# 0213. 打家劫舍 II

ITCharge 大约 3 分钟

- 标签：数组、动态规划
- 难度：中等

## 题目链接

- 0213. 打家劫舍 II - 力扣

## 题目大意

**描述**：给定一个数组 $nums$，$num[i]$ 代表第 $i$ 间房屋存放的金额，假设房屋可以围成一圈，最后一间房屋跟第一间房屋可以相连。相邻的房屋装有防盗系统，假如相邻的两间房屋同时被偷，系统就会报警。

**要求**：假如你是一名专业的小偷，计算在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。

**说明**：

- $1 \leq nums.length \leq 100$。
- $0 \leq nums[i] \leq 1000$。

**示例**：

- 示例 1：

```py
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：

```py
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。
```

# 解题思路

## 思路 1：动态规划

这道题可以看做是「198. 打家劫舍 」的升级版。

如果房屋数大于等于 3 间，偷窃了第 1 间房屋，则不能偷窃最后一间房屋。同样偷窃了最后一间房屋则不能偷窃第 1 间房屋。

假设总共房屋数量为 $size$，这种情况可以转换为分别求解 $[0, size - 2]$ 和 $[1, size - 1]$ 范围下首尾不相连的房屋所能偷窃的最高金额，然后再取这两种情况下的最大值。而求解 $[0, size - 2]$ 和 $[1, size - 1]$ 范围下首尾不相连的房屋所能偷窃的最高金额问题就跟「198. 打家劫舍 」所求问题一致了。

这里来复习一下「198. 打家劫舍 」 题思路。

#### 1. 划分阶段

按照房屋序号进行阶段划分。

#### 2. 定义状态

定义状态 $dp[i]$ 表示为：前 $i$ 间房屋所能偷窃到的最高金额。

#### 3. 状态转移方程

$i$ 间房屋的最后一个房子是 $nums[i - 1]$。

如果房屋数大于等于 2 间，则偷窃第 $i - 1$ 间房屋的时候，就有两种状态：

1. 偷窃第 $i - 1$ 间房屋，那么第 $i - 2$ 间房屋就不能偷窃了，偷窃的最高金额为：前 $i - 2$ 间房屋的最高总金额 + 第 $i - 1$ 间房屋的金额，即 $dp[i] = dp[i - 2] + nums[i - 1]$；

2. 不偷窃第 $i-1$ 间房屋，那么第 $i-2$ 间房屋可以偷窃，偷窃的最高金额为：前 $i-1$ 间房屋的最高总金额，即 $dp[i] = dp[i-1]$。

然后这两种状态取最大值即可，即状态转移方程为：

$$dp[i] = \begin{cases} nums[0] & i = 1 \\ max(dp[i-2] + nums[i-1], dp[i-1]) & i \geq 2 \end{cases}$$

#### 4. 初始条件

- 前 0 间房屋所能偷窃到的最高金额为 0，即 $dp[0] = 0$。
- 前 1 间房屋所能偷窃到的最高金额为 $nums[0]$，即：$dp[1] = nums[0]$。

#### 5. 最终结果

根据我们之前定义的状态，$dp[i]$ 表示为：前 $i$ 间房屋所能偷窃到的最高金额。假设求解 $[0, size-2]$ 和 $[1, size-1]$ 范围下（$size$ 为总的房屋数）首尾不相连的房屋所能偷窃的最高金额问题分别为 $ans1$、$ans2$，则最终结果为 $max(ans1, ans2)$。

## 思路 1：动态规划代码

py

```
class Solution:
    def helper(self, nums):
        size = len(nums)
        if size == 0:
            return 0

        dp = [0 for _ in range(size + 1)]
        dp[0] = 0
        dp[1] = nums[0]

        for i in range(2, size + 1):
            dp[i] = max(dp[i - 2] + nums[i - 1], dp[i - 1])

        return dp[size]

    def rob(self, nums: List[int]) -> int:
        size = len(nums)
        if size == 1:
```

```
            return nums[0]

        ans1 = self.helper(nums[:size - 1])
        ans2 = self.helper(nums[1:])
        return max(ans1, ans2)
```

## 思路 1：复杂度分析

- **时间复杂度**：$O(n)$。一重循环遍历的时间复杂度为 $O(n)$。
- **空间复杂度**：$O(n)$。用到了一维数组保存状态，所以总体空间复杂度为 $O(n)$。