# 213.打家劫舍Ⅱ - 陈洋

🏷️动态规划数组

**题目描述：**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 。

**示例3：**

输入：nums = [1,2,3] 输出：3

💡 分析：

* 1. 本题是[**198.打家劫舍**](https://leetcode.cn/problems/house-robber/description/)的进阶，本题房间**首尾相连**，故第一间房和最后一间房不能同时偷窃

* 1. 考虑最简单的情况，即只有**一间房屋**，那么偷窃的**最高金额**就是**该房间的金额**

* 1. 考虑**两间房**的情况，能偷窃的**最高金额**就是两间房**金额最高的一间的金额**

* 1. 如果房间数**大于两间，**就得考虑首尾相连的情况，得分情况讨论，n 为**房间总数：**

* + 1. 如果选择**不偷第 1 间房，**那么可以偷窃的房间可为[1, n-1]

* + 1. 如果选择**不偷窃最后一间房**，那么可以偷窃的房间可为[0, n-2]

* 最后，分别计算这两种情况的金额，**二者最大金额**即为可偷窃的最高金额

🏆**动态规划：**

如果使用 dp[i] 来表示前 i 间房**（从 0 开始）**能够偷窃的最高金额，那么**状态转移方程**可为：

dp[i] = max(dp[i-2] + money[i], dp[i-1])

**初始化为：**

* + - * 情况1：选择**不偷第 1 间房，**那么可以偷窃的房间可为[1, n-1]

* + - * 1. dp[0] = 0 -> 无意义

* + - * 1. dp[1] = money[1] -> 表示总共两间房

* + - * 1. dp[2] = max(money[1], money[2]) -> 选择两间房金额最高的一间

* + - * 情况2：选择**不偷窃最后一间房**，那么可以偷窃的房间可为[0, n-2]

* + - * 1. dp[0] = money[0] -> 表示只有一间房

* + - * 1. dp[1] = max(money[0], money[1]) -> 表示只有一间房

**算法伪代码：**

**变量说明：**

nums[] - 存储每个房间的金额

length - 存储 nums 数组的长度

dp[i] - 表示前 i 间**（从 0 开始）**房能够偷窃的最高金额

**输入：**

nums[]

**输出：**

能偷窃的最大金额

**处理：**

Step1: 记录序列的长度，存储在 length 变量中

Step2: 讨论简单情况，进行初始化（只有一间房 or 只有两间房）

- 只有一间房，最高金额为该房间的金额数 输出 nums[0]

- 两间房，选择两间房金额最高的一间的金额输出 输出 max(nums[0], nums[1])

Step3: 分情况进行 for 循环填补状态转移方程

- 不偷第一间房，初始化 dp[1], dp[2] 其中 dp[0] 无意义

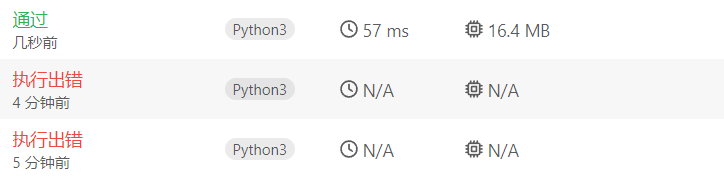
- 不偷最后一间房，初始化 dp[0], dp[1] 其中 dp[0] 无意义

Step4: 输出两种情况的最大值 即为最大金额数

🏅**Python代码**

class Solution:  
 def rob(self, nums: List[int]) -> int:  
 def select(nums: List[int], selection: int) -> int:  
 length = len(nums)  
 if length == 1:  
 return nums[0]  
 if length == 2:  
 return max(nums[0], nums[1])  
 if selection == 1:  
 dp = [0] \* length  
 dp[1],dp[2] = nums[1],max(nums[1], nums[2])  
 for i in range(3, length):  
 dp[i] = max(dp[i-2] + nums[i], dp[i-1])  
 return dp[length-1]  
 if selection == 2:  
 dp = [0] \* length  
 dp[0],dp[1] = nums[0],max(nums[0], nums[1])  
 for i in range(2, length-1):  
 dp[i] = max(dp[i-2] + nums[i], dp[i-1])  
 return dp[length-2]  
 return max(select(nums, 1), select(nums, 2))

**提交结果：**



🔬**时间复杂度分析：**

两次都执行一重 for 循环，所以时间复杂度为 **O(n)**