王争的算法训练营

习题课：打家劫舍&股票买卖

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）

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。

动态规划

王争的算法训练营

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

打家劫舍：

198. 打家劫舍

213. 打家劫舍 II

337. 打家劫舍 III (树形DP)

习题课：动态规划

王争的算法训练营

198. 打家劫舍

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 。
```

示例 2：

```
输入：[2,7,9,3,1]
输出：12
解释：偷窃 1 号房屋 （金额 = 2），偷窃 3 号房屋 （金额 = 9），接着偷窃 5 号房屋 （金额 = 1）。
     偷窃到的最高金额 = 2 + 9 + 1 = 12 。
```

提示：

- `1 <= nums.length <= 100`
- `0 <= nums[i] <= 400`

习题课：动态规划

王争的算法训练营

198. 打家劫舍

1、构建多阶段决策模型

n个房屋对应n个阶段，每个阶段决定一个房屋偷还是不偷，两种决策：偷、不偷

2、定义状态

不能只记录每个阶段决策完之后，小偷可偷的最大金额

需要记录不同决策对应的最大金额，也就是：这个房屋偷-对应的最大金额；这个房屋不偷-对应的最大金额

int dp[n][2]记录每个阶段的状态

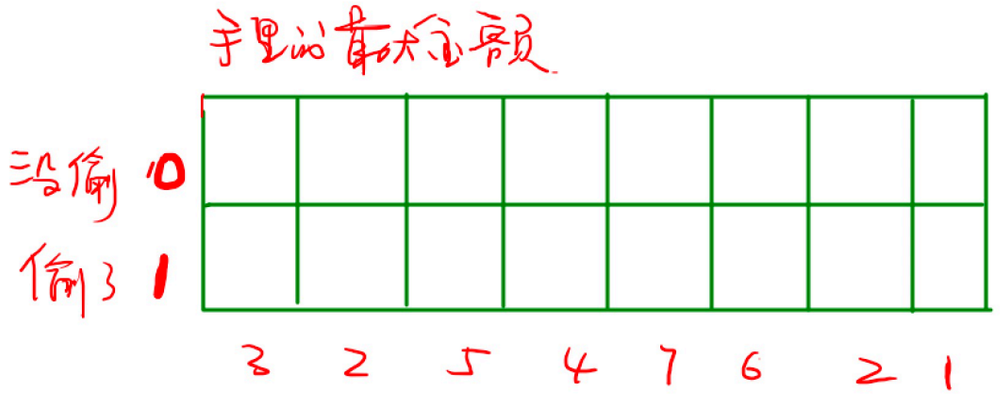
dp[i][0]表示第i个物品不偷，当下的最大金额

dp[i][1]表示第i个物品偷，当下的最大金额

3、定义状态转移方程

dp[i][0] = max(dp[i-1][0], dp[i-1][1])

dp[i][1] = dp[i-1][0] + nums[i]



198. 打家劫舍

```
class Solution {
    public int rob(int[] nums) {
        if (nums.length == 0) return 0;
        int n = nums.length;
        // dp[i][0]表示第i个物品没有选时的最大金额
        // dp[i][1]表示第i个物品选择时的最大金额
        int[][] dp = new int[n][2];
        dp[0][0] = 0;
        dp[0][1] = nums[0];
        for (int i = 1; i < n; ++i) {
            dp[i][0] = Math.max(dp[i-1][0], dp[i-1][1]);
            dp[i][1] = dp[i-1][0] + nums[i];
        }
        return Math.max(dp[n-1][0], dp[n-1][1]);
    }
}
```

习题课：动态规划

王争的算法训练营

213. 打家劫舍 II

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** ，今晚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。
```

示例 2：

```
输入：nums = [1,2,3,1]
输出：4
解释：你可以先偷窃 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 。
```

提示：

- `1 <= nums.length <= 100`
- `0 <= nums[i] <= 1000`

习题课：动态规划

王争的算法训练营

213. 打家劫舍 II

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

        // 第0个不偷窃，偷窃1~n-1之间的房子
        int max1 = rob_dp(nums, 1, n-1);
        // 第0个偷窃，偷窃2~n-2之间的房子
        int max2 = nums[0] + rob_dp(nums, 2, n-2);
        return Math.max(max1, max2);
    }

    private int rob_dp(int[] nums, int p, int r) {
        int n = nums.length;
        // dp[i][0]表示第i个物品没有选时的最大金额
        // dp[i][1]表示第i个物品选择时的最大金额
        int[][] dp = new int[n][2];
        dp[p][0] = 0;
        dp[p][1] = nums[p];
        for (int i = p+1; i <= r; ++i) {
            dp[i][0] = Math.max(dp[i-1][0], dp[i-1][1]);
            dp[i][1] = dp[i-1][0] + nums[i];
        }
        return Math.max(dp[r][0], dp[r][1]);
    }
}
```

习题课：动态规划

王争的算法训练营

337. 打家劫舍 III

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.
```

示例 2:

```
输入: [3,4,5,1,3,null,1]

     3
    / \
   4   5
  / \   \
 1   3   1

输出: 9
解释: 小偷一晚能够盗取的最高金额 = 4 + 5 = 9.
```

习题课：动态规划

王争的算法训练营

198. 打家劫舍

1、构建多阶段决策模型

树形DP基于树这种数据结构上做状态推导，一般都是从下往上推，子节点状态推导父节点状态。一般都是基于后序遍历来实现。

2、定义状态

每个节点有两个状态：偷、不偷

int money[2]表示每个节点的状态

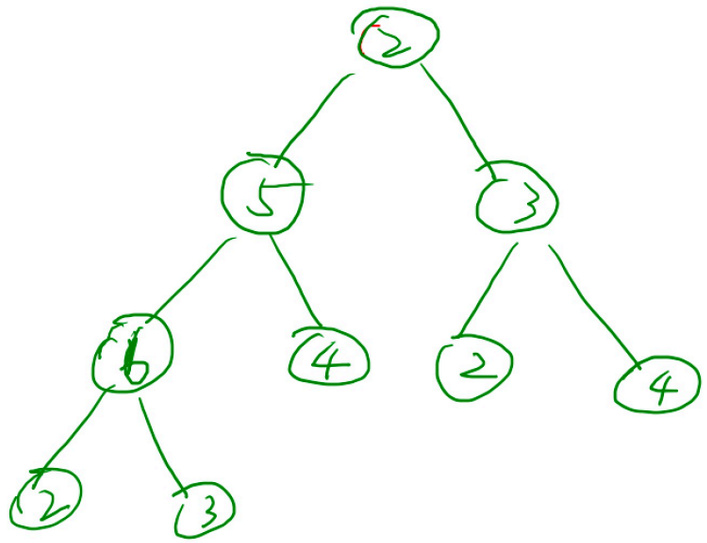
money[0]表示选择不偷此节点，当下最大金额

money[1]表示选择偷此节点，当下的最大金额

3、定义状态转移方程

root.money[0] = max(left.money[0], left.money[1]) +
max(right.money[0], right.money[1])

root.money[1] =left.money[0]+right.money[0]+root.val



习题课：动态规划

337. 打家劫舍 III

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

    private int[] postorder(TreeNode root) {
        if (root == null) {
            return new int[] {0, 0};
        }
        int[] leftMoney = postorder(root.left);
        int[] rightMoney = postorder(root.right);
        int[] money = new int[2];
        money[0] = Math.max(leftMoney[0], leftMoney[1]) + Math.max(rightMoney[0], rightMoney[1]);
        money[1] = (leftMoney[0] + rightMoney[0]) + root.val;
        return money;
    }
}
```

买卖股票：

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

习题课：动态规划

王争的算法训练营

714. 买卖股票的最佳时机含手续

给定一个整数数组 `prices`，其中第 `i` 个元素代表了第 `i` 天的股票价格；非负整数 `fee` 代表了交易股票的手续费用。

你可以无限次地完成交易，但是你每笔交易都需要付手续费。如果你已经购买了一个股票，在卖出它之前你就不能再继续购买股票了。

返回获得利润的最大值。

注意：这里的一笔交易指买入持有并卖出股票的整个过程，每笔交易你只需要为支付一次手续费。

示例 1：

```
输入：prices = [1, 3, 2, 8, 4, 9], fee = 2
输出：8
解释：能够达到的最大利润:
在此处买入 prices[0] = 1
在此处卖出 prices[3] = 8
在此处买入 prices[4] = 4
在此处卖出 prices[5] = 9
总利润: ((8 - 1) - 2) + ((9 - 4) - 2) = 8.
```

注意:

- `0 < prices.length <= 50000`.
- `0 < prices[i] < 50000`.
- `0 <= fee < 50000`.

习题课：动态规划

王争的算法训练营

1、构建多阶段决策模型

n天对应n个阶段，每个阶段决策：买股票、卖股票、不操作

买股票只有当前不持有股票才可

卖股票只有当前持有股票才可

不操作无限制

2、定义状态

每天有两种状态：持有股票、不持有股票。

int dp[n][2]记录每个阶段的状态

dp[i][0]表示第i天持有股票，赚到的最大利润

dp[i][1]表示第i天不持有股票，赚到的最大利润

3、定义状态转移方程

```
dp[i][0] = Math.max(dp[i-1][0], dp[i-1][1]-prices[i]);
dp[i][1] = Math.max(dp[i-1][0]+prices[i]-fee, dp[i-1][1]);
```

714. 买卖股票的最佳时机含手续

```java
class Solution {
    public int maxProfit(int[] prices, int fee) {
        int n = prices.length;
        int[][] dp = new int[n][2];
        dp[0][0] = -prices[0]; // 第i天后持有股票，手里利润的最大值
        dp[0][1] = 0; // 第i天后不持有股票，手里利润的最大值
        for (int i = 1; i < n; ++i) {
            dp[i][0] = Math.max(dp[i-1][0], dp[i-1][1]-prices[i]);
            dp[i][1] = Math.max(dp[i-1][0]+prices[i]-fee, dp[i-1][1]);
        }
        return Math.max(dp[n-1][0], dp[n-1][1]);
    }
}
```

习题课：动态规划

王争的算法训练营

309. 最佳买卖股票时机含冷冻期

给定一个整数数组，其中第 *i* 个元素代表了第 *i* 天的股票价格 。

设计一个算法计算出最大利润。在满足以下约束条件下，你可以尽可能地完成更多的交易（多次买卖一支股票）：

- 你不能同时参与多笔交易（你必须在再次购买前出售掉之前的股票）。
- 卖出股票后，你无法在第二天买入股票 (即冷冻期为 1 天)。

示例:

```
输入: [1,2,3,0,2]
输出: 3
解释: 对应的交易状态为: [买入, 卖出, 冷冻期, 买入, 卖出]
```

习题课：动态规划

1、构建多阶段决策模型

n天对应n个阶段，每个阶段决策：买股票、卖股票、不操作

买股票：前一天不持有股票，并且处于冷冻期或者非冷冻期，而不是刚刚昨天卖掉股票

卖股票：当前持有股票

不操作无限制

2、定义状态

每天有两种状态：持有股票、不持有股票。不持有股票又分为三种小情况。

int dp[n][4]表示n天的状态。

dp[i][0]表示第i天持有股票时的利润

dp[i][1]表示第i天不持有股票时的利润（当天刚卖掉）

dp[i][2]表示第i天不持有股票时的利润（冷冻期），昨天刚卖了股票

dp[i][3]表示第i天不持有股票时的利润（非冷冻期），昨天也没持有

3、定义状态转移方程

```
dp[i][0] = max3(dp[i-1][0], dp[i-1][2]-prices[i], dp[i-1][3]-prices[i]);
dp[i][1] = dp[i-1][0]+prices[i];
dp[i][2] = dp[i-1][1];
dp[i][3] = Math.max(dp[i-1][2], dp[i-1][3]);
```

习题课：动态规划

王争的算法训练营

309. 最佳买卖股票时机含冷冻期

```
class Solution {
    public int maxProfit(int[] prices) {
        if (prices.length == 0) return 0;
        int n = prices.length;
        int[][] dp = new int[n][4];
        // dp[i][0]表示第i天持有股票时的利润
        // dp[i][1]表示第i天不持有股票时的利润（当天刚卖掉）
        // dp[i][2]表示第i天不持有股票时的利润（冷冻期），昨天刚卖了股票
        // dp[i][3]表示第i天不持有股票时的利润（非冷冻期），昨天也没持有
        dp[0][0] = -prices[0];
        dp[0][1] = 0;
        dp[0][2] = 0;
        dp[0][3] = 0;
        for (int i = 1; i < n; ++i) {
            dp[i][0] = max3(dp[i-1][0], dp[i-1][2]-prices[i], dp[i-1][3]-prices[i]);
            dp[i][1] = dp[i-1][0]+prices[i];
            dp[i][2] = dp[i-1][1];
            dp[i][3] = Math.max(dp[i-1][2], dp[i-1][3]);
        }
        return max4(dp[n-1][0], dp[n-1][1], dp[n-1][2], dp[n-1][3]);
    }
}
```

提问环节

关注微信公众号“**小争哥**”，

后台回复“**PDF**”获取独家算法资料