                    break;
            }
        }
    }
    return res;
}
```

一个长度为 N 的 express，假设计算 express[i..j]的过程记为 $p(i,j)$，那么计算 $p(0,N\text{-}1)$需要计算 $p(0,0)$与 $p(1,N\text{-}1)$、$p(0,1)$与 $p(2,N\text{-}1)$...p(0,i)与 $p(i\text{+}1,N\text{-}1)$...$p(0,N\text{-}2)$与 $p(N\text{-}1,N\text{-}1)$，起码 $2N$ 种状态。对于每一组 $p(0,i)$与 $p(i\text{+}1,N\text{-}1)$来说，两者相加的划分种数又是 N-1 种，所以

起码要计算 2(N-1)种状态。所以用 num1 方法来计算一个长度为 N 的 express，总的时间复杂度为 $O(N!)$，额外空间复杂度为 $O(N)$，因为函数栈的大小为 N。之所以用暴力递归方法的时间复杂度这么高，是因为每一种状态计算过后没有保存下来，导致重复计算的大量发生。

动态规划的方法。如果 express 长度为 N，生成两个大小为 $N\times N$ 的矩阵 t 和 f，t[j][i]表示 express[j..i]组成 true 的种数，f[j][i]表示 express[j..i]组成 false 的种数。t[j][i]和 f[j][i]的计算方式还是枚举 express[j..i]上的每种划分。具体过程请参看如下代码中的 num2 方法。

```
public int num2(String express, boolean desired) {
    if (express == null || express.equals("")) {
    return 0;
    }
    char[] exp = express.toCharArray();
    if (!isValid(exp)) {
    return 0;
    }
    int[][] t = new int[exp.length][exp.length];
    int[][] f = new int[exp.length][exp.length];
    t[0][0] = exp[0] == '0' ? 0 : 1;
    f[0][0] = exp[0] == '1' ? 0 : 1;
    for (int i = 2; i < exp.length; i += 2) {
    t[i][i] = exp[i] == '0' ? 0 : 1;
    f[i][i] = exp[i] == '1' ? 0 : 1;
    for (int j = i - 2; j >= 0; j -= 2) {
        for (int k = j; k < i; k += 2) {
            if (exp[k + 1] == '&') {
                t[j][i]+=t[j][k] * t[k + 2][i];
                f[j][i]+=(f[j][k] + t[j][k]) * f[k + 2][i] + f[j][k] * t[k + 2][i];
            } else if (exp[k + 1] == '|') {
                t[j][i]+=(f[j][k] + t[j][k]) * t[k + 2][i] + t[j][k] * f[k + 2][i];
                f[j][i]+=f[j][k] * f[k + 2][i];
            } else {
                t[j][i]+=f[j][k] * t[k + 2][i] + t[j][k] * f[k + 2][i];
                f[j][i]+=f[j][k] * f[k + 2][i] + t[j][k] * t[k + 2][i];
            }
        }
    }
    }
    return desired ? t[0][t.length - 1] : f[0][f.length - 1];
}
```

矩阵 t 和 f 的大小为 $N\times N$，每个位置在计算的时候都有枚举的过程，所以动态规划方法的时间复杂度为 $O(N^3)$，额外空间复杂度为 $O(N^2)$。

排成一条线的纸牌博弈问题

【题目】

给定一个整型数组 arr，代表数值不同的纸牌排成一条线。玩家 A 和玩家 B 依次拿走每张纸牌，规定玩家 A 先拿，玩家 B 后拿，但是每个玩家每次只能拿走最左或最右的纸牌，玩家 A 和玩家 B 都绝顶聪明。请返回最后获胜者的分数。

【举例】

arr=[1,2,100,4]。

开始时玩家 A 只能拿走 1 或 4。如果玩家 A 拿走 1，则排列变为[2,100,4]，接下来玩家 B 可以拿走 2 或 4，然后继续轮到玩家 A。如果开始时玩家 A 拿走 4，则排列变为[1,2,100]，接下来玩家 B 可以拿走 1 或 100，然后继续轮到玩家 A。玩家 A 作为绝顶聪明的人不会先拿 4，因为拿 4 之后，玩家 B 将拿走 100。所以玩家 A 会先拿 1，让排列变为[2,100,4]，接下来玩家 B 不管怎么选，100 都会被玩家 A 拿走。玩家 A 会获胜，分数为 101。所以返回 101。

arr=[1,100,2]。

开始时玩家 A 不管拿 1 还是 2，玩家 B 作为绝顶聪明的人，都会把 100 拿走。玩家 B 会获胜，分数为 100。所以返回 100。

【难度】

尉 ★★☆☆

【解答】

暴力递归的方法。定义递归函数 $f(i,j)$，表示如果 arr[i..j]这个排列上的纸牌被绝顶聪明的人先拿，最终能获得什么分数。定义递归函数 $s(i,j)$，表示如果 arr[i..j]这个排列上的纸牌被绝顶聪明的人后拿，最终能获得什么分数。

首先来分析 $f(i,j)$，具体过程如下：

1．如果 i==j（即 arr[i..j]）上只剩一张纸牌。当然会被先拿纸牌的人拿走，所以返回 arr[i]。

2．如果 i!=j。当前拿纸牌的人有两种选择，要么拿走 arr[i]，要么拿走 arr[j]。如果拿走 arr[i]，那么排列将剩下 arr[i+1..j]。对当前的玩家来说，面对 arr[i+1..j]排列的纸牌，他成

了后拿的人，所以后续他能获得的分数为 $s(i+1,j)$。如果拿走 arr[j]，那么排列将剩下 arr[i..j-1]。对当前的玩家来说，面对 arr[i..j-1]排列的纸牌，他成了后拿的人，所以后续他能获得的分数为 $s(i,j-1)$。作为绝顶聪明的人，必然会在两种决策中选最优的。所以返回 max{arr[i]+s(i+1,j) , arr[j]+s(i,j-1)}。

然后来分析 $s(i,j)$，具体过程如下：

1．如果 i==j（即 arr[i..j]）上只剩一张纸牌。作为后拿纸牌的人必然什么也得不到，返回 0。

2．如果 i!=j。根据函数 s 的定义，玩家的对手会先拿纸牌。对手要么拿走 arr[i]，要么拿走 arr[j]。如果对手拿走 arr[i]，那么排列将剩下 arr[i+1..j]，然后轮到玩家先拿。如果对手拿走 arr[j]，那么排列将剩下 arr[i..j-1]，然后轮到玩家先拿。对手也是绝顶聪明的人，所以必然会把最差的情况留给玩家。所以返回 min{f(i+1,j) , f(i,j-1)}。

具体过程请参看如下代码中的 win1 方法。

```
public int win1(int[] arr) {
        if (arr == null || arr.length == 0) {
                return 0;
        }
        return Math.max(f(arr, 0, arr.length - 1), s(arr, 0, arr.length - 1));
}

public int f(int[] arr, int i, int j) {
        if (i == j) {
                return arr[i];
        }
        return Math.max(arr[i] + s(arr, i + 1, j), arr[j] + s(arr, i, j - 1));
}

public int s(int[] arr, int i, int j) {
        if (i == j) {
                return 0;
        }
        return Math.min(f(arr, i + 1, j), f(arr, i, j - 1));
}
```

暴力递归的方法中，递归函数一共会有 N 层，并且是 f 和 s 交替出现的。$f(i,j)$会有 $s(i+1,j)$和 $s(i,j-1)$两个递归分支，$s(i,j)$也会有 $f(i+1,j)$和 $f(i,j-1)$两个递归分支。所以整体的时间复杂度为 $O(2^N)$，额外空间复杂度为 $O(N)$。下面介绍动态规划的方法，如果 arr 长度为 N，生成两个大小为 $N\times N$ 的矩阵 f 和 s，f[i][j]表示函数 $f(i,j)$的返回值，s[i][j]表示函数 $s(i,j)$的返回值。规定一下两个矩阵的计算方向即可。具体过程请参看如下代码中的 win2 方法。

```
public int win2(int[] arr) {
```

```
        if (arr == null || arr.length == 0) {
            return 0;
        }
        int[][] f = new int[arr.length][arr.length];
        int[][] s = new int[arr.length][arr.length];
        for (int j = 0; j < arr.length; j++) {
            f[j][j] = arr[j];
            for (int i = j - 1; i >= 0; i--) {
                f[i][j] = Math.max(arr[i] + s[i + 1][j], arr[j] + s[i][j - 1]);
                s[i][j] = Math.min(f[i + 1][j], f[i][j - 1]);
            }
        }
        return Math.max(f[0][arr.length - 1], s[0][arr.length - 1]);
    }
```

如上的 win2 方法中，矩阵 f 和 s 一共有 $O(N^2)$个位置，每个位置计算的过程都是 $O(1)$的比较过程，所以 win2 方法的时间复杂度为 $O(N^2)$，额外空间复杂度为 $O(N^2)$。

跳跃游戏

【题目】

给定数组 arr，arr[i]==k 代表可以从位置 i 向右跳 1~k 个距离。比如，arr[2]==3，代表从位置 2 可以跳到位置 3、位置 4 或位置 5。如果从位置 0 出发，返回最少跳几次能跳到 arr 最后的位置上。

【举例】

arr=[3,2,3,1,1,4]。

arr[0]==3，选择跳到位置 2；arr[2]==3，可以跳到最后的位置。所以返回 2。

【要求】

如果 arr 长度为 N，要求实现时间复杂度为 $O(N)$、额外空间复杂度为 $O(1)$的方法。

【难度】

士 ★☆☆☆

【解答】

具体过程如下：

1．整型变量 jump，代表目前跳了多少步。整型变量 cur，代表如果只能跳 jump 步，最远能够达到的位置。整型变量 next，代表如果再多跳一步，最远能够达到的位置。初始时，jump=0，cur=0，next=0。

2．从左到右遍历 arr，假设遍历到位置 *i*。

1）如果 cur>=i，说明跳 jump 步可以到达位置 *i*，此时什么也不做。

2）如果 cur<i，说明只跳 jump 步不能到达位置 *i*，需要多跳一步才行。此时令 jump++，cur=next。表示多跳了一步，cur 更新成跳 jump+1 步能够达到的位置，即 next。

3）将 next 更新成 math.max(next, i+arr[i])，表示下一次多跳一步到达的最远位置。

3．最终返回 jump 即可。

具体过程请参看如下代码中的 jump 方法。

```
public int jump(int[] arr) {
        if (arr == null || arr.length == 0) {
                return 0;
        }
        int jump = 0;
        int cur = 0;
        int next = 0;
        for (int i = 0; i < arr.length; i++) {
                if (cur < i) {
                        jump++;
                        cur = next;
                }
                next = Math.max(next, i + arr[i]);
        }
        return jump;
}
```

数组中的最长连续序列

【题目】

给定无序数组 arr，返回其中最长的连续序列的长度。

【举例】

arr=[100,4,200,1,3,2]，最长的连续序列为[1,2,3,4]，所以返回 4。

【难度】

尉 ★★☆☆

【解答】

本题利用哈希表可以实现时间复杂度为 $O(N)$、额外空间复杂度为 $O(N)$的方法。具体过程如下：

1．生成哈希表 HashMap<Integer, Integer> map，key 代表遍历过的某个数，value 代表 key 这个数所在的最长连续序列的长度。同时 map 还可以表示 arr 中的一个数之前是否出现过。

2．从左到右遍历 arr，假设遍历到 arr[i]。如果 arr[i]之前出现过，直接遍历下一个数，只处理之前没出现过的 arr[i]。首先在 map 中加入记录(arr[i],1)，代表目前 arr[i]单独作为一个连续序列。然后看 map 中是否含有 arr[i]-1，如果有，则说明 arr[i]-1 所在的连续序列可以和 arr[i]合并，合并后记为 A 序列。利用 map 可以得到 A 序列的长度，记为 lenA，最小值记为 leftA，最大值记为 rightA，只在 map 中更新与 leftA 和 rightA 有关的记录，更新成(leftA,lenA)和(rightA,lenA)。接下来看 map 中是否含有 arr[i]+1，如果有，则说明 arr[i]+1 所在的连续序列可以和 A 合并，合并后记为 B 序列。利用 map 可以得到 B 序列的长度为 lenB，最小值记为 leftB，最大值记为 rightB，只在 map 中更新与 leftB 和 rightB 有关的记录，更新成(leftB,lenB)和(rightB,lenB)。

3．遍历的过程中用全局变量 max 记录每次合并出的序列的长度最大值，最后返回 max。

整个过程中，只是每个连续序列最小值和最大值在 map 中的记录有意义，中间数的记录不再更新，因为再也不会使用到。这是因为我们只处理之前没出现的数，如果一个没出现的数能够把某个连续区间扩大，或把某两个连续区间连在一起，毫无疑问，只需要 map 中有关这个连续区间最小值和最大值的记录。

具体过程请参看如下代码中的 longestConsecutive 方法。

```
public int longestConsecutive(int[] arr) {
    if (arr == null || arr.length == 0) {
        return 0;
    }
    int max = 1;
    HashMap<Integer, Integer> map = new HashMap<Integer, Integer>();
    for (int i = 0; i < arr.length; i++) {
        if (!map.containsKey(arr[i])) {
            map.put(arr[i], 1);
            if (map.containsKey(arr[i] - 1)) {
                max = Math.max(max, merge(map, arr[i] - 1, arr[i]));
            }
```

```
                if (map.containsKey(arr[i] + 1)) {
                    max = Math.max(max, merge(map, arr[i], arr[i] + 1));
                }
            }
        }
        return max;
    }

    public int merge(HashMap<Integer, Integer> map, int less, int more) {
        int left = less - map.get(less) + 1;
        int right = more + map.get(more) - 1;
        int len = right - left + 1;
        map.put(left, len);
        map.put(right, len);
        return len;
    }
```

N 皇后问题

【题目】

N 皇后问题是指在 *N*×*N* 的棋盘上要摆 *N* 个皇后，要求任何两个皇后不同行、不同列，也不在同一条斜线上。给定一个整数 *n*，返回 *n* 皇后的摆法有多少种。

【举例】

n=1，返回 1。

n=2 或 3，2 皇后和 3 皇后问题无论怎么摆都不行，返回 0。

n=8，返回 92。

【难度】

校 ★★★☆

【解答】

本题是非常著名的问题，甚至可以用人工智能相关算法和遗传算法进行求解，同时可以用多线程技术达到缩短运行时间的效果。本书不涉及专项算法，仅提供在面试过程中 10 至 20 分钟内可以用代码实现的解法。本书提供的最优解做到在单线程的情况下，计算 16 皇后问题的运行时间约为 13 秒左右。在介绍最优解之前，先来介绍一个容易理解的解法。

如果在(*i*,*j*)位置(第 *i* 行第 *j* 列)放置了一个皇后，接下来在哪些位置不能放置皇后呢？

1．整个第 i 行的位置都不能放置。

2．整个第 j 列的位置都不能放置。

3．如果位置(a,b)满足$|a-i|==|b-j|$，说明(a,b)与(i,j)处在同一条斜线上，也不能放置。

把递归过程直接设计成逐行放置皇后的方式，可以避开条件 1 的那些不能放置的位置。接下来用一个数组保存已经放置的皇后位置，假设数组为 record，record[i]的值表示第 i 行皇后所在的列数。在递归计算到第 i 行第 j 列时，查看 record[0..k]($k<i$)的值，看是否有 j 相等的值，若有，则说明(i,j)不能放置皇后，再看是否有$|k-i|==|record[k]-j|$，若有，也说明(i,j)不能放置皇后。具体过程请参看如下代码中的 num1 方法。

```
public int num1(int n) {
        if (n < 1) {
                return 0;
        }
        int[] record = new int[n];
        return process1(0, record, n);
}

public int process1(int i, int[] record, int n) {
        if (i == n) {
                return 1;
        }
        int res = 0;
        for (int j = 0; j < n; j++) {
                if (isValid(record, i, j)) {
                        record[i] = j;
                        res += process1(i + 1, record, n);
                }
        }
        return res;
}

public boolean isValid(int[] record, int i, int j) {
    for (int k = 0; k < i; k++) {
        if (j == record[k] || Math.abs(record[k] - j) == Math.abs(i - k)) {
                return false;
        }
    }
    return true;
}
```

下面介绍最优解，基本过程与上面的方法一样，但使用了位运算来加速。具体加速的递归过程中，找到每一行还有哪些位置可以放置皇后的判断过程。因为整个过程比较超自然，所以先列出代码，然后对代码进行解释，请参看如下代码中的 num2 方法。

```
public int num2(int n) {
```

```
        // 因为本方法中位运算的载体是 int 型变量，所以该方法只能算 1~32 皇后问题
        // 如果想计算更多的皇后问题，需使用包含更多位的变量
        if (n < 1 || n > 32) {
                return 0;
        }
        int upperLim = n == 32 ? -1 : (1 << n) - 1;
        return process2(upperLim, 0, 0, 0);
}

public int process2(int upperLim, int colLim, int leftDiaLim,
                int rightDiaLim) {
        if (colLim == upperLim) {
                return 1;
        }
        int pos = 0;
        int mostRightOne = 0;
        pos = upperLim & (~(colLim | leftDiaLim | rightDiaLim));
        int res = 0;
        while (pos != 0) {
                mostRightOne = pos & (~pos + 1);
                pos = pos - mostRightOne;
                res += process2(upperLim, colLim | mostRightOne,
                                (leftDiaLim | mostRightOne) << 1,
                                (rightDiaLim | mostRightOne) >>> 1);
        }
        return res;
}
```

num2 方法中，变量 upperLim 表示当前行哪些位置是可以放置皇后的，1 代表可以放置，0 代表不能放置。8 皇后问题中，初始时 upperLim 为 00000000000000000000000011111111，即 32 位整数的 255。32 皇后问题中，初始时 upperLim 为 11111111111111111111111111111111，即 32 位整数的-1。

接下来解释一下 process2 方法，先介绍每个参数。

- upperLim：已经解释过了，而且这个变量的值在递归过程中是始终不变的。
- colLim：表示递归计算到上一行为止，在哪些列上已经放置了皇后，1 代表已经放置，0 代表没有放置。
- leftDiaLim：表示递归计算到上一行为止，因为受已经放置的所有皇后的左下方斜线的影响，导致当前行不能放置皇后，1 代表不能放置，0 代表可以放置。举个例子，如果在第 0 行第 4 列放置了皇后。计算到第 1 行时，第 0 行皇后的左下方斜线影响的是第 1 行第 3 列。当计算到第 2 行时，第 0 行皇后的左下方斜线影响的是第 2 行第 2 列。当计算到第 3 行时，影响的是第 3 行第 1 列。当计算到第 4 行时，影响的是第 4 行第 0 列。当计算到第 5 行时，第 0 行的那个皇后的左下方斜

线对第 5 行无影响，并且之后的行都不再受第 0 行皇后左下方斜线的影响。也就是说，leftDiaLim 每次左移一位，就可以得到之前所有皇后的左下方斜线对当前行的影响。

- rightDiaLim：表示递归计算到上一行为止，因为已经放置的所有皇后的右下方斜线的影响，导致当前行不能放置皇后的位置，1 代表不能放置，0 代表可以放置。与 leftDiaLim 变量类似，rightDiaLim 每次右移一位就可以得到之前所有皇后的右下方斜线对当前行的影响。

process2 方法的返回值代表剩余的皇后在之前皇后的影响下，有多少种合法的摆法。其中，变量 pos 代表当前行在 colLim、leftDiaLim 和 rightDiaLim 这三个状态的影响下，还有哪些位置是可供选择的，1 代表可以选择，0 代表不能选择。变量 mostRightOne 代表在 pos 中，最右边的 1 是在什么位置。然后从右到左依次筛选出 pos 中可选择的位置进行递归尝试。