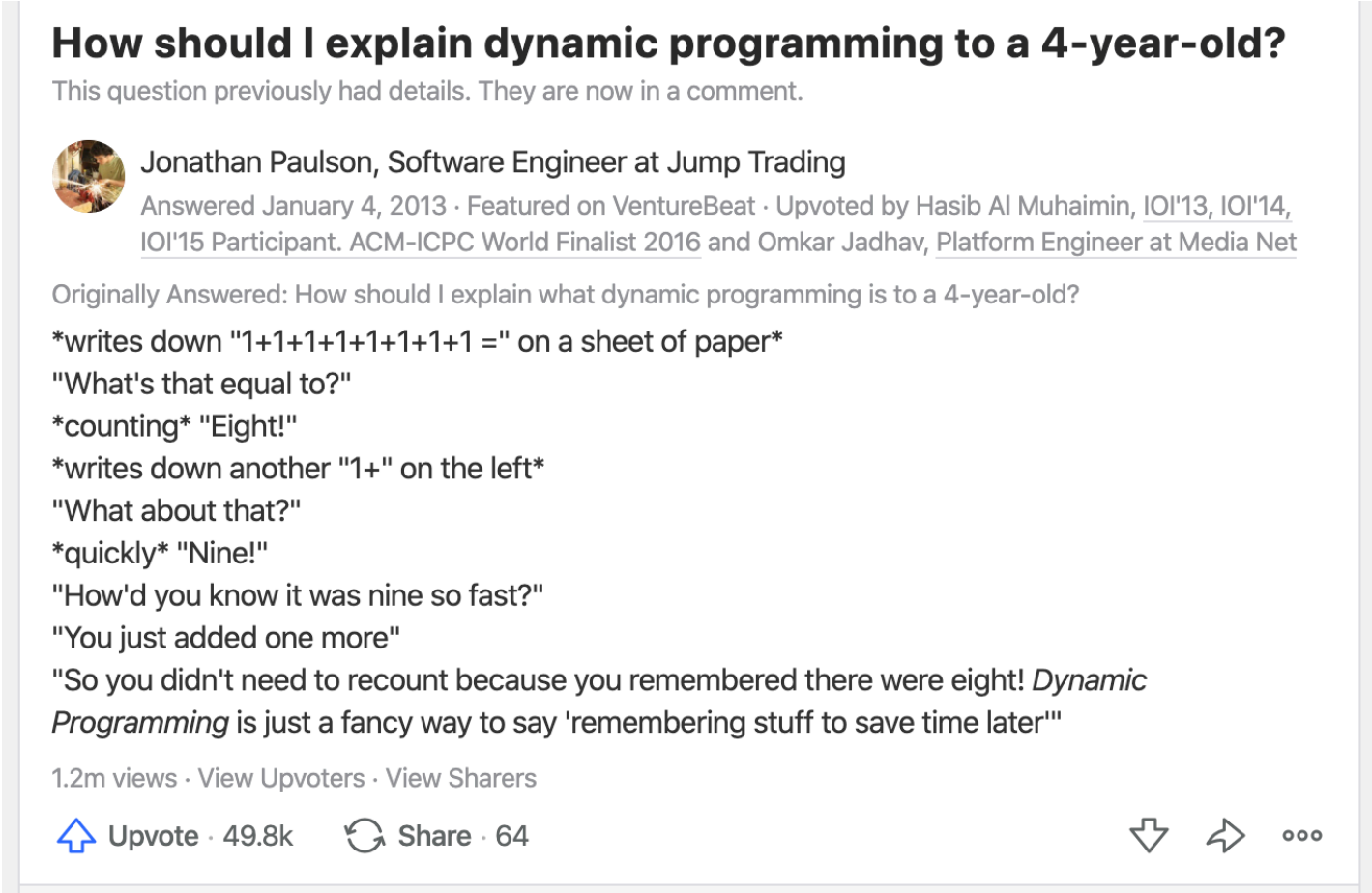
简介

动态规划 是一种将一个复杂问题分解为多个简单的子问题求解的方法。将子问题的答案存储在记忆数据结构中，当子问题再次需要解决时，只需查表查看结果，而不需要再次重复计算，因此节约了计算时间。
国外知乎 Quora 上一个帖子问应该怎样给四岁的孩子解释什么是动态规划，其中一个非常经典的回答如下：
How should I explain dynamic programming to a 4-year-old?



How should I explain dynamic programming to a 4-year-old?

This question previously had details. They are now in a comment.

Jonathan Paulson, Software Engineer at Jump Trading

Answered January 4, 2013 · Featured on VentureBeat · Upvoted by Hasib Al Muhaimin, IOI'13, IOI'14, IOI'15 Participant. ACM-ICPC World Finalist 2016 and Omkar Jadhav, Platform Engineer at Media Net

Originally Answered: How should I explain what dynamic programming is to a 4-year-old?

writes down "1+1+1+1+1+1+1+1 =" on a sheet of paper
"What's that equal to?"
counting "Eight!"
writes down another "1+" on the left
"What about that?"
quickly "Nine!"
"How'd you know it was nine so fast?"
"You just added one more"
"So you didn't need to recount because you remembered there were eight! *Dynamic Programming* is just a fancy way to say 'remembering stuff to save time later'"

1.2m views · View Upvoters · View Sharers

Upvote · 49.8k Share · 64

动态规划通常基于一个递推公式和一个或多个初始状态。当前问题的解可以分解为多个子问题解得出。使用动态规划只需要多项式时间复杂度，因为比回溯法和暴力法快很多。

动态规划中非常重要的两个概念：状态和状态转移方程。

下面通过例子由浅至深详细讲解。
所有例题均来自leetcode，所示代码均通过所有测试。

参考文章，将所有的DP问题分成11大类，本文将每一类的题目进行补充，并对这些题目的解法进行探讨。

题目

1、线性 DP

- 最经典单串：
  300.最长上升子序列 中等
- 其他单串
  32.最长有效括号 困难
  376.摆动序列
  368.最大整除子集
  410.分割数组的最大值
- 最经典双串：
  1143.最长公共子序列 中等
- 其他双串
  97.交错字符串 中等
  115.不同的子序列 困难
  583.两个字符串的删除操作

- 经典问题：
  53.最大子序和 简单
  120.三角形最小路径和 中等
  152.乘积最大子数组 中等
  354.俄罗斯套娃信封问题
  887.鸡蛋掉落（DP+二分） 困难

- 打家劫舍系列: (打家劫舍3 是树形DP)
  198.打家劫舍 中等
  213.打家劫舍 II 中等

- 股票系列:
  121.买卖股票的最佳时机
  122.买卖股票的最佳时机 II
  123.买卖股票的最佳时机 III
  188.买卖股票的最佳时机 IV
  309.最佳买卖股票时机含冷冻期
  714.买卖股票的最佳时机含手续费

- 字符串匹配系列
  72.编辑距离 困难
  44.通配符匹配 困难
  10.正则表达式匹配 困难

- 其他
  375.猜数字大小 II

2、区间 DP

5.最长回文子串 中等
516.最长回文子序列
87. 扰乱字符串 困难
312.戳气球 困难
730.统计不同回文子字符串
1039.多边形三角剖分的最低得分
664.奇怪的打印机
1246. 删除回文子数组

3、背包 DP

377. 组合总和 IV
416.分割等和子集 (01背包-要求恰好取到背包容量)
494.目标和 (01背包-求方案数)
322.零钱兑换 (完全背包)
518.零钱兑换 II (完全背包-求方案数)
474.一和零 (二维费用背包)

4、树形 DP

124.二叉树中的最大路径和 困难
1245.树的直径 (邻接表上的树形DP)
543.二叉树的直径 简单
333.最大 BST 子树
337.打家劫舍 III 中等

5、状态压缩 DP

464.我能赢吗
526.优美的排列
935.骑士拨号器
1349.参加考试的最大学生数

6、数位 DP

233.数字 1 的个数 困难
902.最大为 N 的数字组合
1015.可被 K 整除的最小整数

7、计数型 DP

计数型DP都可以以组合数学的方法写出组合数，然后dp求组合数
62.不同路径
63.不同路径 II
96.不同的二叉搜索树
1259.不相交的握手 (卢卡斯定理求大组合数模质数)

8、递推型 DP

70.爬楼梯
509.斐波那契数
576. 出界的路径数
688. “马”在棋盘上的概率
935.骑士拨号器
957.N 天后的牢房
1137.第 N 个泰波那契数

9、概率型 DP

求概率，求数学期望
808.分汤
837.新21点

10、博弈型 DP

策梅洛定理，SG 定理，minimax

- 翻转游戏
  293.翻转游戏
  294.翻转游戏 II
- Nim游戏
  292.Nim 游戏
- 石子游戏
  877.石子游戏
  1140.石子游戏 II
- 井字游戏
  348.判定井字棋胜负
  794.有效的井字游戏
  1275.找出井字棋的获胜者

11、记忆化搜索

本质是 dfs + 记忆化，用在状态的转移方向不确定的情况
329.矩阵中的最长递增路径
576.出界的路径数

解析

1、线性 DP

300. 最长上升子序列

题目描述

给你一个整数数组 nums ，找到其中最长严格递增子序列的长度。

子序列是由数组派生而来的序列，删除（或不删除）数组中的元素而不改变其余元素的顺序。例如，[3,6,2,7] 是数组 [0,3,1,6,2,2,7] 的子序列。

示例 1：

> 输入：nums = [10,9,2,5,3,7,101,18]
> 输出：4
> 解释：最长递增子序列是 [2,3,7,101]，因此长度为 4 。

分析

1. DP。
   定义dp[i]，表示
   转移方程：如果nums[j] > nums[i]，dp[i] = dp[j] + 1。
   初始状态：dp[i] = 1，表示只有一个元素的递增子序列。
   时间复杂度：O(n^2)，空间复杂度：O(n)

2. 贪心+二分。
   维护一个单调递增的数组d[i]，表示长度为 i 的最长上升子序列的末尾元素的最小值。起始长度为1，d[1] = nums[0].
   以输入序列 [0, 8, 4, 12, 2] 为例：
   第一步插入 0，d=[0]；
   第二步插入 8，d=[0,8]；
   第三步插入 4，d=[0,4]；
   第四步插入 12，d=[0,4,12]；
   第五步插入 2，d=[0,2,12]。
   最终得到最大递增子序列长度为 3。
   时间复杂度：O(nlogn)，空间复杂度：O(n)

代码

- DP

```
class Solution {
    public int lengthOfLIS(int[] nums) {
        int n = nums.length;
        int res = 1;
        int[] dp = new int[n];
        for (int i = 0; i < n; i++) {
            dp[i] = 1;
            for (int j = 0; j < i; j++) {
                if (nums[i] > nums[j]) {
                    dp[i] = Math.max(dp[i], dp[j] + 1);
                    res = Math.max(res, dp[i]);
                }
            }
        }
        return res;
    }
}
```

- 贪心+二分

```
lass Solution {
    public int lengthOfLIS(int[] nums) {
        int n = nums.length;
        int[] d = new int[n+1];
        int len = 1;
        d[len] = nums[0];
        for (int i = 1; i < n; i++) {
            if (nums[i] > d[len]) {
                d[++len] = nums[i];
            } else {
```

进阶：需要返回最长的上升子序列?

```
public class Solution {
    public int[] LIS (int[] arr) {
        // write code here
        int n = arr.length;
        if (n == 0) {
            return new int[0];
        }
        int[] size = new int[n];   // 记录最长子序列的个数，用于事前不知道长度，初始化最长n
        int[] maxLen = new int[n]; // maxLen[i] 以i结尾的最长子序列
        int index = 0;
```


32. 最长有效括号

题目描述

给你一个只包含 ‘(’ 和 ‘)’ 的字符串，找出最长有效（格式正确且连续）括号子串的长度。

示例 1：

> 输入：s = “(()”
> 输出：2
> 解释：最长有效括号子串是 “()”

分析

定义 dp[i] 为以 i 结束的最长有效括号长度。
状态转移方程：

1. s[i] == ‘(’
   dp[i] = 0
2. s[i] == ‘)’
   a. s[i-1] == ‘(’ : dp[i] = dp[i-2] + 2
   例如：()()
   b. s[i-1] == ‘)’ and s[i - 1 - dp[i-1]] == ‘(’ : dp[i] = dp[i-1] + dp[i - 1 - dp[i-1] - 1] + 2
   例如：(())

注意数组越界情况。

代码

```
class Solution {
    public int longestValidParentheses(String s) {
        int n = s.length(), res = 0;
        int[] dp = new int[n];
        for (int i = 0; i < n; i++) {
            char c = s.charAt(i);
            if (c == '(') {
                dp[i] = 0;
            } else {
                if (i > 0 && s.charAt(i-1) == '(') {
```


进阶：
解法二：栈

```
class Solution {
    public int longestValidParentheses(String s) {
```

```
        Deque<Integer> stack = new LinkedList<>();
        stack.addLast(-1);

        int n = s.length(), res = 0;
        for (int i = 0; i < n; i++) {
            char c = s.charAt(i);
            if (c == '(') {
                stack.addLast(i);
```

1143. 最长公共子序列

题目描述

给定两个字符串 text1 和 text2，返回这两个字符串的最长 公共子序列 的长度。如果不存在 公共子序列 ，返回 0 。

一个字符串的 子序列 是指这样一个新的字符串：它是由原字符串在不改变字符的相对顺序的情况下删除某些字符（也可以不删除任何字符）后组成的新字符串。

例如，“ace” 是 “abcde” 的子序列，但 “aec” 不是 “abcde” 的子序列。
两个字符串的 公共子序列 是这两个字符串所共同拥有的子序列。

示例 1：

> 输入：text1 = “abcde”, text2 = “ace”
> 输出：3
> 解释：最长公共子序列是 “ace” ，它的长度为 3 。

分析

1. 使用DP。
   定义dp[i][j]，表示text1[0:i]和text[0:j]的最长公共子序列长度。
   转移方程：

$$\begin{cases} dp[i][j] = dp[i-1][j-1] + 1, & text1[i] = text2[j] \\ dp[i][j] = \max(dp[i][j-1], dp[i-1][j]), & text1[i] != text2[j] \end{cases}$$

代码

```
class Solution {
    public int longestCommonSubsequence(String text1, String text2) {
        int m = text1.length(), n = text2.length();
        int[][] dp = new int[m+1][n+1];
        for (int i = 1; i <= m; i++) {
            char c1 = text1.charAt(i - 1);
            for (int j = 1; j <= n; j++) {
                if (c1 == text2.charAt(j - 1)) {
                    dp[i][j] = dp[i-1][j-1] + 1;
                } else {
```

进阶：如果要求返回最长的子序列呢?
根据得到的dp矩阵，逆序寻找路径

```
public String LCS (String s1, String s2) {
    int m = s1.length(), n = s2.length();
    if (m == 0 || n == 0) {
        return "-1";
    }
    int[][] dp = new int[m + 1][n + 1];
```

```java
    for (int i = 1; i <= m; i++) {
        for (int j = 1; j <= n; j++) {
            if (s1.charAt(i - 1) == s2.charAt(j - 1)) {
                dp[i][j] = dp[i-1][j-1] + 1;
```

97. 交错字符串

题目描述

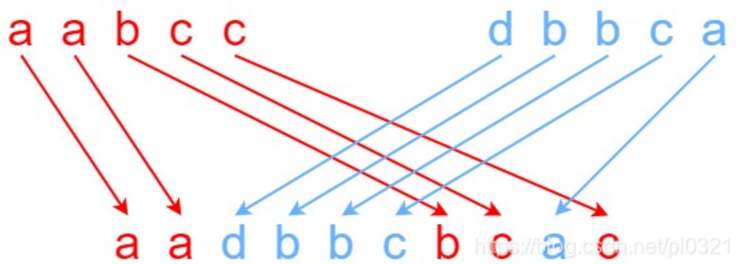
给定三个字符串 s1、s2、s3，请你帮忙验证 s3 是否是由 s1 和 s2 交错 组成的。

两个字符串 s 和 t 交错 的定义与过程如下，其中每个字符串都会被分割成若干 非空 子字符串：

- s = s1 + s2 + ... + sn
- t = t1 + t2 + ... + tm
- |n - m| <= 1
- 交错 是 s1 + t1 + s2 + t2 + s3 + t3 + ... 或者 t1 + s1 + t2 + s2 + t3 + s3 + ...

提示：a + b 意味着字符串 a 和 b 连接。

示例 1：



> 输入：s1 = “aabcc”，s2 = “dbbca”，s3 = “aadbbcbcac”
> 输出：true

分析

定义dp[i][j]，表示s1[:i]与s2[:j]交错组成s3[:i+j-1]
转移方程：

$$\begin{cases} dp[i][j] = true, & dp[i-1][j] \&\& s1[i] = s3[i+j-1] \\ dp[i][j] = true, & dp[i][j-1] \&\& s2[j] = s3[i+j-1] \end{cases}$$

$$\begin{cases} dp[i][j] = true, & dp[i-1][j] \&\& s1[i] = s3[i+j-1] \\ dp[i][j] = true, & dp[i][j-1] \&\& s2[j] = s3[i+j-1] \end{cases}$$

$$\begin{cases} dp[i][j] = true, & dp[i-1][j] \&\& s1[i] = s3[i+j-1] \\ dp[i][j] = true, & dp[i][j-1] \&\& s2[j] = s3[i+j-1] \end{cases}$$

初始状态：

$$\begin{cases} dp[0][0] = true; \\ dp[i][0] = true, & dp[i-1][0] \&\& s1[i] = s3[i] \\ dp[0][j] = true, & dp[0][j-1] \&\& s2[j] = s3[j] \end{cases}$$

代码

```java
class Solution {
    public boolean isInterleave(String s1, String s2, String s3) {
        int m = s1.length(), n = s2.length(), k = s3.length();
        if (m + n != k) {
```

```
            return false;
        }
        if (k == 0) {
            return true;
        }
        if (m == 0) {
```

进一步简单代码：

```
class Solution {
    public boolean isInterleave(String s1, String s2, String s3) {
        int m = s1.length(), n = s2.length(), k = s3.length();
        if (m + n != k) {
            return false;
        }
        boolean[][] dp = new boolean[m + 1][n + 1];
        dp[0][0] = true;

        for (int i = 0; i <= m; i++) {
```

115.不同的子序列

题目描述

给定一个字符串 s 和一个字符串 t ，计算在 s 的子序列中 t 出现的个数。

字符串的一个 子序列 是指，通过删除一些（也可以不删除）字符且不干扰剩余字符相对位置所组成的新字符串。（例如，“ACE” 是 “ABCDE” 的一个子序列，而 “AEC” 不是）

题目数据保证答案符合 32 位带符号整数范围。

示例 1：

> 输入：s = “rabbbit”, t = “rabbit”
> 输出：3
> 解释：
> 如下图所示, 有 3 种可以从 s 中得到 “rabbit” 的方案。
> (上箭头符号 ^ 表示选取的字母)
> rabbbit
> ^^^^ ^^
> rabbbit
> ^^ ^^^^
> rabbbit
> ^^^ ^^^

分析

定义dp[i][j]，表示s[:i]中t[:j]出现的次数
转移方程：

$$\begin{cases} dp[i][j] = dp[i-1][j-1] + dp[i-1][j], & s[i] == t[j] \\ dp[i][j] = dp[i-1][j], & s[i] != t[j] \end{cases}$$

$$\begin{cases} dp[i][j] = dp[i-1][j-1] + dp[i-1][j], & s[i] == t[j] \\ dp[i][j] = dp[i-1][j], & s[i] != t[j] \end{cases}$$

说明：dp[i − 1][j − 1] 和 dp[i − 1][j] 分别表示使用和不使用当前 s[i] 的次数。而 s[i]! = t[j] 时，不能使用 s[i]，所以只有一种情况。

初始状态：

$$dp[i][0] = 1$$

表示当 t 为空时在 s[: i] 中出现一次。

代码

```
class Solution {
    public int numDistinct(String s, String t) {
        int m = s.length(), n = t.length();
        int[][] dp = new int[m+1][n+1];
        for (int i = 0; i <= m; i++) {
            dp[i][0] = 1;
        }
        for (int i = 1; i <= m; i++) {
            for (int j = 1; j <= Math.min(i, n); j++) {
                if (s.charAt(i - 1) == t.charAt(j - 1)) {
```

120. 三角形最小路径和

题目描述

给定一个三角形 triangle ，找出自顶向下的最小路径和。

每一步只能移动到下一行中相邻的结点上。相邻的结点 在这里指的是 下标 与 上一层结点下标 相同或者等于 上一层结点下标 + 1 的两个结点。也就是说，如果正位于当前行的下标 i ，那么下一步可以移动到下一行的下标 i 或 i + 1 。

示例 1：

> 输入：triangle = [[2],[3,4],[6,5,7],[4,1,8,3]]
> 输出：11
> 解释：如下面简图所示：
>   **2**
>   **3** 4
> 6 **5** 7
> 4 **1** 8 3
> 自顶向下的最小路径和为 11（即，2 + 3 + 5 + 1 = 11）。

分析

1. DP。
   定义dp[i][j]，表示第i行的第j个位置的最小路径和。
   转移方程：

$$dp[i][j] = \min(dp[i-1][j], dp[i-1][j-1]) + triangle[i][j]$$

   初始状态：

$$dp[0][0] = triangle[0][0]$$

代码

```
class Solution {
    public int minimumTotal(List<List<Integer>> triangle) {
        int n = triangle.size();
        if (n == 0) {
            return -1;
```

```
        }
        int[][] dp = new int[n][n];
        dp[0][0] = triangle.get(0).get(0);
        for (int i = 1; i < n; i++) {
                // 针对每一行的第一个位置特殊处理
```

53. 最大子序和

题目描述

给定一个整数数组 nums ， 找到一个具有最大和的连续子数组（子数组最少包含一个元素），返回其最大和。

示例 1：

> 输入：nums = [-2,1,-3,4,-1,2,1,-5,4]
> 输出：6
> 解释：连续子数组 [4,-1,2,1] 的和最大，为 6 。

分析

1. DP.
   定义dp[i]为以第 i 个数结尾的「连续子数组的最大和」。
   转移方程：dp[i] = max(dp[i-1] + nums[i], nums[i])
   由于状态dp[i]只依赖前一个状态dp[i-1]，所以不用数组来存储状态，空间复杂度可以优化到O(1).

代码

```java
public class Solution {
    public int maxSubArray(int[] nums) {
        int n = nums.length;
        int res = nums[0], total = 0;
        for (int i = 0; i < n; i++) {
            total = Math.max(total + nums[i], nums[i]);
            res = Math.max(res, total);
        }
        return res;
    }
}
```

152. 乘积最大子数组

题目描述

给你一个整数数组 nums ， 请你找出数组中乘积最大的连续子数组（该子数组中至少包含一个数字），并返回该子数组所对应的乘积。

示例 1:

> 输入: [2,3,-2,4]
> 输出: 6
> 解释: 子数组 [2,3] 有最大乘积 6。

分析

1. DP
   考虑到负负得正的情况，所以需要维护最小和最大的状态。
   dp1[i]和dp2[i]分别表示以第 i 个数结尾的「最大乘积」和以第 i 个数结尾的「最小乘积」。
   转移方程：
   dp1[i] = max(dp1[i-1] * nums[i], dp2[i-1] * nums[i], nums[i])
   dp2[i] = min(dp1[i-1] * nums[i], dp2[i-1] * nums[i], nums[i])

代码

```java
class Solution {
    public int maxProduct(int[] nums) {
```

```
        int n = nums.length;
        int[] dp1 = new int[n];
        int[] dp2 = new int[n];
        dp1[0] = nums[0];
        dp2[0] = nums[0];
        int res = dp1[0];

        for (int i = 1; i < n; i++) {
```

由于dp1[i]和dp2[i]都只依赖于前一个状态，所以可以将空间复杂度优化到O(1).

```
public int maxProduct(int[] nums) {
        int n = nums.length;
        int minValue = nums[0], maxValue = nums[0], res = nums[0], tmp;
        for (int i = 1; i < n; i++) {
            tmp = maxValue;
            maxValue = Math.max(Math.max(maxValue * nums[i], minValue * nums[i]), nums[i]);
            minValue = Math.min(Math.min(tmp * nums[i], minValue * nums[i]), nums[i]);
            res = Math.max(res, maxValue);
        }
        return res;
    }
```

887. 鸡蛋掉落

题目描述

给你 k 枚相同的鸡蛋，并可以使用一栋从第 1 层到第 n 层共有 n 层楼的建筑。

已知存在楼层 f ，满足 0 <= f <= n ，任何从 高于 f 的楼层落下的鸡蛋都会碎，从 f 楼层或比它低的楼层落下的鸡蛋都不会破。

每次操作，你可以取一枚没有碎的鸡蛋并把它从任一楼层 x 扔下（满足 1 <= x <= n）。如果鸡蛋碎了，你就不能再次使用它。如果某枚鸡蛋扔下后没有摔碎，则可以在之后的操作中 重复使用 这枚鸡蛋。

请你计算并返回要确定 f 确切的值 的 最小操作次数 是多少?

示例 1:

> 输入：k = 1, n = 2
> 输出：2
> 解释：
> 鸡蛋从 1 楼掉落。如果它碎了，肯定能得出 f = 0 。
> 否则，鸡蛋从 2 楼掉落。如果它碎了，肯定能得出 f = 1 。
> 如果它没碎，那么肯定能得出 f = 2 。
> 因此，在最坏的情况下我们需要移动 2 次以确定 f 是多少。

分析

dp(k, n) 表示k个鸡蛋n层楼的最小值
转移方程:

$$dp(k,n) = \min_{1<=x<=n} \max\left(dp(k,n-x),\ dp(k-1,x-1)\right)$$

x 表示扔鸡蛋的楼层，要么鸡蛋碎了，转移到 $dp(k-1,x-1)$；要么鸡蛋没有碎，转移到 $dp(k,n-x)$。
初始状态:

$$dp(k,0) = 0$$
$$dp(1,n) = n$$

由于 $dp(k-1, x-1)$ 随 x 单调递增，$dp(k, n-x)$ 随 x 单调递减，要求两者最大值的最小值，就是求两个 函数 的交点附近的值，则可以用二分查询进行求解，将时间复杂度从 $O(kn^2)$ 优化到 $O(knlon)$.

参考

代码

```
class Solution {
    Map<Integer, Integer> mem;

    public int superEggDrop(int k, int n) {
        mem = new HashMap<>();
        return dp(k, n);
    }

    private int dp(int k, int n) {
        if (n == 0) {
```

198. 打家劫舍

题目描述

你是一个专业的小偷，计划偷窃沿街的房屋。每间房内都藏有一定的现金，影响你偷窃的唯一制约因素就是相邻的房屋装有相互连通的防盗系统，如果两间相邻的房屋在同一晚上被小偷闯入，系统会自动报警。

给定一个代表每个房屋存放金额的非负整数数组，计算你 不触动警报装置的情况下 ， 一夜之内能够偷窃到的最高金额。

示例 1：

> 输入：[1,2,3,1]
> 输出：4
> 解释：偷窃 1 号房屋 (金额 = 1) ，然后偷窃 3 号房屋 (金额 = 3)。
> 偷窃到的最高金额 = 1 + 3 = 4 。

分析

1. DP
   定义dp[i]，表示小偷经过第i个房间时的最大金额（不一定偷窃第i个房间）。所以在经过第i个房间时，可以选择偷或不偷这个房间，分别对应$dp[i-2]+nums[i]$和$dp[i-1]$。
   转移方程：

$$dp[i] = \max(dp[i-2] + nums[i], dp[i-1]$$

由于当前状态dp[i]仅依赖于前两个状态dp[i-2]和dp[i-1]，所以可以使用两个常量来进行状态的转移，从而将空间复杂度从O(n)降低为O(1)。

代码

```
public class Solution {
    public int rob(int[] nums) {
        int n = nums.length;
        int pre1 = 0, pre2 = 0, tmp;

        for (int i = 0; i < n; i++) {
            tmp = Math.max(pre1 + nums[i], pre2);
            pre1 = pre2;
            pre2 = tmp;
        }
        return Math.max(pre1, pre2);
    }
```

213. 打家劫舍 II

题目描述

你是一个专业的小偷，计划偷窃沿街的房屋，每间房内都藏有一定的现金。这个地方所有的房屋都 围成一圈 ，这意味着第一个房屋和最后一个房屋是紧挨着的。同时，相邻的房屋装有相互连通的防盗系统，如果两间相邻的房屋在同一晚上被小偷闯入，系统会自动报警 。

给定一个代表每个房屋存放金额的非负整数数组，计算你 在不触动警报装置的情况下 ，能够偷窃到的最高金额。

示例 1：

> 输入：nums = [2,3,2]
> 输出：3
> 解释：你不能先偷窃 1 号房屋（金额 = 2），然后偷窃 3 号房屋（金额 = 2），因为他们是相邻的。

分析

和上题的区别在于房间首位相连，也就是选择偷了第一个房间就不能偷最后一个房间；同理选择偷了最后一个房间就不能偷盗第一个房间。于是我们把这个问题拆解为两个问题，第一个问题偷盗的房间是$\text{nums}[1:n]$，第二个问题偷盗的房间是$\text{nums}[0:n-1]$。
转移方程和上题都一样。

代码

```
public class Solution {
    public int rob(int[] nums) {
        int n = nums.length;
        if (n == 1) {
            return nums[0];
        }
        return Math.max(getRes(nums, 0, n - 2), getRes(nums, 1, n - 1));
    }

    private int getRes(int[] nums, int i, int j) {
```

72. 编辑距离

题目描述

给你两个单词 word1 和 word2，请你计算出将 word1 转换成 word2 所使用的最少操作数 。

你可以对一个单词进行如下三种操作：

插入一个字符
删除一个字符
替换一个字符

示例 1：

> 输入：word1 = “horse”, word2 = “ros”
> 输出：3
> 解释：
> horse -> rorse (将 ‘h’ 替换为 ‘r’)
> rorse -> rose (删除 ‘r’)
> rose -> ros (删除 ‘e’)

分析

1. DP
   经典的DP问题。定义dp[i][j]，表示text1[:i]和text2[:j]的编辑距离。
   转移方程：

$$\begin{cases} dp[i][j] = dp[i-1][j-1] & text1[i] = text2[j] \\ dp[i][j] = \min(dp[i-1][j], dp[i][j-1], dp[i-1][j-1]) & text1[i] != text2[j] \end{cases}$$

初始状态：

$$\begin{cases} dp[0][j] = j + 1 \\ dp[i][0] = i + 1 \end{cases}$$

时间复杂度为O(mn)，空间复杂度为O(mn)

代码

```
class Solution {
    public int minDistance(String word1, String word2) {
        int m = word1.length(), n = word2.length();
        int[][] dp = new int[m+1][n+1];

        for (int i = 0; i <= m; i++) {
            for (int j = 0; j <= n; j++) {
                if (i == 0 || j == 0) {
                    dp[i][j] = Math.max(i, j);
                } else {
```


44. 通配符匹配

题目描述

给定一个字符串 (s) 和一个字符模式 §，实现一个支持 ‘?’ 和 ‘*’ 的通配符匹配。

‘?’ 可以匹配任何单个字符。
‘*’ 可以匹配任意字符串（包括空字符串）。
两个字符串完全匹配才算匹配成功。

说明:

s 可能为空，且只包含从 a-z 的小写字母。
p 可能为空，且只包含从 a-z 的小写字母，以及字符 ? 和 *。

示例 1:

> 输入:
> s = “aa”
> p = “a”
> 输出: false
> 解释: “a” 无法匹配 “aa” 整个字符串。

分析

1. DP
   定义dp[i][j]，表示s[:i]和p[:j]是否匹配。
   转移方程：

$$\begin{cases} dp[i][j] = dp[i-1][j-1], & s[i] == p[j] \\ dp[i][j] = dp[i-1][j-1], & p[j] == ? \\ dp[i][j] = dp[i][j-1] \mid\mid dp[i-1][j], & p[j] == * \end{cases}$$

第三种情况中，dp[i][j − 1]表示” *“匹配0次*，dp[i − 1][j]*表示*” “匹配多次。
第一种情况和第二种情况的转移方程相同，因此可以合并。
初始状态：

$$\begin{cases} dp[0][0] = true \\ dp[0][j] = dp[0][j-1], & p[j] == * \\ dp[i][0] = false \end{cases}$$

代码

```
class Solution {
    public boolean isMatch(String s, String p) {
        int n = s.length(), m = p.length();
        boolean[][] dp = new boolean[n + 1][m + 1];
                // 初始状态
        dp[0][0] = true;
        for (int j = 1; j <= m; j++) {
            if (p.charAt(j - 1) == '*') {
                dp[0][j] = dp[0][j-1];
            }
```

10. 正则表达式匹配

题目描述

给你一个字符串 s 和一个字符规律 p，请你来实现一个支持 ‘.’ 和 ‘*’ 的正则表达式匹配。

‘.’ 匹配任意单个字符
‘*’ 匹配零个或多个前面的那一个元素
所谓匹配，是要涵盖 整个 字符串 s的，而不是部分字符串。

示例 1：

> 输入：s = “aa” p = “a”
> 输出：false
> 解释：“a” 无法匹配 “aa” 整个字符串。

分析

这道题和上题「通配符匹配」的区别在于上题的”*“可以匹配0个或者任意个字符，而这题的”*“匹配0个或者任意个前面的元素。
还是DP
定义dp[i][j]表示s[:i]和p[:j]是否匹配。
转移方程：

$$\begin{cases} dp[i][j] = dp[i-1][j-1], & s[i] == p[j] \\ dp[i][j] = dp[i-1][j-1], & p[j] == "." \\ dp[i][j] = dp[i][j-2], & p[j] == "*" \\ dp[i][j] = dp[i-1][j], & p[j] == "*" \ \&\& \ (s[i] == p[j-1] \end{cases}$$

$$\begin{cases} dp[i][j] = dp[i-1][j-1], & s[i] == p[j] \\ dp[i][j] = dp[i-1][j-1], & p[j] == "." \\ dp[i][j] = dp[i][j-2], & p[j] == "*" \\ dp[i][j] = dp[i-1][j], & p[j] == "*" \ \&\& \ (s[i] == p[j-1] \ || \ p[j-1] == ".") \end{cases}$$

$$\begin{cases} dp[i][j] = dp[i-1][j-1], & s[i] == p[j] \\ dp[i][j] = dp[i-1][j-1], & p[j] == "." \\ dp[i][j] = dp[i][j-2], & p[j] == "*" \\ dp[i][j] = dp[i-1][j], & p[j] == "*" \ \&\& \ (s[i] == p[j-1] \ || \ p[j-1] == ".") \end{cases}$$

初始状态：

$$\begin{cases} dp[0][0] = true \\ dp[0][j] = dp[0][j-1], & p[j] == * \\ dp[i][0] = false \end{cases}$$

代码

```
class Solution {
    public boolean isMatch(String s, String p) {
        int n = s.length(), m = p.length();
        boolean[][] dp = new boolean[n + 1][m + 1];
                // 初始状态
        dp[0][0] = true;
        for (int j = 2; j <= m; j++) {
            if (p.charAt(j - 1) == '*') {
                dp[0][j] = dp[0][j - 2];
            }
```

2、区间DP

5. 最长回文子串

题目描述

给你一个字符串 s，找到 s 中最长的回文子串。

示例 1：

> 输入：s = “babad”
> 输出：“bab”
> 解释：“aba” 同样是符合题意的答案。

分析

定义dp[i][j]，表示s[i:j]是否是回文
转移方程：$dp[i][j] = dp[i+1][j-1], \quad s[i] == s[j]$
初始状态：

$$\begin{cases} dp[i][i] = true \\ dp[i][i+1], & s[i] == s[i+1] \end{cases}$$

$$\begin{cases} dp[i][i] = true \\ dp[i][i+1], & s[i] == s[i+1] \end{cases}$$

$$\begin{cases} dp[i][i] = true \\ dp[i][i+1], & s[i] == s[i+1] \end{cases}$$

时空复杂度都为O(n^2)

代码

```
class Solution {
    public String longestPalindrome(String s) {
        int n = s.length();
        boolean[][] dp = new boolean[n][n];

        for (int i = 0; i < n; i++) {
            dp[i][i] = true;
        }

        int res = 1;
```

另一种写法：

```
public int getLongestPalindrome(String A, int n) {
        boolean[][] dp = new boolean[n][n];
        for (int i = 0; i < n; i++) {
            Arrays.fill(dp[i], true);
        }

        int maxLen = 0;
        for (int i = n - 2; i >= 0; i--) {
            for (int j = i + 1; j < n; j++) {
                dp[i][j] = A.charAt(i) == A.charAt(j) && dp[i+1][j-1];
```

87. 扰乱字符串

题目描述

使用下面描述的 算法 可以扰乱字符串 s 得到字符串 t：

1. 如果字符串的长度为 1 ，算法停止
2. 如果字符串的长度 > 1 ，执行下述步骤：
   - 在一个随机下标处将字符串分割成两个非空的子字符串。即，如果已知字符串 s ，则可以将其分成两个子字符串 x 和 y ，且满足 s = x + y 。
   - 随机 决定是要「交换两个子字符串」还是要「保持这两个子字符串的顺序不变」。即，在执行这一步骤之后，s 可能是 s = x + y 或者 s = y + x 。
   - 在 x 和 y 这两个子字符串上继续从步骤 1 开始递归执行此算法。

给你两个 长度相等 的字符串 s1 和 s2，判断 s2 是否是 s1 的扰乱字符串。如果是，返回 true ；否则，返回 false 。

示例 1：

> 输入：s1 = “great”, s2 = “rgeat”
> 输出：true
> 解释：s1 上可能发生的一种情形是：
> “great” --> “gr/eat” // 在一个随机下标处分割得到两个子字符串
> “gr/eat” --> “gr/eat” // 随机决定：「保持这两个子字符串的顺序不变」
> “gr/eat” --> “g/r / e/at” // 在子字符串上递归执行此算法。两个子字符串分别在随机下标处进行一轮分割
> “g/r / e/at” --> “r/g / e/at” // 随机决定：第一组「交换两个子字符串」，第二组「保持这两个子字符串的顺序不变」
> “r/g / e/at” --> “r/g / e/ a/t” // 继续递归执行此算法，将 “at” 分割得到 “a/t”
> “r/g / e/ a/t” --> “r/g / e/ a/t” // 随机决定：「保持这两个子字符串的顺序不变」
> 算法终止，结果字符串和 s2 相同，都是 “rgeat”
> 这是一种能够扰乱 s1 得到 s2 的情形，可以认为 s2 是 s1 的扰乱字符串，返回 true

分析

区间DP
定义mem[i][j][len]，表示s1[i:j+len]和s2[j:j+len]是否可以通过扰乱得到。
在i~i+len的范围内寻找分割点，假如s1[i:j+len]被切分成s11和s12，s2[j:j+len]被切分为s21和s22，则分为交换顺序和不交换两种情况：

1. s11和s21、s12和s22可以通过扰乱得到；
2. s11和s22、s12和s21可以通过扰乱得到。

求解过程中，可以加上字符串中字符个数判断用来剪枝，如下面的checkFreq方法。

时间复杂度：
填满数组mem[i][j][len]需要O(n^3)的时间复杂度，对于每一个mem[i][j][len]状态，需要O(n)的时间复杂度来求解，因此总时间复杂度为O(n^4)。

代码

```
class Solution {
    private int[][][] mem;
    private String s1;
    private String s2;

    public boolean isScramble(String s1, String s2) {
        int n = s1.length();
        mem = new int[n][n][n+1];
        this.s1 = s1;
        this.s2 = s2;
```


312.戳气球

题目描述

有 n 个气球，编号为0 到 n - 1，每个气球上都标有一个数字，这些数字存在数组 nums 中。

现在要求你戳破所有的气球。戳破第 i 个气球，你可以获得 nums[i - 1] * nums[i] * nums[i + 1] 枚硬币。 这里的 i - 1 和 i + 1 代表和 i 相邻的两个气球的序号。如果 i - 1或 i + 1 超出了数组的边界，那么就当它是一个数字为 1 的气球。

求所能获得硬币的最大数量。

示例 1：

> 输入：nums = [3,1,5,8]
> 输出：167
> 解释：
> nums = [3,1,5,8] --> [3,5,8] --> [3,8] --> [8] --> []
> coins = 3*1*5 + 3*5*8 + 1*3*8 + 1*8*1 = 167

分析

区间DP
定义dp[i][j] 表示(i, j)内的位置全部填满气球能够得到的最多硬币数
转移方程

$$dp[i][j] = \max_{i<mid<j} nums[i] * nums[mid] * nums[j] + dp[i][mid] + dp[mid][j]$$

时间复杂度O(n^3)，空间复杂度O(n^2)

代码

```
class Solution {
    public int maxCoins(int[] nums) {
        int n = nums.length;
        int[] newNums = new int[n + 2];
        newNums[0] = newNums[n + 1] = 1;
        for (int i = 0; i < n; i++) {
            newNums[i+1] = nums[i];
        }
        int[][] dp = new int[n + 2][n + 2];
```

```
                                                                ]);
```


3、背包 DP

377. 组合总和 Ⅳ

题目描述

给你一个由 不同 整数组成的数组 nums ，和一个目标整数 target 。请你从 nums 中找出并返回总和为 target 的元素组合的个数。

题目数据保证答案符合 32 位整数范围。

示例 1：

> 输入：nums = [1,2,3], target = 4
> 输出：7
> 解释：
> 所有可能的组合为：
> (1, 1, 1, 1)
> (1, 1, 2)
> (1, 2, 1)
> (1, 3)
> (2, 1, 1)
> (2, 2)
> (3, 1)
> 请注意，顺序不同的序列被视作不同的组合。

提示：

- 1 <= nums.length <= 200
- 1 <= nums[i] <= 1000
- nums 中的所有元素 互不相同
- 1 <= target <= 1000

分析

1. 回溯。组合总和问题还有系列的三道题：「39. 组合总和」「40. 组合总和 II」「216. 组合总和 III」，这三道题都是用回溯解决的，因此这道题也很容易回溯解法。但是观察提示中给定的数据长度，回溯的时间复杂度是阶乘级别的，可想而知继续使用回溯肯定超时了。
2. 动态规划。这道题只需要求解组合的数量，而组合具体是什么不需要求解。
   定义dp[x]，表示组合之和为x的组合数，那么 $dp[x] = \sum_{num:nums} dp[x - num]$。这是一道类似的背包问题题目。即需要装价值为target的物品，有多少中不同的装法。
   时间复杂度：两次遍历，O(kn)，k为nums长度，n=target。相比回溯解法，时间上优化了太多。详细见下面代码。

代码

```
class Solution {
    public int combinationSum4(int[] nums, int target) {
        // dp[i] 表示和为i的种类数
        int n = nums.length;
        int[] dp = new int[target + 1];
        dp[0] = 1;

        for (int i = 1; i <= target; i++) {

            for (int num: nums) {
```

4、树形 DP

124. 二叉树中的最大路径和

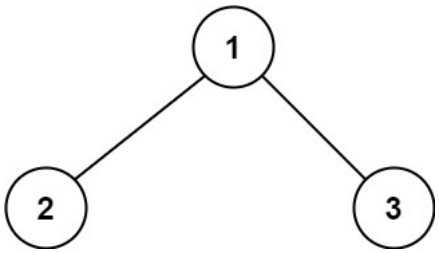
题目描述

路径 被定义为一条从树中任意节点出发，沿父节点-子节点连接，达到任意节点的序列。同一个节点在一条路径序列中 至多出现一次 。该路径 至少包含一个 节点，且不一定经过根节点。

路径和 是路径中各节点值的总和。

给你一个二叉树的根节点 root ，返回其 最大路径和 。

示例 1：



> 输入：root = [1,2,3]
> 输出：6
> 解释：最优路径是 2 -> 1 -> 3 ，路径和为 2 + 1 + 3 = 6

分析

树形dp，定义dp[x]，表示以x为根节点的子树到叶子节点的最大路径。
对于节点x，要求以x为根节点的树的最大路径和，可以拆解为x.left和x.right两个子问题left和right，
left = max(dp[x.left], 0)
right = max(dp[x.right], 0)
dp[x] = x.val + max(left, right)
每次递归返回的时候，都会把这个节点对应的值返回给上一级调用，这样可以不用存储中间过程。

代码

```
class Solution {
    private int res;

    public int maxPathSum(TreeNode root) {
        if (root == null) {
            return 0;
        }
        res = Integer.MIN_VALUE;
        dp(root);
        return res;
```

543. 二叉树的直径

题目描述

给定一棵二叉树，你需要计算它的直径长度。一棵二叉树的直径长度是任意两个结点路径长度中的最大值。这条路径可能穿过也可能不穿过根结点。

示例 :
给定二叉树

```
  1
 / \
2  3
/ \
4  5
```

返回 3, 它的长度是路径 [4,2,1,3] 或者 [5,2,1,3]。

分析

树形dp
定义dp[x]表示以x为根节点的子树的最大深度。
left = dp[x.left]
right = dp[x.right]
dp[x] = max(left, right) + 1

代码

```
class Solution {

    private int res;
    public int diameterOfBinaryTree(TreeNode root) {
        res = 0;
        dp(root);
        return res;
    }

    private int dp(TreeNode root) {
```


337. 打家劫舍 III

题目描述

在上次打劫完一条街道之后和一圈房屋后，小偷又发现了一个新的可行窃的地区。这个地区只有一个入口，我们称之为“根”。 除了“根”之外，每栋房子有且只有一个“父“房子与之相连。一番侦察之后，聪明的小偷意识到“这个地方的所有房屋的排列类似于一棵二叉树”。 如果两个直接相连的房子在同一天晚上被打劫，房屋将自动报警。

计算在不触动警报的情况下，小偷一晚能够盗取的最高金额。

示例 1:

> 输入: [3,2,3,null,3,null,1]
> 3
> / \
> 2 3
> \   \
> 3   1
> 输出: 7
> 解释: 小偷一晚能够盗取的最高金额 = 3 + 3 + 1 = 7.

分析

树形dp
定义dp[x]，表示以x为根节点的子树，选择或不选择x能够盗取的最大金额。(dp[x]的值是一个长度为2的数组，第一个元素的选择x的最大金额，第二个元素是不选择x的最大金额)
left = dp[x.left]
right = dp[x.right]
selected = x.val + left[1] + right[1]
notSelected = max(left[0], left[1]) + max(right[0], right[1])
dp[x] = [selected, notSelected]

代码

```
public class Solution {
    public int rob(TreeNode root) {
        int[] res = dp(root);
        return Math.max(res[0], res[1]);
    }

    private int[] dp(TreeNode root) {
        if (root == null) {
            return new int[]{0, 0};
        }
```

6、数位 DP

233. 数字 1 的个数

题目描述

给定一个整数 n，计算所有小于等于 n 的非负整数中数字 1 出现的个数。

示例 1：

> 输入：n = 13
> 输出：6

分析

计算n中每一位为1时的数字个数之和。
定义高位hight，低位low，位数digit

分三种情况：

- 该位为0时
  以2407为例，十位为1的数字个数为：
  0010~2319
  000~239
  总共为240个，即24*10
  high*digit
- 该位为1时
  以2417为例，十位为1的数字个数为：
  0010~2417
  000~247
  总共为248个，即24*10+7+1
  high*digit+low+1
- 该位为其他数时
  以2427为例，十位为1的数字个数为：
  0010~2419
  000~249
  总共为250个，即(24 + 1)*10
  (high+1)*digit

代码

```
class Solution {
    public int countDigitOne(int n) {
        int high = n / 10, low = 0, digit = 1, cur = (n / digit) % 10, res = 0;
        while (high != 0 || cur != 0) {
            if (cur == 0) {
                res += (high * digit);
            } else if (cur == 1) {
                res += (high * digit + low + 1);
            } else {
                res += (high + 1) * digit;
```

注意while的条件是「$\text{high}! = 0$ ｜ ｜ $\text{cur}! = 0$」，因为当high为0时，此时还没结束，还需要最后一次计算，直到cur也为0才结束。

62. 不同路径

题目描述

一个机器人　位于一个 m x n 网格的左上角 （起始点在下图中标记为 “Start” ）。

机器人每次只能向下或者向右移动一步。机器人试图达到网格的右下角（在下图中标记为 “Finish” ）。

问总共有多少条不同的路径?

示例 1：

> 输入：m = 3, n = 7
> 输出：28

分析

dp
转移方程：dp[i][j] = dp[i-1][j] + dp[i][j-1]
可以将空间进一步优化到O(n)

代码

```
class Solution {
    public int uniquePaths(int m, int n) {
        int[] dp = new int[n];
        Arrays.fill(dp, 1);

        for (int i = 1; i < m; i++) {
            for (int j = 1; j < n; j++) {
                dp[j] = dp[j-1] + dp[j];
            }
        }
        return dp[n - 1];
    }
}
```

63. 不同路径 II

题目描述

一个机器人位于一个 m x n 网格的左上角 （起始点在下图中标记为“Start” ）。

机器人每次只能向下或者向右移动一步。机器人试图达到网格的右下角（在下图中标记为“Finish”）。

现在考虑网格中有障碍物。那么从左上角到右下角将会有多少条不同的路径?

网格中的障碍物和空位置分别用 1 和 0 来表示。

示例 1：

> 输入：obstacleGrid = [[0,0,0],[0,1,0],[0,0,0]]
> 输出：2
> 解释：
> 3x3 网格的正中间有一个障碍物。
> 从左上角到右下角一共有 2 条不同的路径：
>
> 1. 向右 -> 向右 -> 向下 -> 向下
> 2. 向下 -> 向下 -> 向右 -> 向右

分析

在上题的基础上考虑障碍物的影响，如果有障碍，则不进行状态转移。同理也可以优化空间为O(n).

代码

```
class Solution {
    public int uniquePathsWithObstacles(int[][] obstacleGrid) {
        int m = obstacleGrid.length, n = obstacleGrid[0].length;
        int[] dp = new int[n];
        dp[0] = obstacleGrid[0][0] == 0 ? 1 : 0;


```

```
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                if (obstacleGrid[i][j] == 1) {
                    dp[i] = 0;
```


96. 不同的二叉搜索树

题目描述

给定一个整数 n，求以 1 ... n 为节点组成的二叉搜索树有多少种?

示例:

> 输入: 3
> 输出: 5
> 解释:
> 给定 n = 3, 一共有 5 种不同结构的二叉搜索树:
> 1 3 3 2 1
> \ / / / \ \
> 3 2 1 1 3 2
> / / \ \ \
> 2 1 2 3

分析

定义dp[n]，表示长度为n的序列能构成的不同二叉搜索树的个数。
转移方程：

$$dp[n] = \sum_{i=0}^{n} dp[i] * dp[n-i-1]$$

初始状态：

$$dp[0] = 1$$
$$dp[1] = 1$$

代码

```
class Solution {
    public int numTrees(int n) {
        int[] dp = new int[n + 1];
        dp[0] = 1;
        dp[1] = 1;
        for (int i = 2; i <= n; i++) {
            for (int j = 0; j < i; j++) {
                dp[i] += dp[j] * dp[i - j - 1];
            }
        }
        return dp[n];
    }
}
```

8、递推型 DP

70. 爬楼梯

题目描述

假设你正在爬楼梯。需要 n 阶你才能到达楼顶。

每次你可以爬 1 或 2 个台阶。你有多少种不同的方法可以爬到楼顶呢?

注意：给定 n 是一个正整数。

示例 1：

> 输入： 2
> 输出： 2
> 解释： 有两种方法可以爬到楼顶。
>
> 1. 1 阶 + 1 阶
> 2. 2 阶

分析

定义dp[i]，表示爬到第i阶的不同方法数。dp[i] = dp[i-1] + dp[i-2]。
用两个变量可以优化空间到O(1)。

代码

```
class Solution {
    public int climbStairs(int n) {
        if (n < 2) {
            return 1;
        }
        int a = 1, b = 2, tmp;
        for (int i = 3; i <= n; i++) {
            tmp = a + b;
            a = b;
            b = tmp;
        }
        return b;
    }
}
```

509. 斐波那契数

题目描述

斐波那契数，通常用 F(n) 表示，形成的序列称为 斐波那契数列 。该数列由 0 和 1 开始，后面的每一项数字都是前面两项数字的和。也就是：

F(0) = 0，F(1) = 1
F(n) = F(n - 1) + F(n - 2)，其中 n > 1
给你 n ，请计算 F(n) 。

示例 1：

> 输入：2
> 输出：1
> 解释：F(2) = F(1) + F(0) = 1 + 0 = 1

分析

和上题类似，不再赘述。

代码

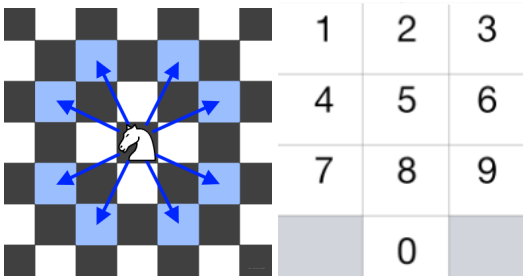
```
class Solution {
    public int fib(int n) {
        if (n == 0) {
            return 0;
        }
        int a = 0, b = 1, tmp;
        for (int i = 2; i <= n; i++) {
            tmp = a + b;
            a = b;
            b = tmp;
        }
        return b;

```

```
        }
    }
```

935. 骑士拨号器

题目描述

国际象棋中的骑士可以按下图所示进行移动：



这一次，我们将 “骑士” 放在电话拨号盘的任意数字键（如上图所示）上，接下来，骑士将会跳 N-1 步。每一步必须是从一个数字键跳到另一个数字键。

每当它落在一个键上（包括骑士的初始位置），都会拨出键所对应的数字，总共按下 N 位数字。

你能用这种方式拨出多少个不同的号码?

因为答案可能很大，所以输出答案模 10^9 + 7。

示例 1：

> 输入：1
> 输出：10

分析

定义dp[i][k]，表示在k位置走了i步的不用号码数。

$$dp[i][k] = \sum_{j \in moves} dp[i-1][j]$$

其中 moves 是定义好的可转移的状态对。
时间复杂度：O(n * 10 * k)，k最大为3，最小为0，所以时间复杂度为O(n)

代码

```
class Solution {
    public int knightDialer(int n) {
        // dp[i][j] = \sum_{k=0}^{9} dp[i-1][k]
        int mod = 1000000000 + 7;
        int[][] dp = new int[n][10];
        for (int k = 0; k < 10; k++) {
            dp[0][k] = 1;
        }

        int[][] moves = new int[][]{
```