

# OneStepOffer 算法讲义

## 第六讲

Dynamic Programming Part 2

## 二维dp - Maximal Square

给定一个二维的 0-1 矩阵, 求全由 1 构成的最大正方形面积。

输入:

```
[["1","0","1","0","0"],
```

```
["1","0","1","1","1"],
```

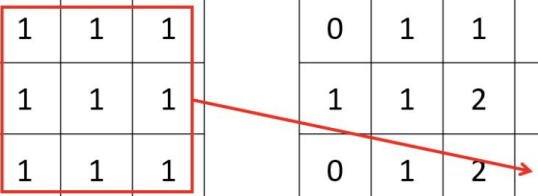
```
["1","1","1","1","1"],
```

```
["1","0","0","1","0"]]
```

输出:4

## 题目分析

对于在矩阵内搜索正方形或长方形的题型，一种常见的做法是定义一个二维 dp 数组，其中  $dp[i][j]$  表示满足题目条件的、以  $(i, j)$  为右下角的正方形或者长方形的属性。对于本题，则表示以  $(i, j)$  为右下角的全由 1 构成的最大正方形面积。如果当前位置是 0，那么  $dp[i][j]$  即为 0；如果当前位置是 1，我们假设  $dp[i][j] = k^2$ ，其充分条件为  $dp[i-1][j-1]$ 、 $dp[i][j-1]$  和  $dp[i-1][j]$  的值必须都不小于  $(k-1)^2$ ，否则  $(i, j)$  位置不可以构成一个边长为  $k$  的正方形。同理，如果这三个值中的最小值为  $k-1$ ，则  $(i, j)$  位置一定且最大可以构成一个边长为  $k$  的正方形。



0	0	1	0
0	1	1	1
1	1	1	1
0	1	1	1

0	0	1	0
0	1	1	1
1	1	2	2
0	1	2	3

## 代码讲解 - C++

```
int maximalSquare(vector<vector<char>>& matrix) {  
    if (matrix.empty() || matrix[0].empty()) {  
        return 0;  
    }  
    int m = matrix.size(), n = matrix[0].size(), max_side = 0;  
    vector<vector<int>> dp(m + 1, vector<int>(n + 1, 0));  
    for (int i = 1; i <= m; ++i) {  
        for (int j = 1; j <= n; ++j) {  
            if (matrix[i-1][j-1] == '1') {  
                dp[i][j] = min(dp[i-1][j-1], min(dp[i][j-1], dp[i-1][j])) + 1;  
            }  
            max_side = max(max_side, dp[i][j]);  
        }  
    }  
    return max_side * max_side;  
}
```

# 代码讲解 java

```
public int maximalSquare(char[][] matrix) {  
    if (matrix.length == 0 || matrix[0].length == 0) return 0;  
    int m = matrix.length; int n = matrix[0].length;  
    int[][] dp = new int[m+1][n+1];  
    int maxSide = 0;  
    for (int i = 1; i <= m; i++) {  
        for (int j = 1; j <= n; j++) {  
            if (matrix[i-1][j-1] == '1') {  
                dp[i][j] = Math.min(dp[i-1][j-1], Math.min(dp[i][j-1], dp[i-1][j])) + 1;  
            }  
            maxSide = Math.max(maxSide, dp[i][j]);  
        }  
    }  
  
    return maxSide * maxSide;  
}
```

# 代码讲解 - python

```
class Solution:
    def maximalSquare(self, matrix: List[List[str]]) -> int:
        for i, row in enumerate(matrix):
            for j, col in enumerate(row):
                matrix[i][j] = int(col)
            row.append(0)

        matrix.append([0] * (len(matrix[0])+1))
        besta = 0

        for i in range(len(matrix)-1, -1, -1):
            for j in range(len(matrix[0])-1, -1, -1):
                if matrix[i][j] and matrix[i+1][j] and matrix[i][j+1] and matrix[i+1][j+1]:
                    matrix[i][j] = min([matrix[i+1][j], matrix[i][j+1], matrix[i+1][j+1]]) + 1
                    besta = max(besta, matrix[i][j]**2)
        return besta
```

# 分割类型

对于分割类型题，动态规划的状态转移方程通常并不依赖相邻的位置，而是依赖于满足分割条件的位置。

解题步骤：先确认分割的状态方程 -- 我们上次课讲的子状态 然后划分条件和位置

再确认dp过程 最后规划dp的边角情况

# 分割类型题目 -- perfect Squares

题目描述: 给定一个正整数, 求其最少可以由几个完全平方数相加构成。

输入是给定的正整数, 输出也是一个正整数, 表示输入的数字最少可以由几个完全平方数相加构成。

Input:  $n = 13$

Output: 2

在这个样例中, 13 的最少构成方法为  $4+9$ 。



## 题目分析

我们定义一个一维矩阵  $dp$ , 其中  $dp[i]$  表示数字  $i$  最少可以由几个完全平方数相加构成。在本题中, 位置  $i$  只依赖  $i - k^2$  的位置, 如  $i - 1$ 、 $i - 4$ 、 $i - 9$  等等, 才能满足完全平方分割的条件。因此  $dp[i]$  可以取的最小值即为  $1 + \min(dp[i-1], dp[i-4], dp[i-9] \cdots)$ 。

# C++ 代码

```
int numSquares(int n) {  
    vector<int> dp(n + 1, INT_MAX); dp[0] = 0;  
    for (int i = 1; i <= n; ++i) {  
        for (int j = 1; j * j <= i; ++j) {  
            dp[i] = min(dp[i], dp[i-j*j] + 1);  
        }  
    }  
    return dp[n];  
}
```

# Java 代码

```
public int numSquares(int n) {  
    int[] dp = new int[n+1];  
    Arrays.fill(dp, Integer.MAX_VALUE);  
  
    dp[0] = 0;  
    for (int i = 1; i <= n; i++) {  
        for (int j = 1; j*j <= i; j++) {  
            dp[i] = Math.min(dp[i], dp[i-j*j] + 1)  
        }  
    }  
  
    return dp[n];  
}
```

# Python 代码

```
class Solution(object):
    def numSquares(self, n):
        square_nums = [i**2 for i in range(0, int(math.sqrt(n))+1)]

        dp = [float('inf')] * (n+1)
        # bottom case
        dp[0] = 0

        for i in range(1, n+1):
            for square in square_nums:
                if i < square:
                    break
                dp[i] = min(dp[i], dp[i-square] + 1)

        return dp[-1]
```

## 分割类型题目二: Decode Ways

已知字母 A-Z 可以表示成数字 1-26。给定一个数字串, 求有多少种不同的字符串等价于 这个 数字串。

输入是一个由数字组成的字符串, 输出是满足条件的解码方式总数。

Input: "226"

Output: 3

在这个样例中, 有三种解码方式: BZ(2 26)、VF(22 6) 或 BBF(2 2 6)。

# 题目分析

这是一道很经典的动态规划题，难度不大但是十分考验耐心。这是因为只有 1-26 可以表示字母，因此对于一些特殊情况，比如数字0 或者当相邻两数字大于 26 时，需要有不同的状态转移方程。

$$dp[i] = dp[i-1] + dp[i-2]$$

## 具体操作

如果字符串  $s$  为空或 `null`, 则返回结果为0。

初始化  $dp$  数组。  $dp[0] = 1$  以提供要传递的指挥棒。

如果字符串的第一个字符 为零, 则无法解码, 因此将  $dp[1]$  初始化为0, 否则第一个字符无法有效地传递给后面,  $dp[1] = 1$ 。

从索引2开始迭代  $dp$  数组。  $dp$  的索引  $i$  是字符串  $s$  的第  $i$  个字符, 即字符串  $s$  的索引  $i-1$  的字符。

我们检查是否可以进行有效的一位数字解码。这仅表示索引  $s[i-1]$  处的字符为非零。由于我们没有零解码。如果可以进行有效的一位数字解码, 则将  $dp[i-1]$  添加到  $dp[i]$ 。由于直到第  $(i-1)$  个字符的所有方式现在也导致了第  $i$  个字符。

我们检查是否可以进行有效的两位数解码。这意味着子字符串  $s[i-2]s[i-1]$  在10到26之间。如果可能进行有效的两位数解码, 则将  $dp[i-2]$  添加到  $dp[i]$ 。

一旦我们到达  $dp$  数组的末尾, 我们将有多种解码字符串  $s$  的方法。

# 代码详解 - C++

```
int numDecodings(string s) {  
    vector<int> dp(s.length() + 1);  
    dp[0] = 1;  
    dp[1] = s[0] == '0' ? 0 : 1;  
  
    // Ways to decode a string of size 1 is 1. Unless the string is '0'.  
    // '0' doesn't have a single digit decode.  
    for (size_t i = 2; i < dp.size(); i++) {  
        if (s[i - 1] != '0') dp[i] = dp[i - 1];  
  
        // Check if successful two digit decode is possible.  
        int two_digit = stoi(s.substr(i - 2, 2));  
        if (two_digit >= 10 && two_digit <= 26) dp[i] += dp[i - 2];  
    }  
    return dp[s.length()];  
}
```



## 代码详解 - Java

```
public int numDecodings(String s) {  
    int[] dp = new int[s.length() + 1];  
    dp[0] = 1;  
    dp[1] = s.charAt(0) == '0' ? 0 : 1;  
    for(int i = 2; i < dp.length; i++) {  
        if (s.charAt(i - 1) != '0') {  
            dp[i] = dp[i - 1];  
        }  
        int twoDigit = Integer.valueOf(s.substring(i - 2, i));  
        if (twoDigit >= 10 && twoDigit <= 26) {  
            dp[i] += dp[i - 2];  
        }  
    }  
    return dp[s.length()];  
}
```

# 代码详解 - python

```
def numDecodings(self, s: str) -> int:
    dp = [0 for _ in range(len(s) + 1)]

    dp[0] = 1
    dp[1] = 0 if s[0] == '0' else 1

    for i in range(2, len(dp)):
        if s[i - 1] != '0':
            dp[i] = dp[i - 1]
        two_digit = int(s[i - 2 : i])
        if two_digit >= 10 and two_digit <= 26:
            dp[i] += dp[i - 2]

    return dp[len(s)]
```

## 例题3 - Word Break (Medium)

题目描述: 给定一个字符串和一个字符串集合, 求是否存在一种分割方式, 使得原字符串分割后的子字符串都可以在集合内找到。

Input: s = "applepenapple", wordDict = ["apple", "pen"]

Output: true

在这个样例中, 字符串可以被分割为 ["apple", "pen", "apple"]。

## 题目解答

类似于完全平方数分割问题，这道题的分割条件由集合内的字符串决定，因此在考虑每个分割位置时，需要遍历字符串集合，以确定当前位置是否可以成功分割。注意 对于位置 0，需要初始化值为真。

## C++ 答案

```
bool wordBreak(string s, vector<string>& wordDict) {  
    int n = s.length();  
    vector<bool> dp(n + 1, false);  
    dp[0] = true;  
    for (int i = 1; i <= n; ++i) {  
        for (const string & word: wordDict) {  
            int len = word.length();  
            if (i >= len && s.substr(i - len, len) == word) {  
                dp[i] = dp[i] || dp[i - len];  
            }  
        }  
    }  
    return dp[n];  
}
```

## 每周5题

300. Longest Increasing Subsequence (Medium) - 给定一个未排序的整数数组, 求最长的递增子序列。

416. Partition Equal Subset Sum (Medium) - 给定一个正整数数组, 求是否可以把这个数组分成和相等的两部分。

322. Coin Change (Medium) - 给定一些硬币的面额, 求最少可以用多少颗硬币组成给定的金额。

72. Edit Distance (Hard) - 给定两个字符串, 已知你可以删除、替换和插入任意字符串的任意字符, 求最少编辑几步可以将两个字符串变成相同。

188. Best Time to Buy and Sell Stock IV (Hard) - 给定一段时间内每天的股票价格, 已知你只可以买卖各  $k$  次, 且每次只能拥有一支股票, 求最大的收益。