算法学习之路--动态规划篇之基础篇

一、动态规划理论基础

什么是动态规划

动态规划（Dynamic Programming)，简称DP

如果某一问题有很多重叠子问题，使用动态规划是最有效的 => 所以动态规划中每一个状态是由**上一个状态推导**出来的（这一点区分于贪心，贪心没有状态推导，而是从**局部直接选最优**）

动态规划解题步骤

动归题目，状态转移公式（递推公式）是很重要，但动归不仅仅只有递推公式

动归掌握五部曲

- 确定dp数组（dp table）以及下标的含义
- 确定递推公式
- dp数组如何初始化
- 确定遍历顺序
- 举例推导dp数组

> 为什么要先确定递推公式，然后再考虑初始化?
>
> 因为一些情况是递推公式决定了dp数组要如何初始化

动态规划应该如何debug

写动归题目，代码出问题很正常!

- **找问题最好方式就是把dp数组打印出来，看看究竟是不是按照自己的思路推导的**
- **做动归题目，写代码之前一定要把状态转移在dp数组上的具体情况模拟一遍，心中有数，确定最后推出的是想要的结果**

=> 然后再写代码，如果代码没通过就打印dp数组，看看和自己预先推导的哪里不一样

- 如果打印出来和自己预先模拟推导的是一样的，那么就是自己的递推公式、初始化或者遍历顺序有问题
- 如果打印出来和自己预先模拟推导的不一样，那就是代码实现细节有问题

二、典型例题解析

509、斐波那契数

题目

题目链接：**https://leetcode.cn/problems/fibonacci-number/**

```
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

示例 2：
输入：3
输出：2
解释：F(3) = F(2) + F(1) = 1 + 1 = 2

示例 3：
输入：4
输出：3
解释：F(4) = F(3) + F(2) = 2 + 1 = 3

提示：

- 0 <= n <= 30
```

思路

动态规划

动归五部曲：

这里我们需要一个一维数组来保存递推的解雇

1. 确定dp数组以及下标的含义
    - `dp[i]` 定义：第i个数的斐波那契数值是 `dp[i]`
2. 确定递推关系
    - 状态转移方程（来自于题目）： `dp[i] = dp[i - 1] + dp[i - 2]`
3. dp数组初始化

```
// 题目也将如何初始化直接给出了
dp[0] = 0;
dp[1] = 1;
```

4. 确定遍历顺序

从递推关系 `dp[i] = dp[i - 1] + dp[i - 2]` 可以看出， `dp[i]` 是依赖 `dp[i - 1]` 和 `dp[i - 2]`，那么遍历的顺序一定是从前往后遍历的

5. 举例推导dp数组

   根据递推公式 `dp[i] = dp[i - 1] + dp[i - 2]`，当N=10的时候，dp数组应该是： `0 1 1 2 3 5 8 13 21 34 55`

   如果代码写出来发现结果不对，就把dp数组打印出来看看和我们推导的数列是否一致

实现

Java代码：

```java
// 时间复杂度：O(n)
// 空间复杂度：O(n)
class Solution {
    public int fib(int n) {
        if(n <= 1) return n;
        int[] dp = new int[n + 1];
        dp[0] = 0;
        dp[1] = 1;
        for(int i = 2;i <= n;++i) {
            dp[i] = dp[i - 1] + dp[i - 2];
        }
        return dp[n];
    }
}
```

=> 我们只需要维护两个数值就可以了，不需要记录整个序列

```java
// 时间复杂度：O(n)
// 空间复杂度：O(1)
class Solution {
    public int fib(int n) {
        if(n <= 1) return n;
        int[] dp = new int[2];
        dp[0] = 0;
        dp[1] = 1;
        for(int i = 2;i <= n;++i) {
            int sum = dp[0] + dp[1];
            dp[0] = dp[1];
            dp[1] = sum;
        }
        return dp[1];
    }
}
```

补充

递归解法

```
// 时间复杂度：O(2^n)
// 空间复杂度：O(n)
class Solution {
    public int fib(int n) {
        if(n < 2) return n;
        return fib(n - 1) + fib(n - 2);
    }
}
```

70、爬楼梯

题目

题目链接：**https://leetcode.cn/problems/climbing-stairs/**

```
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 阶 + 1 阶
        2 阶

示例 2：
输入： 3
输出： 3
解释： 有三种方法可以爬到楼顶。
        1 阶 + 1 阶 + 1 阶
        1 阶 + 2 阶
        2 阶 + 1 阶
```

思路

模拟一遍过程

爬到第一层楼梯有一种方法，爬到第二层楼梯有两种方法
那么第一层楼梯再跨两步就到第三层，第二层楼梯再跨一步就到第三层 => 第三层楼梯的状态可以由到第二层楼梯和到第一层楼梯状态推导出来 => 动态规划

动态规划五部曲

定义一个一维数组来记录不同楼层的状态

- 确定dp数组以及下标的含义
  - `dp[i]`：爬到第i层楼梯，有 `dp[i]` 种方法
- 确定递推公式

  从dp[i]的定义可以看出，dp[i]可以由两个方向推出
  - 首先是dp[i - 1]，上i-1层楼梯，有dp[i-1]种方法，那么再跳一个台阶就是dp[i]
  - 还有就是dp[i - 2]，上i-2层楼梯，有dp[i-2]种方法，那么再跳一个台阶就是dp[i]

  => `dp[i] = dp[i - 1] + dp[i - 2]`
- dp数组初始化

  回顾一下dp[i]的定义：**爬到第i层楼梯，有dp[i]中方法**

  那么i为0，dp[i]应该是多少呢，这个可以有很多解释，但都基本是直接奔着答案去解释的。

  例如强行安慰自己爬到第0层，也有一种方法，什么都不做也就是一种方法即：dp[0] = 1，相当于直接站在楼顶。 但总有点牵强的成分。

  那还这么理解呢：我就认为跑到第0层，方法就是0啊，一步只能走一个台阶或者两个台阶，然而楼层是0，直接站楼顶上了，就是不用方法，dp[0]就应该是0。

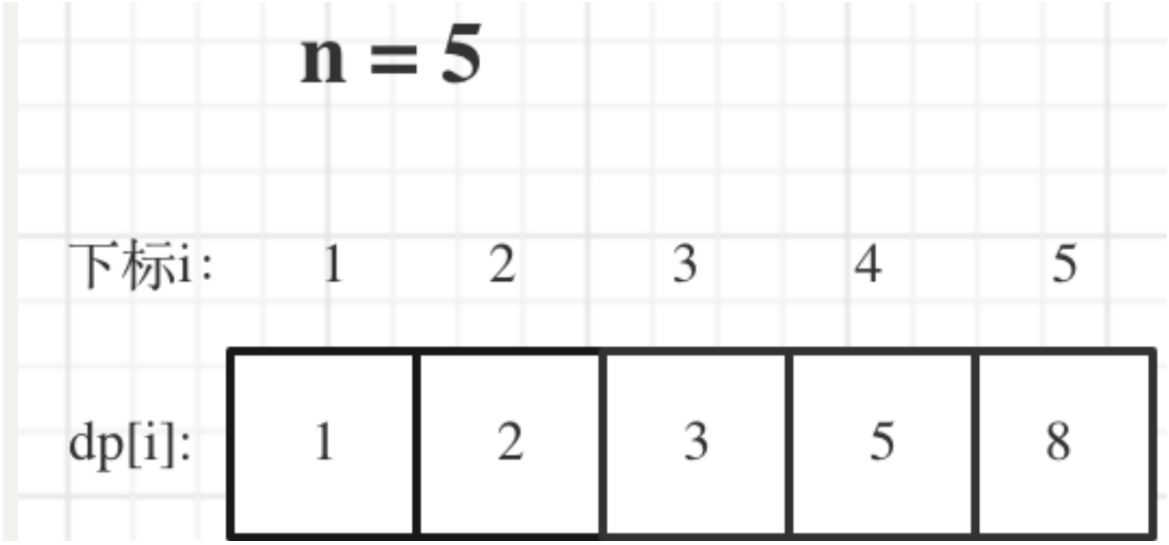
  其实这么争论下去没有意义，大部分解释说**dp[0]**应该为**1**的理由其实是因为**dp[0]=1**的话在递推的过程中**i**从**2**开始遍历本题就能过，然后就往结果上靠去解释**dp[0] = 1**。

  从dp数组定义的角度上来说，dp[0] = 0 也能说得通。

  需要注意的是：题目中说了n是一个正整数，题目根本就没说n有为0的情况 => **本题不应该讨论dp[0]的初始化
  - `dp[1] = 1,dp[2] = 2`
- 确定遍历顺序

  从递推公式 `dp[i] = dp[i - 1] + dp[i - 2]` 可以看出，遍历顺序一定是从前往后遍历的
- 举例推导dp数组

当n=5时，dp table（dp数组）应该是这样的



如果代码出问题了，就把dp table打印出来，看看究竟是不是和自己推导的一样

> 仔细观察，这就是斐波那契数列，唯一的区别就是没有讨论dp[0]应该是什么，因为dp[0]在本题没有意义

实现

Java代码：

```java
// 时间复杂度：O(n)
// 空间复杂度：O(n)
class Solution {
    public int climbStairs(int n) {
        if(n <= 1) return n;
        int[] dp = new int[n + 1];
        dp[1] = 1;
        dp[2] = 2;
        for(int i = 3;i <= n;++i) {
            dp[i] = dp[i - 1] + dp[i - 2];
        }
        return dp[n];
    }
}
```

当然代码可以优化一下空间复杂度

```java
// 时间复杂度：O(n)
// 空间复杂度：O(1)
class Solution {
    public int climbStairs(int n) {
        if(n <= 1) return n;
        int[] dp = new int[3];
        dp[1] = 1;
```

```
        dp[2] = 2;
        for(int i = 3;i <= n;++i) {
            int sum = dp[1] + dp[2];
            dp[1] = dp[2];
            dp[2] = sum;
        }
        return dp[2];
    }
}
```

746、使用最小花费爬楼梯

题目

题目链接：**https://leetcode-cn.com/problems/min-cost-climbing-stairs/**

```
数组的每个下标作为一个阶梯，第 i 个阶梯对应着一个非负数的体力花费值cost[i]（下标从0 开始）。每当你爬上一个阶梯你都要花费对应的体力值，一旦支付了相应的体力值，你就可以选择向上爬一个阶梯或者爬两个阶梯。请你找出达到楼层顶部的最低花费。在开始时，你可以选择从下标为 0 或 1 的元素作为初始阶梯。

示例 1：
输入：cost = [10, 15, 20]
输出：15
解释：最低花费是从 cost[1] 开始，然后走两步即可到阶梯顶，一共花费 15 。

示例 2：
输入：cost = [1, 100, 1, 1, 1, 100, 1, 1, 100, 1]
输出：6
解释：最低花费方式是从 cost[0] 开始，逐个经过那些 1 ，跳过 cost[3] ，一共花费 6 。

提示：

- cost 的长度范围是 [2, 1000]。
- cost[i] 将会是一个整型数据，范围为 [0, 999]
```

思路

注意题目描述：每当你爬上一个阶梯你都要花费对应的体力值，一旦支付了相应的体力值，你就可以选择向上爬一个阶梯或者爬两个阶梯

1. 确认dp数组以及下标的含义、

   使用一个一维数组dp[i]来记录状态：

   `dp[i]` => 到达第i个台阶所花费的最少体力为dp[i]（注意这里认为是第一步一定是要花费）

2. 确定递推公式

   可以有两个途径得到 `dp[i]` ，一个是 `dp[i - 1]` ，一个是 `dp[i - 2]`

选择最小的，所以 `dp[i] = Math.min(dp[i - 1],dp[i - 2]) + cost[i]`

> 注意这里为什么是加 `cost[i]` ，而不是 `cost[i-1]` , `cost[i-2]` 之类的，因为题目中说了：每当你爬上一个阶梯你都要花费对应的体力值

3. dp数组初始化

   根据dp数组的定义，dp数组初始化其实是比较难的，因为不可能初始化为第i台阶所花费的最少体力

   回归递归公式，**dp[i]由dp[i-1]，dp[i-2]推出**，既然初始化所有的dp[i]是不可能的， 那么**只初始化dp[0]和dp[1]**就够了，其他的最终都是dp[0]、dp[1]推出

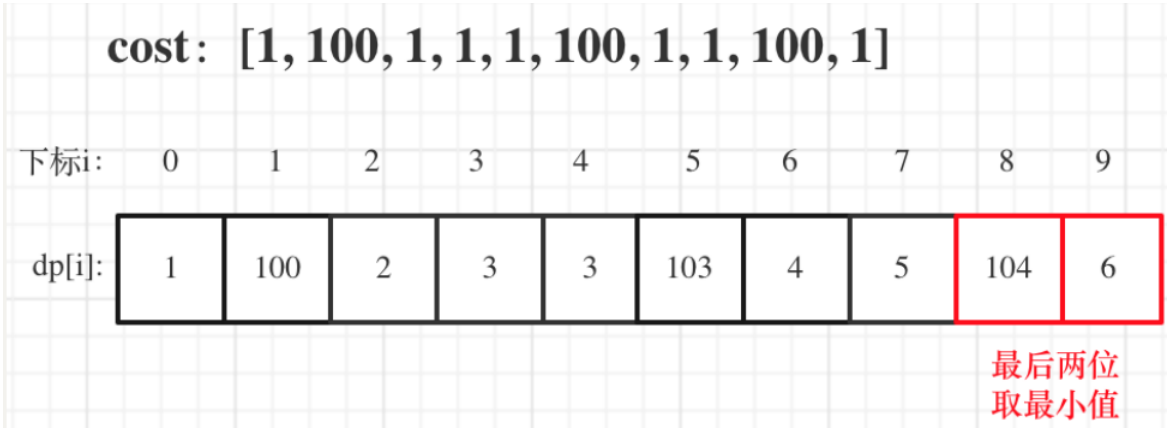
```
int[] dp = new int[cost.length]
dp[0] = cost[0];
dp[1] = cost[1];
```

4. 确定遍历顺序

   模拟台阶，而且dp[i]又dp[i-1]dp[i-2]推出，所以是**从前到后遍历cost数组**就可以了。

5. 举例推导dp数组

   拿示例2：cost = [1, 100, 1, 1, 1, 100, 1, 1, 100, 1] ，来模拟一下dp数组的状态变化



实现

Java代码

```
// 时间复杂度：O(n)
// 空间复杂度：O(n)
class Solution {
    public int minCostClimbingStairs(int[] cost) {
        int[] dp = new int[cost.length];
        dp[0] = cost[0];
        dp[1] = cost[1];
        for(int i = 2;i < cost.length;++i) {
            dp[i] = Math.min(dp[i - 1],dp[i - 2]) + cost[i];
        }
        // 注意最后一步理解为不用花费
```

```
        return Math.min(dp[cost.length - 1],dp[cost.length - 2]);
    }
}
```

扩展

定义 dp[i] 为:第一步是不花费体力，最后一步是花费体力的

Java代码：

```
class Solution {
    public int minCostClimbingStairs(int[] cost) {
        int[] dp = new int[cost.length + 1];
        dp[0] = 0;
        dp[1] = 0;
        for(int i = 2;i <= cost.length;++i) {
            dp[i] = Math.min(dp[i - 1] + cost[i - 1],dp[i - 2] + cost[i - 2]);
        }
        return dp[cost.length];
    }
}
```

62、不同路径

题目

题目链接：**https://leetcode-cn.com/problems/unique-paths/**

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?

示例一：

输入：m = 3, n = 7
输出：28

示例二：
输入：m = 2, n = 3
输出：3

提示：

- 1 <= m, n <= 100
- 题目数据保证答案小于等于 2 * 10^9

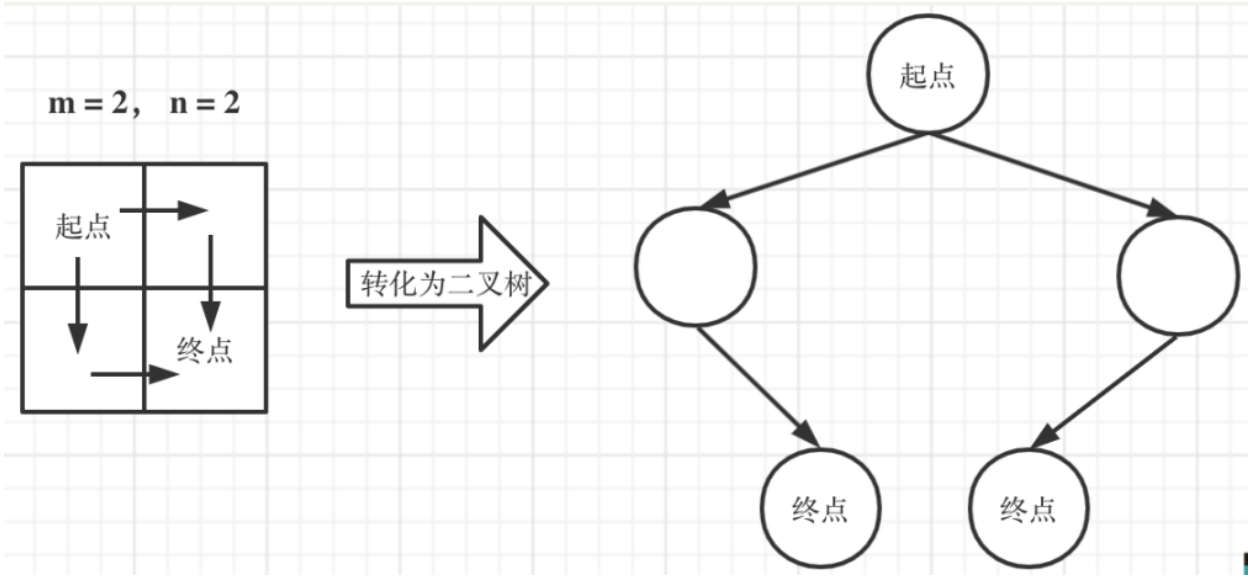
思路

深度优先搜索

题目最直观的想法就是用图论里的**深搜**，来枚举出来有多少种路径

注意题目中说机器人每次只能向下或者向右移动一步，那么其实机器人**走过的路径可以抽象为一颗二叉树，而叶子节点就是终点**

举例：



此时问题就可以转化为**求二叉树叶子节点的个数**

动态规划

机器人从（0,0）位置出发，到（m - 1，n - 1）

动态规划五部曲：

1. 确定dp数组（dp table）以及下标的含义

   `dp[i][j]`：表示从`(0,0)`出发，到`(i,j)`有`dp[i][j]`条不同的路径

2. 确定递推公式

   要求`dp[i][j]`，只能有两个方向来推导出来，即`dp[i - 1][j]`和`dp[i][j - 1]`。
   回顾一下`dp[i - 1][j]`表示什么，是从(0, 0)的位置到(i - 1, j)有几条路径，`dp[i][j - 1]`同理。
   那么很自然，`dp[i][j] = dp[i - 1][j] + dp[i][j - 1]`，因为`dp[i][j]`只有这两

个方向过来

3. dp数组初始化

   `dp[i][0]` 一定都是1，因为从(0, 0)的位置到(i, 0)的路径只有一条，那么 `dp[0][j]` 也同理

```
// 初始化代码
for (int i = 0; i < m; i++) dp[i][0] = 1;
for (int j = 0; j < n; j++) dp[0][j] = 1;
```

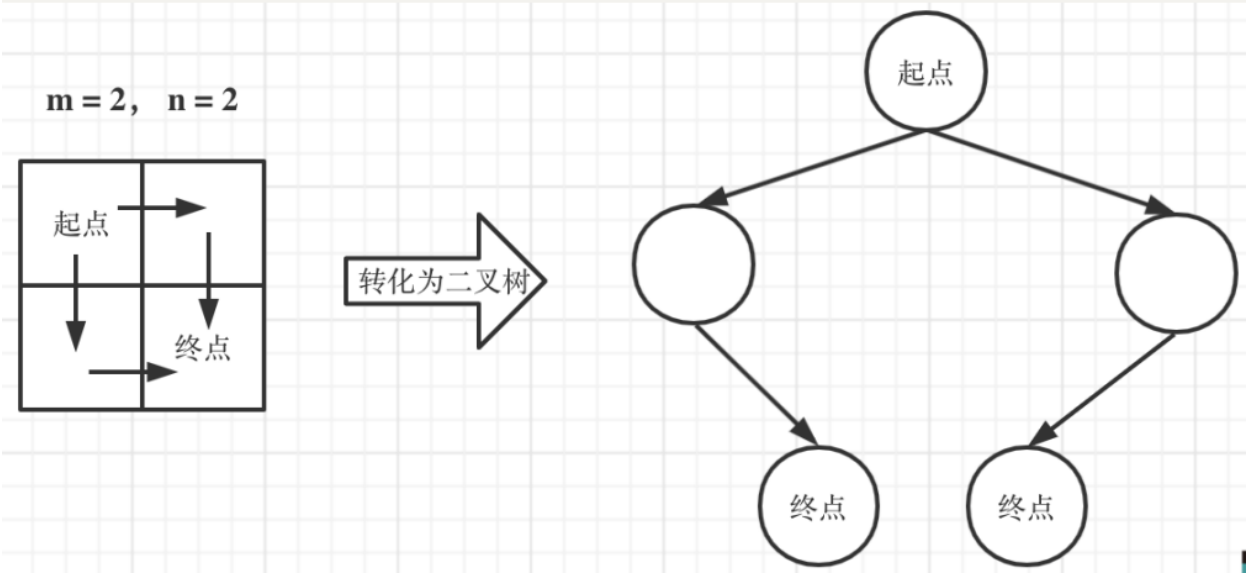
4. 确定遍历顺序

   观察递推公式：`dp[i][j] = dp[i - 1][j] + dp[i][j - 1]`，`dp[i][j]` 都是从其上方和左方推导而来，那么从左到右一层一层遍历即可

5. 举例推导dp数组

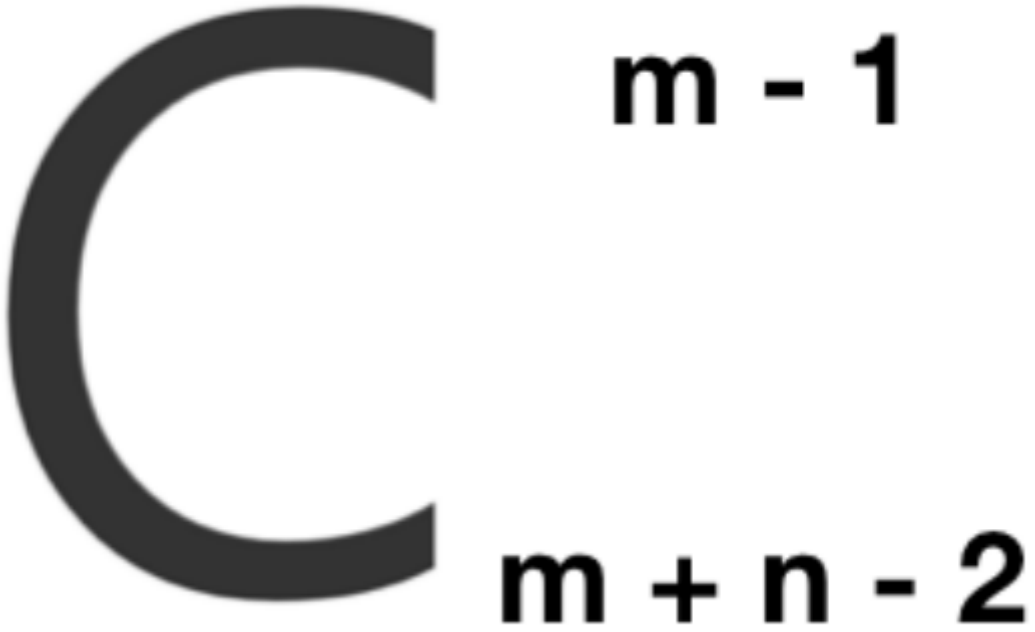
数论

在这个图中，可以看出一共m，n的话，无论怎么走，走到终点都需要 m + n - 2 步



在这m + n - 2 步中，一定有 m - 1 步是要向下走的，不用管什么时候向下走

走法数：



求组合的时候，要**防止两个int相乘溢出**！ 所以不能把算式的分子都算出来，分母都算出来再做除法

实现

深度优先搜索

Java代码：

```
class Solution {
    public int uniquePaths(int m, int n) {
        return dfs(1, 1, m, n);
    }
    public int dfs(int i, int j, int m, int n) {
        if (i > m || j > n) return 0;  // 越界
        if (i == m && j == n) return 1;  // 找到了一种方法，相当于找到了叶子节点
        return dfs(i + 1, j, m, n) + dfs(i, j + 1, m, n);
    }
}
// ===> 超时
/*
 分析一下时间复杂度，这个深搜的算法，其实就是要遍历整个二叉树。这颗树的深度其实就是m+n-1（深度按从1开始计算）。那二叉树的节点个数就是 2^(m + n - 1) - 1。可以理解深搜的算法就是遍历了整个满二叉树（其实没有遍历整个满二叉树，只是近似而已）
 => 上面深搜代码的时间复杂度为O(2^(m + n - 1) - 1)，可以看出，这是指数级别的时间复杂度，是非常大的。
*/
```

动态规划

Java代码

```java
// 时间复杂度：O(m * n)
// 空间复杂度：O(m * n)
class Solution {
    public int uniquePaths(int m, int n) {
        int[][] dp = new int[m][n];
        for(int i = 0;i < m;++i) dp[i][0] = 1;
        for(int j = 0;j < n;++j) dp[0][j] = 1;
        for(int i = 1;i < m;++i) {
            for(int j = 1;j < n;++j) {
                dp[i][j] = dp[i - 1][j] + dp[i][j - 1];
            }
        }
        return dp[m - 1][n - 1];
    }
}
```

数论

Java代码

```java
// 时间复杂度：O(m )
// 空间复杂度：O(1)
class Solution {
    public int uniquePaths(int m, int n) {
        long numerator = 1;// 分子
        int denominator = m - 1; // 分母
        int count = m - 1;
        int t = m + n - 2;
        while(count-- > 0) {
            numerator *= (t--);
            while(denominator != 0 && numerator % denominator == 0) {
                numerator /= denominator;
                denominator--;
            }
        }
        return (int)numerator;
    }
}
```

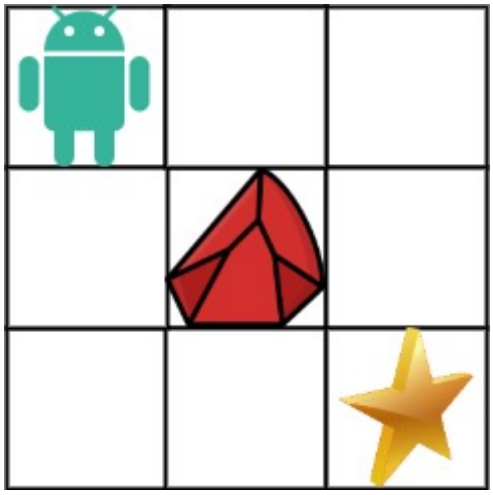
63、不同路径II

题目

题目链接：**https://leetcode-cn.com/problems/unique-paths-ii/**

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?

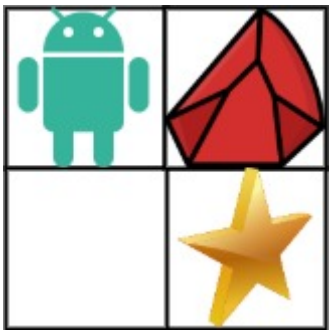
示例1：



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
1.  2.  向下 -> 向下 -> 向右 -> 向右
```

示例2：



输入：obstacleGrid = [[0,1],[0,0]]
输出：1

提示:

- m == obstacleGrid.length
- n == obstacleGrid[i].length
- 1 <= m, n <= 100

- obstacleGrid[i][j] 为 0 或 1

思路

62、不同路径我们已经详细分析了没有障碍的情况，有障碍的话，其实就是标记对应的dp table（dp数组）保持初始值(0)就可以了

动归五部曲

1. 确定dp数组（dp table）以及下标的含义
   `dp[i][j]`：表示从`(0,0)`出发，到`(i, j)`有`dp[i][j]`条不同的路径
2. 确定递推公式
   递推公式和**62、不同路径**一样，`dp[i][j] = dp[i - 1][j] + dp[i][j - 1]`。但这里需要注意一点，因为有了障碍，`(i, j)`如果就是障碍的话应该就保持初始状态（初始状态为0）

```
if (obstacleGrid[i][j] == 0) { // 当(i, j)没有障碍的时候，再推导dp[i][j]
 dp[i][j] = dp[i - 1][j] + dp[i][j - 1];
}
```

3. dp数组初始化
   在**62、不同路径**不同路径中我们给出如下的初始化：

```
// 初始化代码
for (int i = 0; i < m; i++) dp[i][0] = 1;
for (int j = 0; j < n; j++) dp[0][j] = 1;
```

因为从(0, 0)的位置到(i, 0)的路径只有一条，所以`dp[i][0]`一定为1，`dp[0][j]`也同理。
但如果(i, 0) 这条边有了障碍之后，障碍之后（包括障碍）都是走不到的位置了，所以障碍之后的`dp[i][0]`应该还是初始值0

下标(0, j)的初始化情况同理:

```
// 初始化代码
for (int i = 0; i < m && obstacleGrid[i][0] == 0; i++) dp[i][0] = 1;
for (int j = 0; j < n && obstacleGrid[0][j] == 0; j++) dp[0][j] = 1;
```

注意代码里for循环的终止条件，一旦遇到`obstacleGrid[i][0] == 1`的情况就停止`dp[i][0]`的 赋值1的操作，`dp[0][j]`同理

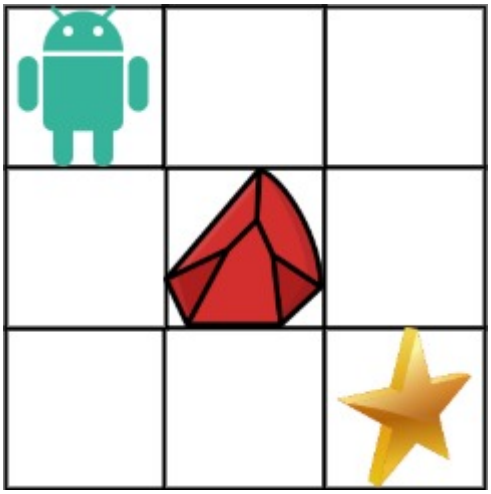
4. 确定遍历顺序

   从递归公式 `dp[i][j] = dp[i - 1][j] + dp[i][j - 1]` 中可以看出，一定是从左到右一层一层遍历，这样保证推导 `dp[i][j]` 的时候，`dp[i - 1][j]` 和 `dp[i][j - 1]` 一定是有数值

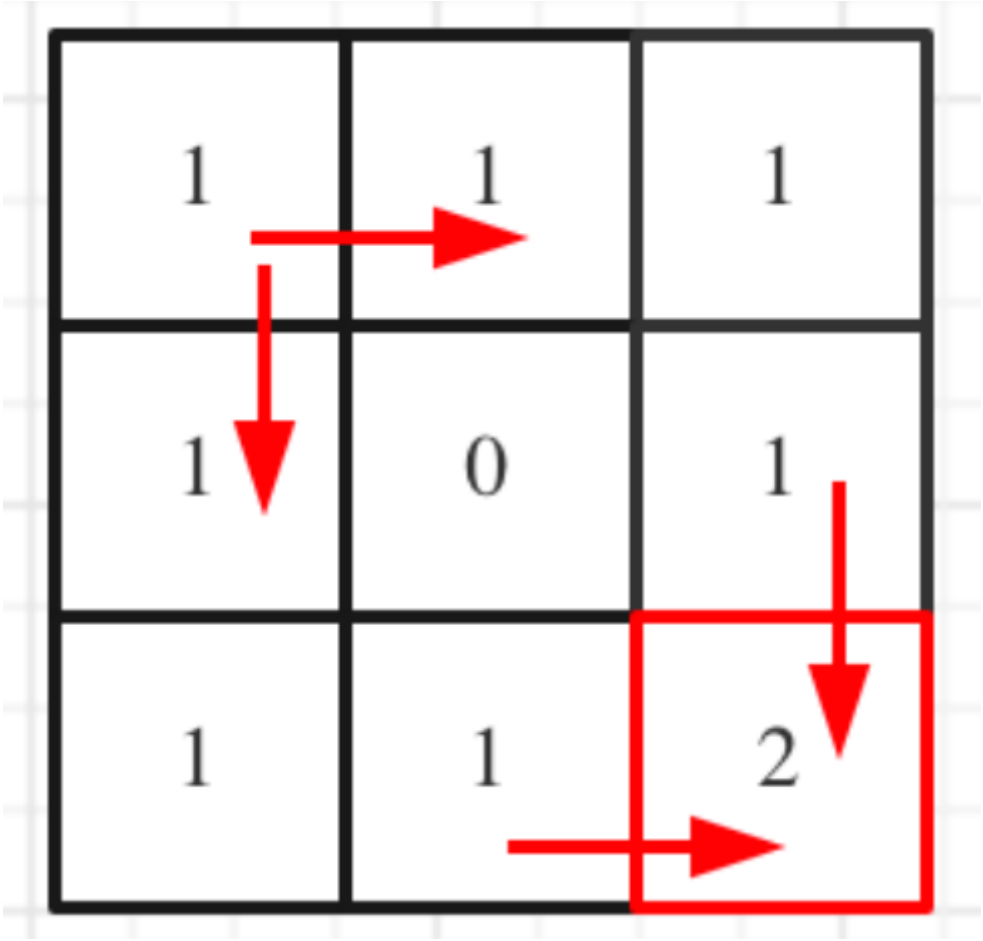
```
for (int i = 1; i < m; i++) {
    for (int j = 1; j < n; j++) {
        if (obstacleGrid[i][j] == 1) continue;
        dp[i][j] = dp[i - 1][j] + dp[i][j - 1];
    }
}
```

5. 举例推导dp数组

   拿示例1来举例：



   对应的dp table 如图：



实现

Java代码

```java
// 时间复杂度O(n * m) n m 分别为obstacleGrid 长度和宽度
// 空间复杂度O(n * m)
class Solution {
    public int uniquePathsWithObstacles(int[][] obstacleGrid) {
        int m = obstacleGrid.length;
        int n = obstacleGrid[0].length;

        int[][] dp = new int[m][n];
        // 初始化
        for(int j = 0;j < n && obstacleGrid[0][j] == 0;++j) dp[0][j] = 1;
        for(int i = 0;i < m && obstacleGrid[i][0] == 0;++i) dp[i][0] = 1;

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

343、整数拆分

题目

题目链接：**https://leetcode.cn/problems/integer-break/**

给定一个正整数 n，将其拆分为至少两个正整数的和，并使这些整数的乘积最大化。 返回你可以获得的最大乘积。

示例1:
输入: 2
输出: 1
解释: 2 = 1 + 1, 1 × 1 = 1。

示例 2:
输入: 10
输出: 36
解释: 10 = 3 + 3 + 4, 3 × 3 × 4 = 36。

提示:

- 2 <= n <= 58

思路

动规五部曲：

1. 确定dp数组(dp table)以及下标的含义
   `dp[i]`：分析数字i，可以得到最大乘积为 `dp[i]`
2. 确定递推公式
   考虑 `dp[i]` 的最大乘积如何得到?
   其实可以从1 遍历到 j，然后存在两种渠道得到 `dp[i]`
   - `j * (i - j)`
   - `j * dp[i - j]` => 相当于是拆分 `i - j`

   > j 从1 开始遍历，拆分 j 的情况在遍历 j 的过程中已经计算过了 => j 无需拆分

   => `dp[i] = Math.max(dp[i],Math.max((i - j) * j,dp[i - j] * j))`

3. dp的初始化

   > 有的题解里会给出 `dp[0] = 1，dp[1] = 1` 的初始化，但解释比较牵强，主要还是因为这么初始 化可以把题目过了。 严格从**dp[i]的定义**来说，**dp[0] dp[1] 就不应该初始化**，也就是没有意义的数值。 拆分0和拆分1的最大乘积是多少？ 这是无解的。

=> 初始化：dp[2] = 1

4. 确定遍历顺序

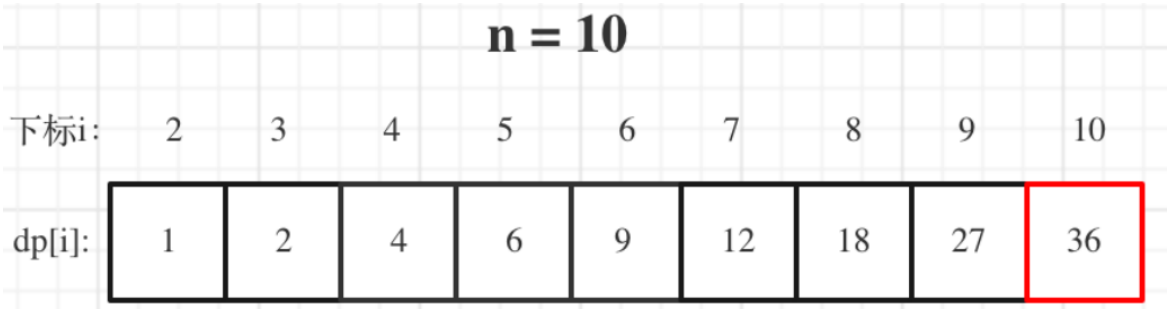
   根据递推公式： `dp[i] = Math.max(dp[i],Math.max((i - j) * j,dp[i - j] * j))`

   `dp[i]` 是依靠 `dp[i - j]` 的状态，所以遍历 i 一定是从前向后遍历，先有 `dp[i - j]` 再有 `dp[i]` 。 枚举 j 的时候，是从1开始的。i 是从 3 开始，这样 `dp[i - j]` 就是 `dp[2]` 正好可以通过我们初始化 的数值求出来

```
for (int i = 3; i <= n ; i++) {
     for (int j = 1; j < i - 1; j++) {
        dp[i] = Math.max(dp[i], max((i - j) * j, dp[i - j] * j));
     }
}
```

5. 举例推导dp数组

   当 `n = 10` 时，dp数组：



实现

Java代码

```
// 时间复杂度：O(n^2)
// 空间复杂度：O(n)
class Solution {
    public int integerBreak(int n) {
        int[] dp = new int[n + 1];
        dp[2] = 1;
        for(int i = 3;i <= n;++i) {
            for(int j = 1;j < i - 1;++j) {
                dp[i] = Math.max(dp[i],Math.max((i - j) * j,dp[i - j] * j));
            }
        }
        return dp[n];
    }
}
```

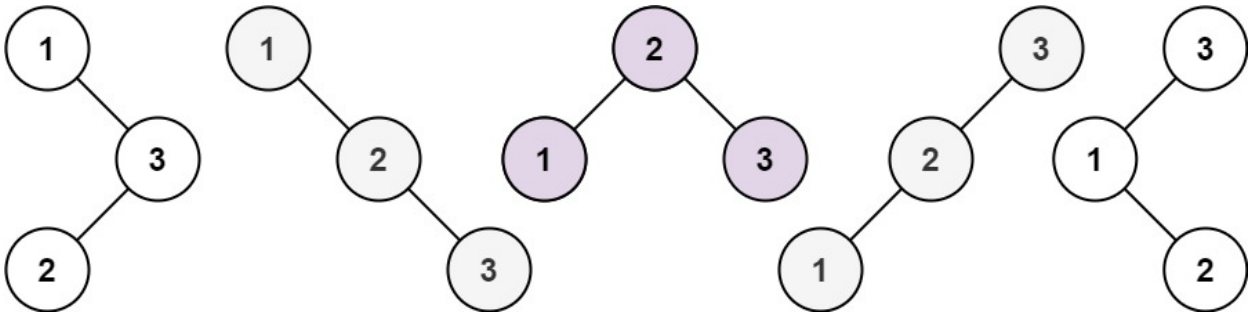
96 不同的二叉搜索树

题目

题目链接：**https://leetcode.cn/problems/unique-binary-search-trees/**

给你一个整数 `n` ，求恰由 `n` 个节点组成且节点值从 `1` 到 `n` 互不相同的 二叉搜索树 有多少种？返回满足题意的二叉搜索树的种数

示例1：



输入：n = 3
输出：5

示例2：
输入：n = 1
输出：1

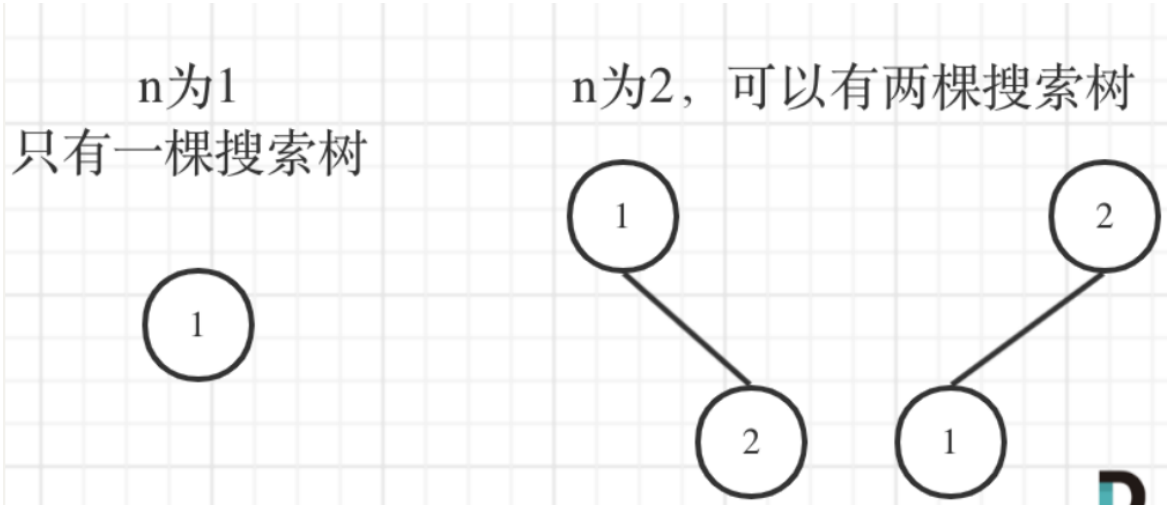
提示：

- 1 <= n <= 19

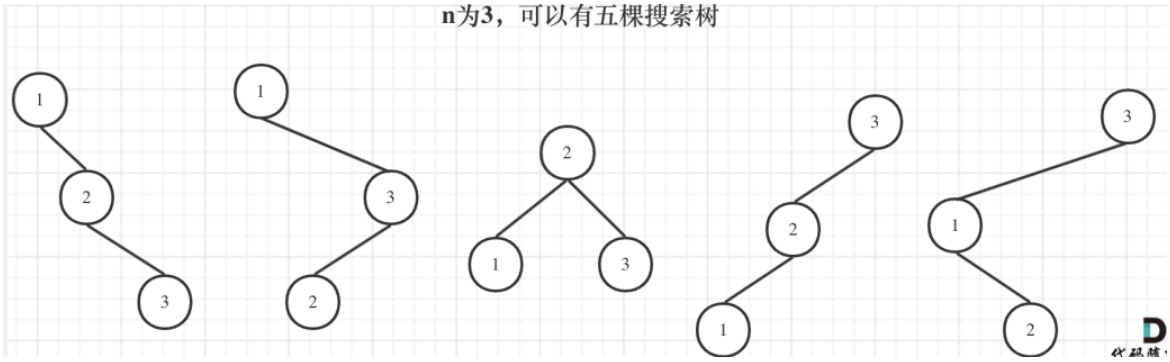
思路

了解了二叉搜索树之后，先举几个例子，画画图，看看有没有什么规律

- n为1的时候有一棵树，n为2有两棵树



- n为3的时候



  - 当1为头结点的时候，其右子树有两个节点，这两个节点的布局和 n 为2的时候两棵树的布局是一样的

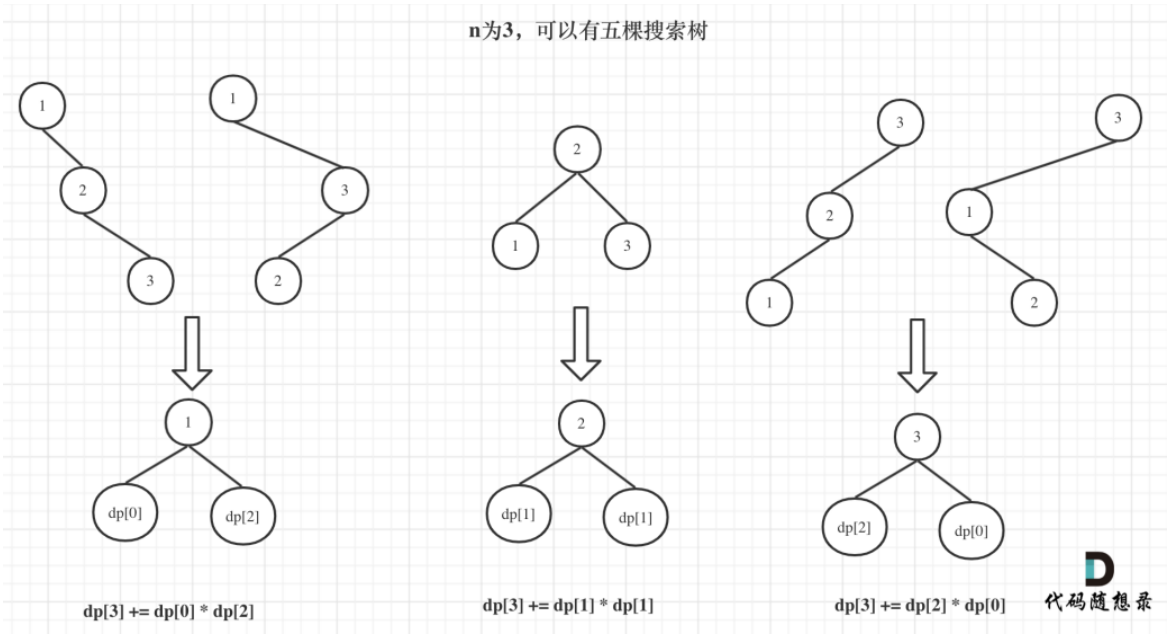
    要求不同树的数量，并不用把搜索树都列出来，所以不用关心其具体数值的差异

  - 当3为头结点的时候，其左子树有两个节点，这两个节点的布局和n为2的时候两棵树的布局是一样的
  - 当2位头结点的时候，其左右子树都只有一个节点，这两个节点布局和n为1的时候只有一棵树的 布局是一样的

=>

- `dp[3]` => 元素1为头节点搜索树的数量 + 元素2为头节点搜索树的数量 + 元素3为头节点搜索树的数量
  - 元素1为头节点搜索树的数量 => 右子树有2个元素的搜索树数量 * 左子树有0个元素的搜索树数量 => dp[2] * dp[0]
  - 元素2为头节点搜索树的数量 => 右子树有1个元素的搜索树数量 * 左子树有1个元素的搜索树数量 => dp[1] * dp[1]
  - 元素3为头节点搜索树的数量 => 右子树有0个元素的搜索树数量 * 左子树有2个元素的搜索树数量 => dp[0] * dp[2]

=> dp[3] = dp[2] * dp[0] + dp[1] * dp[1] + dp[0] * dp[2]



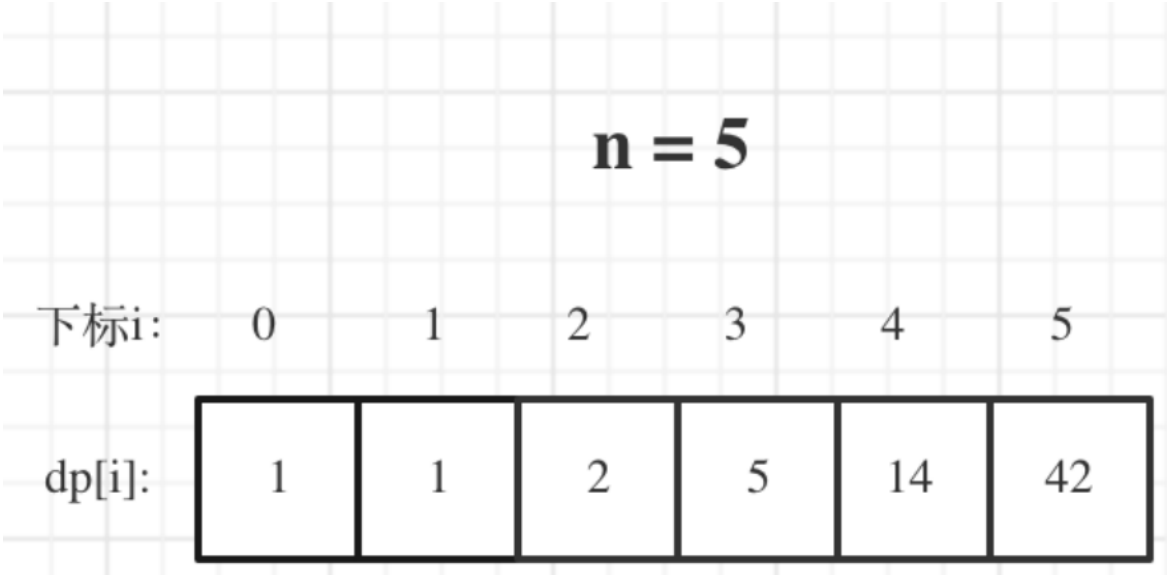
动规五部曲

1. 确定dp数组(dp table)以及下标的含义
   `dp[i]`： 1 到 i 为节点组成的二叉搜索树的个数为 `dp[i]` 。
2. 确定递推公式
   `dp[i] += dp[以j为头结点左子树节点数量] * dp[以j为头结点右子树节点数量]`
   j 相当于是头结点的元素，从1遍历到 i 为止。
   => 递推公式： `dp[i] += dp[j - 1] * dp[i - j]`; ，j-1 为 j 为头结点左子树节点数量，i - j 为以 j 为 头结点右子树节点数量
3. dp数组初始化
   初始化只需要初始化dp[0]即可，递推的基础都是dp[0]
   - 从定义上来讲，空节点也是一颗二叉树，也是一颗二叉搜索树
   - 从递推公式上来讲，dp[以j为头结点左子树节点数量] * dp[以j为头结点右子树节点数量] 中 以j为头结点左子树节点数量为0，也需要dp[以j为头结点左子树节点数量] = 1， 否则乘法的结果就都变成0了

   => `dp[0] = 1`
4. 确定遍历顺序
   首先一定是遍历节点数，从递推公式 `dp[i] += dp[j - 1] * dp[i - j]` 可以看出，节点数为 i 的状态是依赖 i 之前节点数的状态 => 遍历 i 里面每一个书作为头节点的状态，用 j 来遍历

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

5. 举例推导dp数组
   n = 5时的dp数组状态：



实现

Java代码

```java
// 时间复杂度O(n^2)
// 空间复杂度O(n)
class Solution {
    public int numTrees(int n) {
        int[] dp = new int[n + 1];
        dp[0] = 1;
        for(int i = 1;i <= n;++i) {
            for(int j = 1;j <= i;++j) {
                dp[i] += dp[j - 1] * dp[i - j];
            }
        }
        return dp[n];
    }
}
```