王争的算法训练营

习题课：背包和路径

动态规划

王争的算法训练营

重中之重，大厂笔试、面试必考项、学习难但面试不难、掌握模型举一反三

DP专题：

- 专题：适用问题
- 专题：解题步骤
- 专题：最值、可行、计数三种类型　一些特殊小类别：树形DP、区间DP、数位DP
- 专题：空间优化

经典模型：

- 背包问题（0-1、完全、多重、二维费用、分组、有依赖的）
- 路径问题
- 打家劫舍&股票买卖
- 爬楼梯问题
- 匹配问题（LCS、编辑距离）
- 其他（LIS）

配套习题（24）：

背包：

416. 分割等和子集
494. 目标和
322. 零钱兑换
518. 零钱兑换 II

路径问题

62. 不同路径
63. 不同路径 II
64. 最小路径和

剑指 Offer 47. 礼物的最大价值

120. 三角形最小路径和

打家劫舍 & 买卖股票：

198. 打家劫舍
213. 打家劫舍 II
337. 打家劫舍 III (树形DP)
714. 买卖股票的最佳时机含手续
309. 最佳买卖股票时机含冷冻期

爬楼梯问题

70. 爬楼梯
322. 零钱兑换
518. 零钱兑换 II

剑指 Offer 14- I. 剪绳子

剑指 Offer 46. 把数字翻译成字符串

139. 单词拆分

匹配问题

1143. 最长公共子序列
72. 编辑距离

其他

437. 路径总和 III (树形DP)
300. 最长递增子序列

动态规划解题过程：

1. **可用回溯解决：** 需要穷举搜索才能得到结果的问题（最值、可行、计数等）
2. **构建多阶段决策模型。** 看是否能将问题求解的过程分为多个阶段。
3. **查看是否存在重复子问题：** 是否有多个路径到达同一个状态。
4. **定义状态：** 也就是如何记录每一阶段的不重复状态。
5. **定义状态转移方程：** 也就是找到如何通过上一阶段的状态推导下一下阶段的状态。
6. **画状态转移表：** 辅助理解，验证正确性，确定状态转移的初始值。
7. **编写动态规划代码。**

黄色标记的两个步骤是难点，掌握的技巧就是：记忆经典模型的状态和状态转移方程的定义方法，举一反三。

背包：

416. 分割等和子集
494. 目标和
322. 零钱兑换
518. 零钱兑换 II

416. 分割等和子集

给你一个 **只包含正整数** 的 **非空** 数组 `nums` 。请你判断是否可以将这个数组分割成两个子集，使得两个子集的元素和相等。

抽象为模型：
0-1背包问题，是否能装满背包

示例 1：

```
输入：nums = [1,5,11,5]
输出：true
解释：数组可以分割成 [1, 5, 5] 和 [11] 。
```

示例 2：

```
输入：nums = [1,2,3,5]
输出：false
解释：数组不能分割成两个元素和相等的子集。
```

提示：

- `1 <= nums.length <= 200`
- `1 <= nums[i] <= 100`

416. 分割等和子集

```
class Solution {
    public boolean canPartition(int[] nums) {
        int n = nums.length;
        int sum = 0;
        for (int i = 0; i < n; i++) {
            sum += nums[i];
        }
        if (sum % 2 == 1) return false;
        sum /= 2;

        boolean[][] dp = new boolean[n][sum+1];
        dp[0][0] = true;
        if (nums[0]<=sum) {
            dp[0][nums[0]] = true;
        }
        for (int i = 1; i < n; ++i) {
            for (int j = 0; j <= sum; ++j) {
                if (j-nums[i]>=0) {
                    dp[i][j] = dp[i-1][j] || dp[i-1][j-nums[i]];
                } else {
                    dp[i][j] = dp[i-1][j];
                }
            }
        }
        return dp[n-1][sum];
    }
}
```

494. 目标和

抽象为模型：
0-1背包问题，所装物品总重量为target，有多少种装法

给你一个整数数组 `nums` 和一个整数 `target` 。

向数组中的每个整数前添加 `'+'` 或 `'-'` ，然后串联起所有整数，可以构造一个 **表达式** ：

- 例如， `nums = [2, 1]` ，可以在 `2` 之前添加 `'+'` ，在 `1` 之前添加 `'-'` ，然后串联起来得到表达式 `"+2-1"` 。

返回可以通过上述方法构造的、运算结果等于 `target` 的不同 **表达式** 的数目。

示例 1：

```
输入：nums = [1,1,1,1,1], target = 3
输出：5
解释：一共有 5 种方法让最终目标和为 3 。
-1 + 1 + 1 + 1 + 1 = 3
+1 - 1 + 1 + 1 + 1 = 3
+1 + 1 - 1 + 1 + 1 = 3
+1 + 1 + 1 - 1 + 1 = 3
+1 + 1 + 1 + 1 - 1 = 3
```

示例 2：

```
输入：nums = [1], target = 1
输出：1
```

提示：

- `1 <= nums.length <= 20`
- `0 <= nums[i] <= 1000`
- `0 <= sum(nums[i]) <= 1000`
- `-1000 <= target <= 100`

int dp[n][w+1]记录每个阶段可达的状态。

dp[i][j] 表示第i个物品决策完成之后，到达背包中物品的重量为j这种状态有多少种方法。

第i个物品只有两个决策方式：加或者减。所以，(i, j)只有可能从两个状态转移过来：

1）第i个物品加，从状态dp[i-1][j-nums[i]]转移过来。

2）第i个物品减，从状态dp[i-1][j+nums[i]]。

状态转移方程为：dp[i][j] = dp[i-1][j-nums[i]] + dp[i-1][j+nums[i]];

494. 目标和

```
class Solution {
    public int findTargetSumWays(int[] nums, int S) {
        if (S > 1000 || S < -1000) return 0;
        int n = nums.length;
        int offset = 1000;
        int w = 2000;
        int[][] dp = new int[n][w+1];
        dp[0][offset-nums[0]] += 1; // 因为nums[0]有可能为0
        dp[0][offset+nums[0]] += 1;

        for (int i = 1; i < n; ++i) {
            for (int j = 0; j <= w; ++j) {
                if (j-nums[i]>=0 && j-nums[i]<=w) {
                    dp[i][j] = dp[i-1][j-nums[i]];
                }
                if (j+nums[i]>=0 && j+nums[i]<=w) {
                    dp[i][j] += dp[i-1][j+nums[i]];
                }
            }
        }
        return dp[n-1][S+1000];
    }
}
```

322. 零钱兑换

给定不同面额的硬币 `coins` 和一个总金额 `amount`。编写一个函数来计算可以凑成总金额所需的最少的硬币个数。如果没有任何一种硬币组合能组成总金额，返回 `-1`。

你可以认为每种硬币的数量是无限的。

完全背包问题：
“零钱兑换”题目是最优问题：最少需要多少物品能填满背包
“零钱兑换II”是计数问题：填满背包有多少种方法

示例 1：

```
输入：coins = [1, 2, 5], amount = 11
输出：3
解释：11 = 5 + 5 + 1
```

示例 2：

```
输入：coins = [2], amount = 3
输出：-1
```

示例 3：

```
输入：coins = [1], amount = 0
输出：0
```

```java
class Solution {
    public int coinChange(int[] coins, int amount) {
        int n = coins.length;
        // 第i个硬币决策完之后，凑足金额j需要的最少硬币数dp[i][j]
        int[][] dp = new int[n][amount + 1];
        for (int i = 0; i < n; ++i) {
            for (int j = 0; j <= amount; ++j) {
                dp[i][j] = Integer.MAX_VALUE;
            }
        }
        for (int c = 0; c <= amount/coins[0]; ++c) {
            dp[0][c*coins[0]] = c;
        }

        for (int i = 1; i < n; ++i) {
            for (int j = 0; j <= amount; ++j) {
                int k = j/coins[i];
                for (int c = 0; c <= k; ++c) {
                    if (dp[i-1][j-c*coins[i]] != Integer.MAX_VALUE &&
                        dp[i-1][j-c*coins[i]] + c < dp[i][j]) {
                            dp[i][j] = dp[i-1][j-c*coins[i]] + c;
                    }
                }
            }
        }
        if (dp[n-1][amount] == Integer.MAX_VALUE) return -1;
        return dp[n-1][amount];
    }
}
```

518. 零钱兑换 II

给定不同面额的硬币和一个总金额。写出函数来计算可以凑成总金额的硬币组合数。假设每一种面额的硬币有无限个。

完全背包问题：
“零钱兑换”题目是最优问题：最少需要多少物品能填满背包
“零钱兑换II”是计数问题：填满背包有多少种方法

示例 1:

```
输入: amount = 5, coins = [1, 2, 5]
输出: 4
解释: 有四种方式可以凑成总金额:
5=5
5=2+2+1
5=2+1+1+1
5=1+1+1+1+1
```

示例 2:

```
输入: amount = 3, coins = [2]
输出: 0
解释: 只用面额2的硬币不能凑成总金额3。
```

518. 零钱兑换 II

```
class Solution {
    public int change(int amount, int[] coins) {
        int n = coins.length;
        int[][] dp = new int[n][amount+1];
        for (int c = 0; c <= amount/coins[0]; ++c) {
            dp[0][c*coins[0]] = 1;
        }

        for (int i = 1; i < n; ++i) {
            for (int j = 0; j <= amount; ++j) {
                int k = j/coins[i];
                for (int c = 0; c <= k; ++c) {
                    dp[i][j] += dp[i-1][j-c*coins[i]];
                }
            }
        }
        return dp[n-1][amount];
    }
}
```

路径问题

64. 最小路径和

剑指 Offer 47. 礼物的最大价值

120. 三角形最小路径和

62. 不同路径

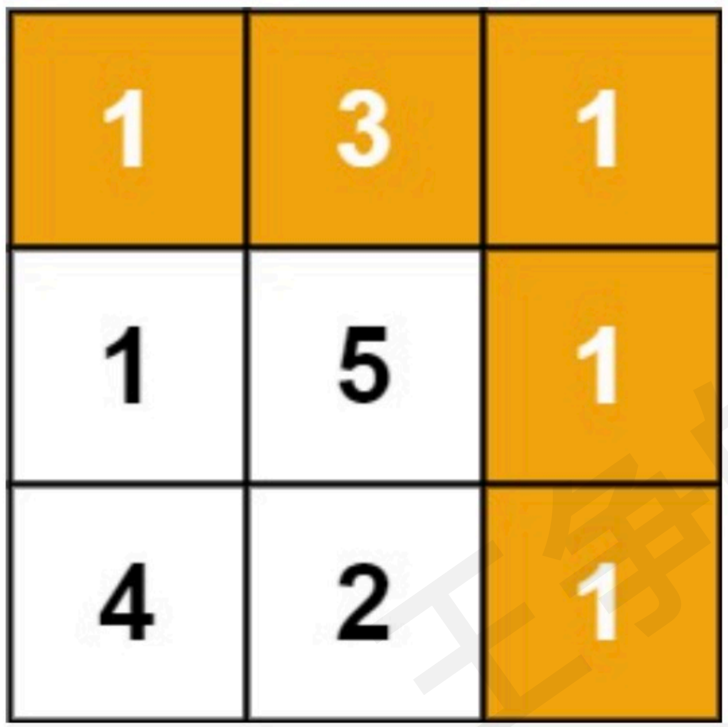
63. 不同路径 II

64. 最小路径和

给定一个包含非负整数的 `m x n` 网格 `grid`，请找出一条从左上角到右下角的路径，使得路径上的数字总和为最小。

说明： 每次只能向下或者向右移动一步。

示例 1：




```
输入：grid = [[1,3,1],[1,5,1],[4,2,1]]
输出：7
解释：因为路径 1→3→1→1→1 的总和最小。
```

1、构建多阶段决策模型

从(0, 0)走到(m-1, n-1)，总共要走m+n-2步，也就对应着m+n-2个决策阶段。每个阶段都有向右走或者向下走两种决策选择。

2、定义状态

int dp[m][n]; dp[i][j]表示到达(i, j)这个位置的最短路径

3、定义状态转移方程

(i, j)这个位置只有可能通过(i-1, j)和(i, j-1)两个位置过来

dp[i][j]=min(dp[i-1][j]+dp[i][j-1])+grid[i][j]

64. 最小路径和

```
class Solution {
    public int minPathSum(int[][] grid) {
        int m = grid.length;
        int n = grid[0].length;
        int[][] dp = new int[m][n];
        int len = 0;
        for (int i = 0; i < m; ++i) {
            len += grid[i][0];
            dp[i][0] = len;
        }
        len = 0;
        for (int j = 0; j < n; ++j) {
            len += grid[0][j];
            dp[0][j] = len;
        }

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

剑指 Offer 47. 礼物的最大价值

在一个 m*n 的棋盘的每一格都放有一个礼物，每个礼物都有一定的价值（价值大于 0）。你可以从棋盘的左上角开始拿格子里的礼物，并每次向右或者向下移动一格、直到到达棋盘的右下角。给定一个棋盘及其上面的礼物的价值，请计算你最多能拿到多少价值的礼物？

示例 1:

```
输入:
[
  [1,3,1],
  [1,5,1],
  [4,2,1]
]
输出: 12
解释: 路径 1→3→5→2→1 可以拿到最多价值的礼物
```

剑指 Offer 47. 礼物的最大价值

```java
class Solution {
    public int maxValue(int[][] grid) {
        int n = grid.length;
        int m = grid[0].length;
        int[][] dp = new int[n][m];
        int sum = 0;
        for (int j = 0; j < m; ++j) {
            sum += grid[0][j];
            dp[0][j] = sum;
        }
        sum = 0;
        for (int i = 0; i < n; ++i) {
            sum += grid[i][0];
            dp[i][0] = sum;
        }

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

120. 三角形最小路径和

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）。
```

示例 2：

```
输入：triangle = [[-10]]
输出：-10
```

提示：

- `1 <= triangle.length <= 200`
- `triangle[0].length == 1`
- `triangle[i].length == triangle[i - 1].length + 1`
- -10^4 `<= triangle[i][j] <=` 10^4

```java
class Solution {
    public int minimumTotal(List<List<Integer>> triangle) {
        int n = triangle.size();
        int[][] dp = new int[n][n];
        dp[0][0] = triangle.get(0).get(0);
        for (int i = 1; i < n; ++i) {
            dp[i][0] = dp[i-1][0] + triangle.get(i).get(0);
            for (int j = 1; j < i; ++j) {
                dp[i][j] = Math.min(dp[i-1][j], dp[i-1][j-1]) + triangle.get(i).get(j);
            }
            dp[i][i] = dp[i-1][i-1] + triangle.get(i).get(i);
        }
        int res = Integer.MAX_VALUE;
        for (int j = 0; j < n; ++j) {
            if (dp[n-1][j] < res) res = dp[n-1][j];
        }
        return res;
    }
}
```

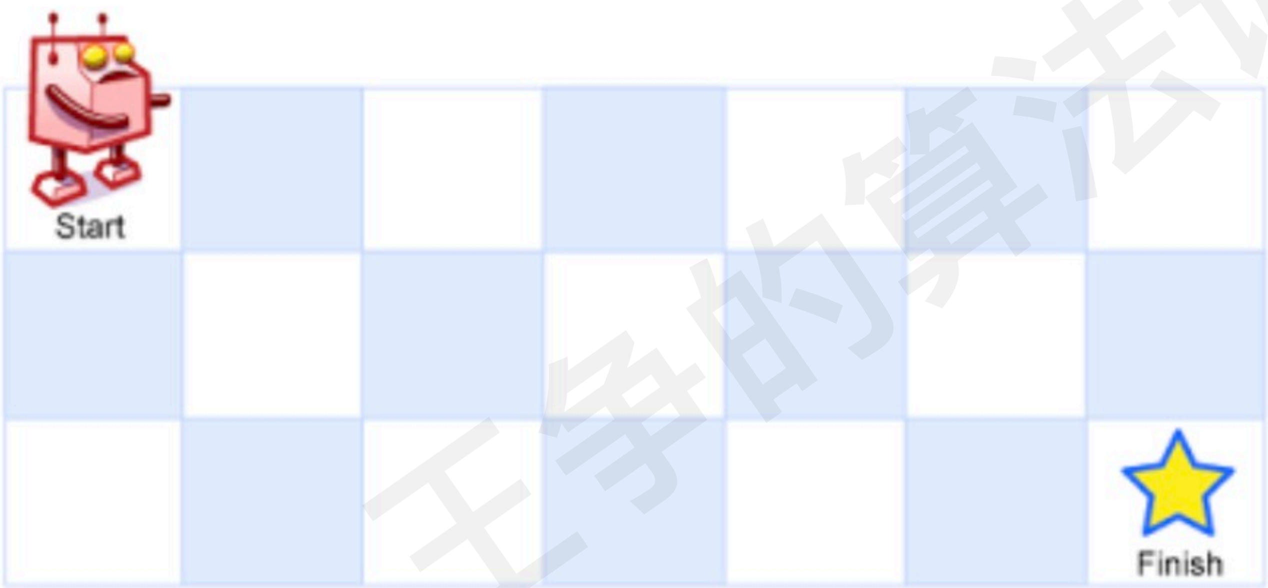
[62. 不同路径]

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
```

1、构建多阶段决策模型

从(0, 0)走到(m-1, n-1)，总共要走m+n-2步，也就对应着m+n-2个决策阶段。每个阶段都有向右走或者向下走两种决策选择。

2、定义状态

int dp[m][n]; dp[i][j]表示到达(i, j)这个位置的路径条数

3、定义状态转移方程

(i, j)这个位置只有可能通过(i-1, j)和(i, j-1)两个位置过来

dp[i][j]=dp[i-1][j]+dp[i][j-1]

62. 不同路径

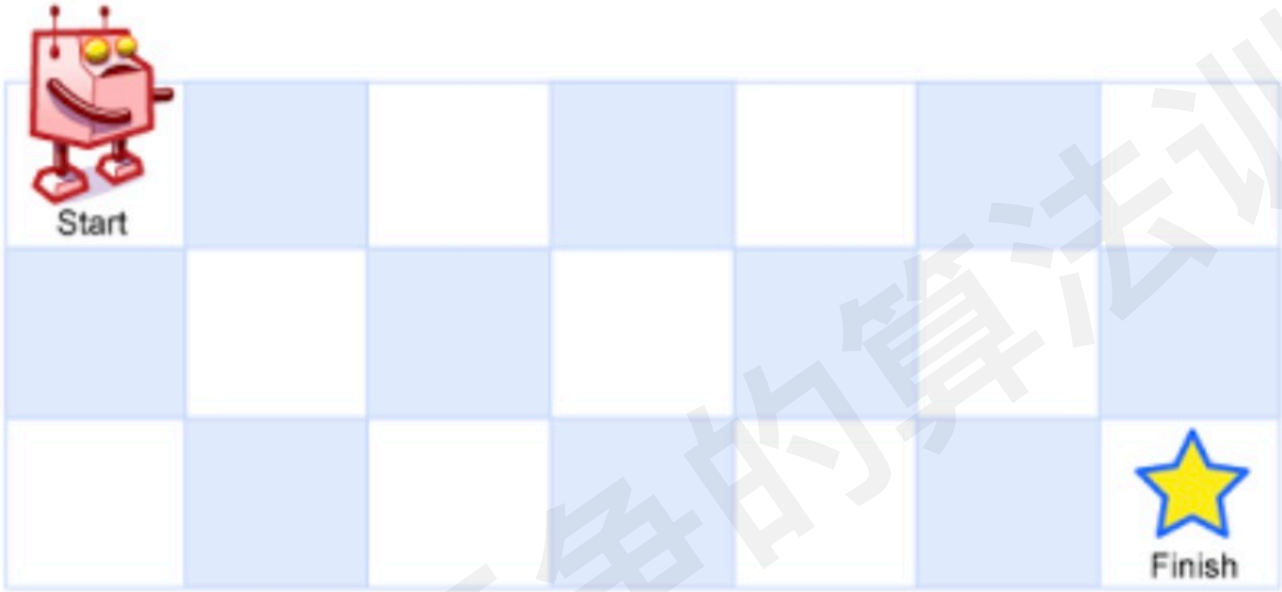
```
class Solution {
    public int uniquePaths(int m, int n) {
    int[][] dp = new int[m][n];
    for (int i = 0; i < m; i++) {
        dp[i][0] = 1;
    }
    for (int i = 0; i < n; i++) {
        dp[0][i] = 1;
    }

    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];
    }
}
```

63. 不同路径 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？



网格中的障碍物和空位置分别用 `1` 和 `0` 来表示。

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

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

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

提问环节