```
        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];
}
```

最终位置的状态

最后一行，只能朝右走

最后一列，只能朝下走

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