- 特性
  - 无后效性
  - 存在最优子问题
  - 问方案数，但不问具体方案的时候，可以考虑DP
  - DP 处理字符串问题的思想是：从一个空串开始，一点一点得到更大规模的问题的解
  - 数组注意不要越界，可以多定义一行一列
- 字符串与DP
  - 字符串定义为DP时，将字符串转为数组
  - 两个字符串的比较变二维（升维）
- DP
  - 字符解码数问题

```java
public int numDecodingsDPOpt(String s) {
  if (s == null || s.length() < 1 || s.charAt(0) == '0') {
    return 0;
  }
  int length = s.length();
  int prepre = 1, pre = s.charAt(0) == '0' ? 0 : 1;

  for (int i = 2; i <= length; i++) {
    int cur = 0;
    int first = Integer.valueOf(s.substring(i - 1, i));
    int second = Integer.valueOf(s.substring(i - 2, i));
    if (first > 0) {
      cur = pre;
    }
    if (second >= 10 && second <= 26) {
      cur += prepre;
    }
    prepre = pre;
    pre = cur;
  }
  return pre;
}
```

  - 最小路径和问题
    - 大问题：从起点走到终点的最小路径和
    - 小问题：从起点走到grid[i][j]的最小路径和，可以从上或者从左走到[i][j]
    - 空间O(M*N), dp[i][j] = grid[i][j] + min(dp[i–1][j], dp[i][j–1]
    - 因为每次只与上一行运行结果以及本行上一列比较，可以优化为O(N): dp[j] = grid[i][j] + min(dp[j],dp[j–1])

```java
// 优化：空间O(N)
public int minPathSumOpt(int[][] grid) {
  int rows = grid.length, cols = grid[0].length;
  // dp[j] = 从起点走到j的最短路径
  // dp[j] = grid[i][j] + Math.min(dp[j], dp[j - 1]);
  int[] dp = new int[cols];
  // 初始化
  dp[0] = grid[0][0];
  for (int j = 1; j < cols; j++) {
    dp[j] = grid[0][j] + dp[j - 1];
  }

  for (int i = 1; i < rows; i++) {
    dp[0] += grid[i][0];
    for (int j = 1; j < cols; j++) {
      dp[j] = grid[i][j] + Math.min(dp[j], dp[j - 1]);
    }
  }
  return dp[cols - 1];
}
```

  - 最大正方形
    - dp[i][j]:以matrix[i–1][j–1]为右正解的最大边长
    - dp[i][j] = min(dp(左上角，上，左))，同时每次赋值的时候，记录下当前最大边长 maxSide
    - Return maxSide*maxSide
    - 优化：可将空间复杂度从m*n降到n，但需单独处理左上角。即每行的时候，nw=0,处理每个dp元素的时候，先将dp[i]保存下来作为下一次nw

```java
// 方法一：DP
public int maximalSquare(char[][] matrix) {
  int rows = matrix.length, cols = matrix[0].length;
  int maxSide = 0;
  int[][] dp = new int[rows + 1][cols + 1];

  // DP(i,j): 以matrix(i-1,j-1)为右下角的正方形的最大边长
  //    if (grid[i - 1][j - 1] == '1') {
  //      dp[i][j] = min(dp[i - 1][j - 1], dp[i - 1][j], dp[i][j - 1]) + 1;
  //    }
  // 需要返回面积，所以在计算每个dp[i][j]的时候，就把当前最大边长记下

  for (int i = 1; i <= rows; i++) {
    for (int j = 1; j <= cols; j++) {
      if (matrix[i - 1][j - 1] == '1') {
        dp[i][j] = Math.min(Math.min(dp[i - 1][j - 1], dp[i - 1][j]), dp[i][j - 1]) + 1;
        maxSide = Math.max(maxSide, dp[i][j]);
      }
    }
  }
  return maxSide * maxSide;
}
```