动态规划
（Dynamic programming）

王红元 coderwhy

目录
content

认识动态规划

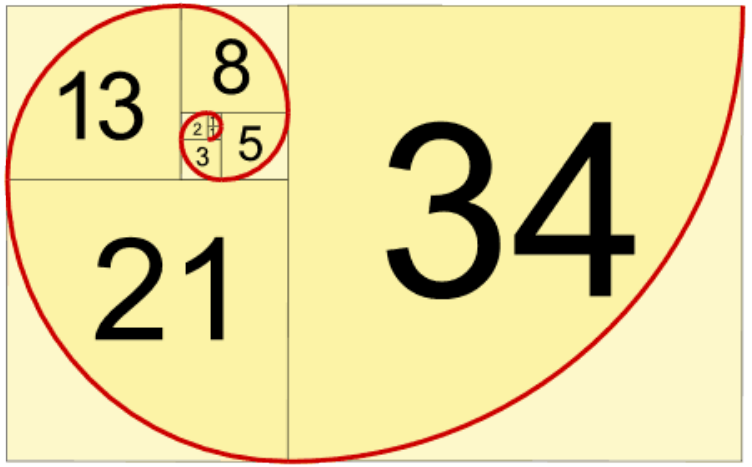
- **什么是动态规划？维基百科的解释**
  - 动态规划（英语：Dynamic programming，简称DP）是一种在数学、管理科学、计算机科学、经济学和生物信息学中使用的，通过把原问题分解为相对简单的子问题的方式求解复杂问题的方法。
- **动态规划的名字来源于20世纪50年代的一个美国数学家 Richard Bellman。**
  - 他在处理一类具有重叠子问题和最优子结构性质的问题时，想到了一种“动态”地求解问题的方法。
  - 它通过将问题划分为若干个子问题，并在计算子问题的基础上，逐步构建出原问题的解。
  - 他使用“动态规划”这个术语来描述这种方法，并将它应用于各种领域，如控制论、经济学、运筹学等。
- **动态规划（Dynamic Programming，可以简称DP）是一个非常重要的算法思想：**
  - 在算法竞赛、数据结构、机器学习等领域中，动态规划都是必不可少的知识之一。
- **动态规划也是互联网大厂和算法竞赛中非常喜欢考察的一类题目：**
  - 因为通过动态规划可以很好的看出一个人的思考问题的能力、逻辑的强度、程序和算法的设计等等。
  - 那么通过学习动态规划，可以提高算法设计和分析的能力，为解决复杂问题提供强有力的工具和思路。

动态规划的解题思路

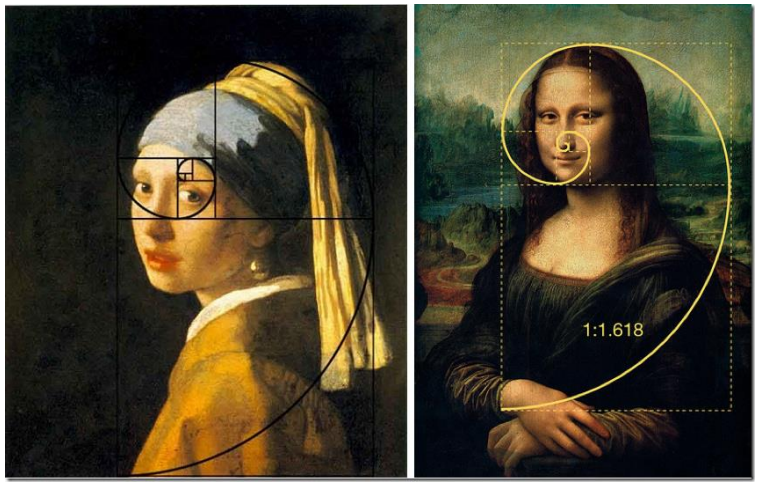
- **高深莫测、晦涩难懂?**
  - 很多人第一次接触动态规划时，往往会觉得这类题目高深莫测、晦涩难懂，不知道从何下手，甚至压根读不懂题意。
  - 往往会因为还没有完全入门就产生了困惑、迷茫，甚至是恐惧，最后直接放弃。
  - 我认为完全没有必要，只要掌握了动态规划的基本思路和实现方法，就可以很好地应用它解决各种问题。
- **动态规划的核心思想是“将问题划分为若干个子问题，并在计算子问题的基础上，逐步构建出原问题的解”。**
- **具体地说，动态规划通常涉及以下四个步骤：**
  - 步骤一：定义状态。
    - ✓ 将原问题划分为若干个子问题，定义状态表示子问题的解，通常使用一个数组或者矩阵来表示。
  - 步骤二：确定状态转移方程。
    - ✓ 在计算子问题的基础上，逐步构建出原问题的解。
    - ✓ 这个过程通常使用“状态转移方程”来描述，表示从一个状态转移到另一个状态时的转移规则。
  - 步骤三：初始化状态。
  - 步骤四：计算原问题的解（最终答案）。
    - ✓ 通过计算状态之间的转移，最终计算出原问题的解。
    - ✓ 通常使用递归或者迭代（循环）的方式计算。
- **这四个步骤是动态规划的核心思想，其中状态定义和状态转移方程是动态规划的关键。**

如何开始动态规划呢?

- **我们可以从一个最简单的算法：斐波那契数列开始。**
- **斐波那契数列是一个经典的数列，在自然界中很多地方都可以找到，它的定义如下：**
  - 第 0 个和第 1 个斐波那契数分别为0和1，即 F0 = 0, F1 = 1。
  - 从第 2 个数开始，每个斐波那契数都是它前面两个斐波那契数之和，即 F2 = F0 + F1，F3 = F1 + F2，F4 = F2 + F3，以此类推。



- **那么我们来看一下，如果我们要求斐波那契数列第N个数的值。**
- **那么我们有多少种求解的办法呢?**
  - 方式一：递归算法
  - 方式二：记忆化搜索
  - 方式三：动态规划的方案
  - 方式四：动态规划 – 状态压缩



斐波那契数列 – 递归求解

- **递归算法是一种基本的算法思想：**
  - 其基本思想是将一个大问题拆分成若干个相似的小问题。
  - 然后通过解决这些小问题来解决整个大问题。
- **递归算法通常采用函数自身调用的方式实现，每次调用函数时都会处理一个规模更小的问题，直到问题规模足够小，可以直接求解为止。**

```
function fibonacci(n: number): number {
  if (n <= 1) {
    return n;
  }
  return fibonacci(n - 1) + fibonacci(n - 2);
}
```

- **当 n 小于等于 1 时，直接返回 n；否则，递归调用 fibonacci 函数来计算 n-1 和 n-2 两个子问题的结果，然后将它们相加得到结果。**
- **递归函数必须有一个终止条件，以确保递归过程能够结束。**

斐波那契数列 – 记忆化搜索

- 对于递归算法，很容易出现重复计算的问题，因为在计算同一个子问题时，可能会被重复地计算多次。
- 为了避免这个问题，我们可以使用记忆化搜索（Memoization）的技巧，将已经计算过的结果保存下来，以便在后续的计算中直接使用。
- 下面是一个使用记忆化搜索优化的斐波那契数列实现，它可以避免重复计算，提高计算效率：

```
function fibonacci(n: number, memo: number[] = []): number {
  if (n <= 1) {
    return n;
  }
  if (memo[n] != null) {
    return memo[n];
  }
  const result = fibonacci(n - 1, memo) + fibonacci(n - 2, memo);
  memo[n] = result;
  return result;
}
```

- 这个实现和前面的递归实现非常相似，只是增加了一个 memo 参数，用于保存已经计算过的结果。
  - 在实际应用中，记忆化搜索可以极大地提高递归算法的效率，特别是对于有大量重复计算的问题，优化效果尤为明显。
  - 这种解法也可以称之为自顶向下的解法；

斐波那契数列 – 动态规划

- **从上面的斐波那契数列的例子中，我们可以发现，通过记忆化搜索的方式，可以避免重复计算，提高计算效率。**
- **而动态规划（Dynamic Programming）算法就是一种利用历史状态信息来避免重复计算的算法。**
  - 动态规划算法可以看作是记忆化搜索的一种扩展，它通常采用自底向上的方式计算子问题的结果，并将结果保存下来以便后续的计算使用。
  - 在动态规划算法中，通常需要明确定义状态、设计状态转移方程、初始化状态，以及确定计算顺序等。
- **下面我们可以以斐波那契数列为例，介绍如何用动态规划算法来解决这个问题：**

```
function fibonacci(n: number): number {
  if (n <= 1) {
    return n;
  }
  const dp = [0, 1];
  for (let i = 2; i <= n; i++) {
    dp[i] = dp[i - 1] + dp[i - 2];
  }
  return dp[n];
}
```

- 需要注意的是，在动态规划算法中，为了保证状态之间的依赖关系正确，通常需要按照一定的计算顺序来计算子问题的结果。
- 对于斐波那契数列问题来说，我们采用自底向上的方式计算子问题的结果，确保 dp[i-1] 和 dp[i-2] 的值已经计算出来了，才能计算 dp[i] 的值。

斐波那契数列 – 动态规划（状态压缩）

- 在动态规划算法中，有一种常见的优化方法叫做状态压缩，可以将状态的存储空间从数组优化为一个常数。
- 对于斐波那契数列问题来说，我们只需要保存 dp[i-1] 和 dp[i-2] 两个状态的值，就能够计算出 dp[i] 的值，因此可以使用两个变量来存储这两个状态的值，从而实现状态压缩的优化。
- 以下是使用状态压缩优化后的代码：

```
function fibonacci(n: number): number {
  if (n <= 1) {
    return n;
  }
  let prev = 0;
  let curr = 1;
  for (let i = 2; i <= n; i++) {
    const next = prev + curr;
    prev = curr;
    curr = next;
  }
  return curr;
}
```

- 这个实现和前面的动态规划实现相比，减少了存储空间的使用，优化了空间复杂度。

509. 斐波那契数

- 509. 斐波那契数
  - 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
```

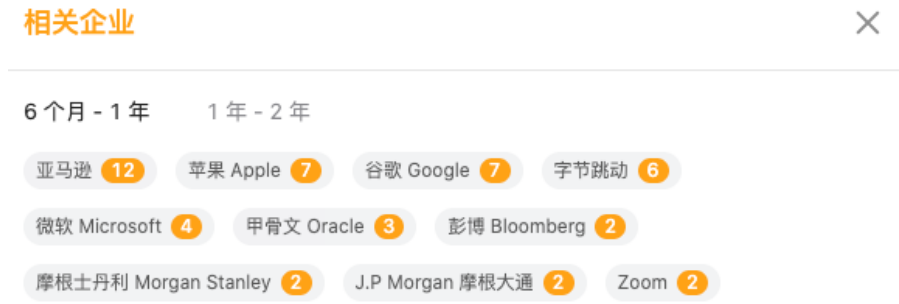
给定 `n` ，请计算 `F(n)` 。

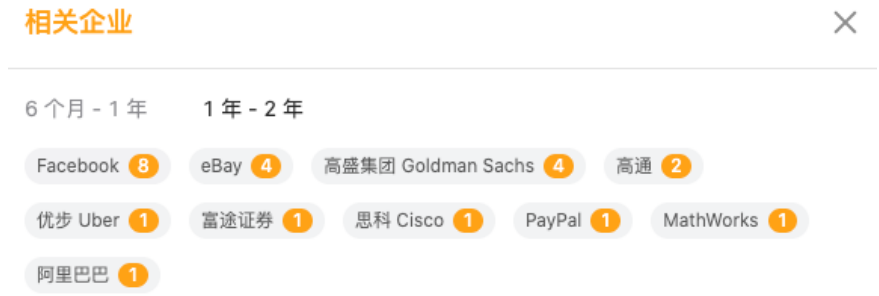
示例 1：

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

示例 2：

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





动态规划的统一解题步骤

- **当遇到需要使用动态规划来解决的问题时，可以按照以下步骤进行解题：**
- **步骤一：定义状态：**
  - 明确状态的含义，通常需要使用一个或多个变量来表示状态。
  - 状态表示问题的解空间中的某个状态。
- **步骤二：找到状态转移方程：**
  - 根据题目的要求和状态的定义，写出状态转移方程。
  - 状态转移方程表示的是从当前状态到下一个状态的转移规律，是动态规划算法的核心。
- **步骤三：确定初始状态：**
  - 确定状态转移过程中的初始状态，也就是问题的边界。
  - 初始状态是转移方程的基础，也是状态转移的起点。
- **步骤四：计算最终状态：**
  - 根据状态转移方程和初始状态，计算出最终状态的值。
  - 最终状态是问题的解，也是状态转移的终点。

爬楼梯（跳台阶）

- **爬楼梯（或者称之为跳台阶，我个人一直叫跳台阶）是一道经典的动态规划题目，也是面试常考的一道题目。**
- **爬楼梯**：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?
- **跳台阶**：假设有 n 级台阶，每次可以跳 1 级或 2 级台阶，问有多少种不同的跳法可以跳到第 n 级台阶。
- **题目解析：**
  - 跳台阶问题是一道经典的动态规划问题，其本质是要求出到达第 n 级台阶的跳法数量。
  - 而到达第 n 级台阶只能由第 n-1 级台阶或第 n-2 级台阶跳上来，因此需要借助动态规划算法进行求解。
  - 通过引入状态、设计状态转移方程、初始化状态等方法，可以高效地求解跳台阶问题。
- **这道题目我们依然采用不同的方案来实现，让大家体会到动态规划的好处：**
  - 方式一：暴力递归；
  - 方式二：记忆化搜索；
  - 方式三：动态规划；
  - 方式四：状态压缩；

跳台阶 – 暴力递归

- **我们可以先从暴力递归的方式开始，分析问题的本质，然后再逐步引入动态规划算法进行优化。**
- **对于跳台阶问题，假设有 n 级台阶，我们要求出到达第 n 级台阶的不同跳法数量。**
  - 可以发现，从第 n 级台阶只能由第 n-1 级台阶或第 n-2 级台阶跳上来；
  - 因此，到达第 n 级台阶的跳法数量等于到达第 n-1 级台阶的跳法数量加上到达第 n-2 级台阶的跳法数量；
  - 即：jump(n) = jump(n-1) + jump(n-2)
  - 对于 n=0 和 n=1 的情况，跳法数量分别为 1 和 1

```
function jump(n: number): number {
  if (n <= 1) {
    return 1;
  }
  return jump(n - 1) + jump(n - 2);
}
```

跳台阶 – 记忆化搜索

- **在介绍完暴力递归之后，我们可以引入记忆化搜索（Memoization）的方式进行优化。**
  - 对于跳台阶问题，我们可以使用一个长度为 n+1 的数组 memo，用来记录每个阶梯的跳法数量。
  - 初始时，我们将 memo 数组中所有元素都初始化为 0。
  - 然后在递归过程中，如果 memo[n] 已经被计算过，直接返回 memo[n]。
  - 否则计算 memo[n] 的值，并将其存储到 memo[n] 中。

```
function jump(n: number): number {
  const memo = new Array(n + 1).fill(0);
  return jumpMemo(n, memo);
}

function jumpMemo(n: number, memo: number[]): number {
  if (n <= 1) {
    return 1;
  }
  if (memo[n] !== 0) {
    return memo[n];
  }
  memo[n] = jumpMemo(n - 1, memo) + jumpMemo(n - 2, memo);
  return memo[n];
}
```

跳台阶 – 动态规划

- **我们可以使用一个一维数组来记录跳台阶的结果。**
- **我们可以定义一个长度为 n+1 的一维数组 dp，用来记录每个阶梯的跳法数量。**
  - 初始时，我们将 dp 数组中所有元素都初始化为 0；
  - 然后设置 dp[0] = 1，dp[1] = 1，表示到达第 0 级台阶和第 1 级台阶时，只有 1 种跳法。
  - 接下来，我们可以使用循环，依次计算 dp[2]、dp[3]、…、dp[n] 的值，最终得到 dp[n] 即为答案。

```
function jump(n: number): number {
  const dp = new Array(n + 1).fill(0);
  dp[0] = 1;
  dp[1] = 1;
  for (let i = 2; i <= n; i++) {
    dp[i] = dp[i - 1] + dp[i - 2];
  }
  return dp[n];
}
```

跳台阶 – 滚动数组（滑动窗口）

- **另外一种常见的优化方法是滚动数组（滑动窗口）的方式。**
- **滚动数组的基本思想是：**
  - 由于每个状态只与它之前的状态有关。
  - 因此我们不需要记录所有的状态，只需要记录当前状态和它之前的若干个状态即可。
  - 通过不断更新这个滚动窗口，可以避免使用额外的空间，将空间复杂度进一步降低。

```
function jump(n: number): number {
  let a = 1;
  let b = 1;
  for (let i = 2; i <= n; i++) {
    const c = a + b;
    a = b;
    b = c;
  }
  return b;
}
```

- **使用滚动数组的方式，可以将算法的空间复杂度降到 O(1)，是一种非常高效的动态规划优化方式。**

70. 爬楼梯 - 亚马逊 – Google – 华为等

- **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?
  - https://leetcode.cn/problems/climbing-stairs/description/

亚马逊 谷歌 Google 微软 Microsoft ···

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?

示例 1:

```
输入: n = 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
```

示例 2:

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

121. 买卖股票的最佳时机

- 121. 买卖股票的最佳时机
  - https://leetcode.cn/problems/best-time-to-buy-and-sell-stock/
- 给定一个数组 prices ， 它的第 i 个元素 prices[i] 表示一支给定股票第 i 天的价格。
- 你只能选择 **某一天** 买入这只股票，并选择在 **未来的某一个不同的日子** 卖出该股票。设计一个算法来计算你所能获取的最大利润。
- 返回你可以从这笔交易中获取的最大利润。如果你不能获取任何利润，返回 0 。

示例 1：

```
输入：[7,1,5,3,6,4]
输出：5
解释：在第 2 天（股票价格 = 1）的时候买入，在第 5 天（股票价格 = 6）的时候卖出，最大利润 = 6-1 = 5 。
     注意利润不能是 7-1 = 6，因为卖出价格需要大于买入价格；同时，你不能在买入前卖出股票。
```

示例 2：

```
输入：prices = [7,6,4,3,1]
输出：0
解释：在这种情况下，没有交易完成，所以最大利润为 0。
```

动态规划的实现思路

- **定义状态：设 dp[i] 表示前 i 天中能够获取的最大利润。**
- **状态转移方程：**
  - 对于第 i 天，有两种情况：
    1. 在第 i 天卖出股票，在第i天的价格 减去 之前最便宜那天的买入价格，因此可以得到利润为 prices[i] - minPrice；
    2. 在第 i 天不卖出股票，那么目前的最大利润依然是前一天的最大利润 dp[i-1]。
  - 可以得到状态转移方程为：dp[i] = max(dp[i-1], prices[i] - minPrice)
- **初始状态：**由于最小的天数是 1，因此初始状态为 dp[0] = 0。
- **计算最终状态：**最后一天保留下来的最大利润的值。

```
function maxProfit(prices: number[]): number {
  // 1.获取数组的长度
  const n = prices.length
  if (n < 2) return 0

  // 2.定义数组，保存每个位置的最大值
  const dp: number[] = []
  dp[0] = 0
  let minPrice = prices[0] // 记录最小值
  for (let i = 1; i < n; i++) {
    // 比较昨天的最大利润和今天卖出的最大利润
    dp[i] = Math.max(dp[i - 1], prices[i] - minPrice)
    minPrice = Math.min(prices[i], minPrice)
  }

  return dp[n-1]
}
```

状态压缩

- 对于这个问题，实际上可以进行状态压缩。
- 由于在状态转移方程中，当前状态只与前一个状态有关，因此可以不用维护整个 dp 数组，只需要用一个变量来表示前一个状态的最大利润即可。

```
function maxProfit(prices: number[]): number {
  // 1.获取数组的长度
  const n = prices.length
  if (n < 2) return 0

  // 2.定义数组，保存每个位置的最大值
  let preValue = 0
  let minPrice = prices[0] // 记录最小值
  for (let i = 1; i < n; i++) {
    // 比较昨天的最大利润和今天卖出的最大利润
    preValue = Math.max(preValue, prices[i] - minPrice)
    minPrice = Math.min(prices[i], minPrice)
  }

  return preValue
}
```

53. 最大子数组和

- 53. 最大子数组和
  - https://leetcode.cn/problems/maximum-subarray/
- 给你一个整数数组 nums， 请你找出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。
- **子数组** 是数组中的一个连续部分。

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 。
```

示例 2：

```
输入：nums = [1]
输出：1
```

示例 3：

```
输入：nums = [5,4,-1,7,8]
输出：23
```

动态规划的实现思路

-2	1	-3	4	-1	2	1	-5	4

-2	1	-2	4	3	5	6	1	5

- **动态规划的规律：**
  - 以每个位置结尾的最大子序列的计算方式：
    - ✓ 如果前面的子序列是负数，那么最大子序列和一定是自己；
    - ✓ 如果前面的子序列是正数，那么最大子序列和是自己+前值；
  - 由此可以得出计算公式：
    - ✓ dp[i] = max(dp[i-1] + nums[i], nums[i])
  - 初始化值：dp[0] = nums[0]
  - 计算最终值：找出所有值中最大的值即可

```
function maxSubArray(nums: number[]): number {
  // 1.获取数组的长度
  const n = nums.length

  // 2.定义保存状态的数组
  const dp: number[] = new Array(n)
  dp[0] = nums[0] // 初始状态

  // 3.从索引值1个开始计算
  let max = dp[0]
  for (let i = 1; i < n; i++) {
    const value1 = nums[i] // 当前位置的值
    const value2 = dp[i - 1] + nums[i] // +前一个的值
    dp[i] = Math.max(value1, value2)

    // 原来的max和最新的dp[i]进行比较
    max = Math.max(max, dp[i])
  }

  return max
}
```

状态压缩的优化

- **在动态规划算法中，我们需要定义一个一维数组 dp，其中 dp[i] 表示以第 i 个元素结尾的子数组的最大和。**
  - 根据动态转移方程 dp[i] = max(dp[i-1] + nums[i], nums[i])，我们可以计算出 dp 数组中的每个元素，从而求解原问题。
  - 这个算法的空间复杂度为 O(n)。
- **然而，我们可以发现，dp 数组中的每个元素只与前一个元素有关。**
- **因此，我们可以使用滚动数组的技巧，将一维数组 dp 压缩成一个变量 maxSum，从而将空间复杂度优化为 O(1)。**

```
function maxSubArray(nums: number[]): number {
  // 1.获取数组的长度
  const n = nums.length

  // 2.定义保存状态的数组(或者命名为curSum)
  let preSum = nums[0]

  // 3.从索引值1个开始计算
  let max = preSum
  for (let i = 1; i < n; i++) {
    const value1 = nums[i] // 当前位置的值
    const value2 = preSum + nums[i] // +前一个的值
    preSum = Math.max(value1, value2)

    // 原来的max和最新的dp[i]进行比较
    max = Math.max(max, preSum)
  }

  return max
}
```

62. 不同路径

- 62. 不同路径
  - https://leetcode.cn/problems/unique-paths/description/
- 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?

输入： m = 3, n = 7
输出： 28

示例 2：

输入： m = 3, n = 2
输出： 3
解释：
从左上角开始，总共有 3 条路径可以到达右下角。
1. 向右 -> 向下 -> 向下
2. 向下 -> 向下 -> 向右
3. 向下 -> 向右 -> 向下

0 - 6 个月　6 个月 - 1 年　1 年 - 2 年

亚马逊 9　谷歌 Google 4　字节跳动 3　彭博 Bloomberg 3

0 - 6 个月　6 个月 - 1 年　1 年 - 2 年

微软 Microsoft 10　苹果 Apple 9　Facebook 7　华为 2

0 - 6 个月　6 个月 - 1 年　1 年 - 2 年

甲骨文 Oracle 3　摩根士丹利 Morgan Stanley 2　高盛集团 Goldman Sachs 2

优步 Uber 2　Wish 2　快手 1　腾讯 1　eBay 1　Citadel 1

PayPal 1

动态规划的实现思路

- **这个题目和跳楼梯其实是一类题目。**
- **设 dp[i][j] 表示从起点到网格的 (i, j) 点的不同路径数。**
- **对于每个格子，由于机器人只能从上面或左边到达该格子，因此有以下两种情况：**
  - 从上面的格子到达该格子，即 dp[i][j] = dp[i-1][j]；
  - 从左边的格子到达该格子，即 dp[i][j] = dp[i][j-1]。
    - ✓ 因此，到达网格的 (i, j) 点的不同路径数就等于到达上面格子的路径数加上到达左边格子的路径数。
  - 动态转移方程为：dp[i][j] = dp[i-1][j] + dp[i][j-1]；
- **初始状态：**对于边界情况，起点的路径数为 1，即 dp[0][0] = 1。
- **计算最终状态：**dp[m-1][n-1]


```
function uniquePaths(m: number, n: number): number {
    let dp = Array.from({length: m}, () => Array(n).fill(0))
    for (let i = 0; i < m; i++) {
        dp[i][0] = 1
    }
    for (let j = 0; j < n; j++) {
        dp[0][j] = 1
    }
    for (let i = 1; i < m; i++) {
        for (let j = 1; j < n; j++) {
            dp[i][j] = dp[i-1][j] + dp[i][j-1]
        }
    }
    return dp[m-1][n-1]
};
```

不同路径的组合数学（课下扩展）

- 我们可以使用组合数学的方法，通过计算总共需要向下和向右走的步数，从而计算不同的路径数目。
- 假设总共需要向下走 n 步，向右走 m 步，则路径的总长度为 n + m，其中需要选择 n 个位置向下走，因此路径的总数目为 C(n + m, n)。

```
function uniquePaths(m: number, n: number): number {
  // 总共需要走的步数
  const total = m + n - 2;
  // 向下走的步数
  const down = n - 1;
  // 不同路径的数目
  let ans = 1;
  // 使用组合数学的方法计算不同路径数目
  for (let i = 1; i <= down; i++) {
    ans *= total - down + i; // C(total-down+i, i)的分子
    ans /= i; // C(total-down+i, i)的分母
  }
  // 返回计算结果
  return ans;
}
```