开始打家劫舍了

198.打家劫舍

题目链接：https://leetcode-cn.com/problems/house-robber/

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 。

> 打家劫舍是dp解决的经典问题,
> 限制因素是相邻的房屋装有相互联通
> 1、确定dp数组下标及其含义
> dp[i]数组表示考虑下标i(包含i)的房屋,最多可以偷窃的金额dp[i]。
>
> 2、递推公式
> 如果偷下标为i的房屋，由于相邻之间不能偷窃，那么只要考虑下标为i-2(包含i-2)的房屋加上nums[i],那么dp[i]=dp[i-2]+nums[i];
> 如果不偷下标为i的房屋，此时要考虑下标i-1(包含i-1)的房屋,那么dp[i]=dp[i-1];
> （**注意这里是考虑，并不是一定要偷i-1房，这是很多同学容易混淆的点**）
> 然后取最大值
> dp[i]=max(dp[i-2]+nums[i],dp[i-1]);
> 3如何初始化

> 由于dp[0]=nums[0];dp[1]=max(dp[0],nums[1]);
> 3、遍历顺序, 从左向右遍历

```
class Solution {
public:
    int rob(vector<int>& nums) {
 vector<int> dp(nums.size()+1,0);
 dp[0]=nums[0];
 dp[1]=max(dp[0],nums[1]);
 for(int i=2;i<nums.size();i++){
 dp[i]=max(dp[i-1],dp[i-2]+nums[i]);
 }
  return dp[nums.size()-1];
     }
};
```

213.打家劫舍II

题目链接：https://leetcode-cn.com/problems/house-robber-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 ，能够偷窃到的最高金额。

示例 1：

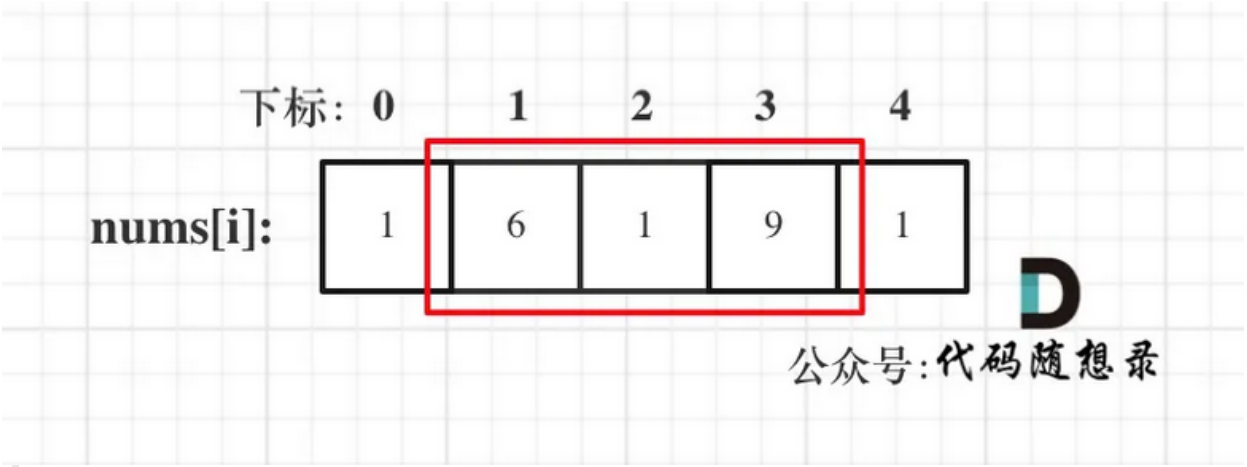
输入：nums = [2,3,2]
输出：3
解释：你不能先偷窃 1 号房屋（金额 = 2），然后偷窃 3 号房屋（金额 = 2），因为他们是相邻的。

> 这一道题与上面一道题的区别就是在于第一间房间与最后一个一个房间是连接的

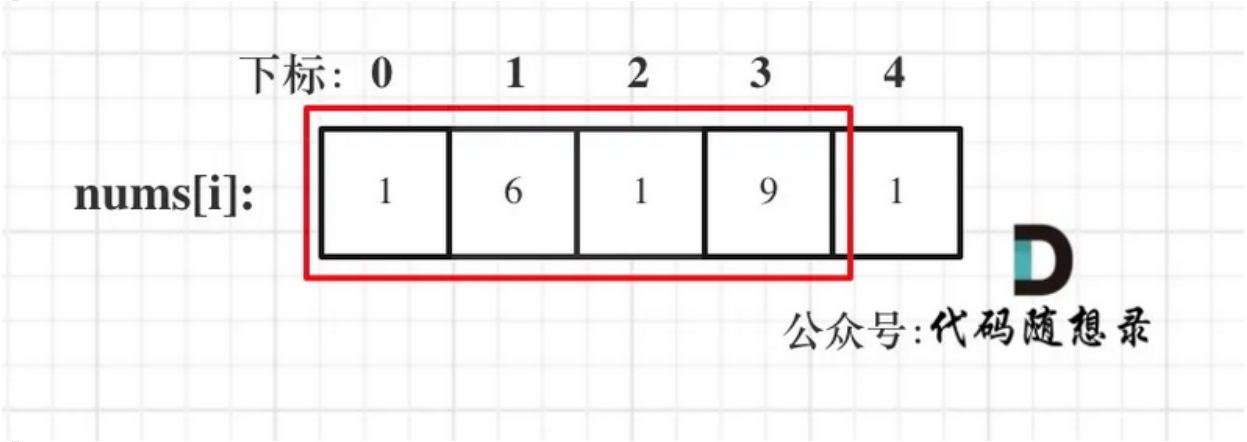
对于一个数组，成环的话主要有三种情况:（考虑的依据是首尾元素不能同时偷）

因此可以根据1、首元素一定不偷尾元素可能偷 2、首元素可能偷尾元素一定不偷 3、首元素和尾元素可能偷三种情况

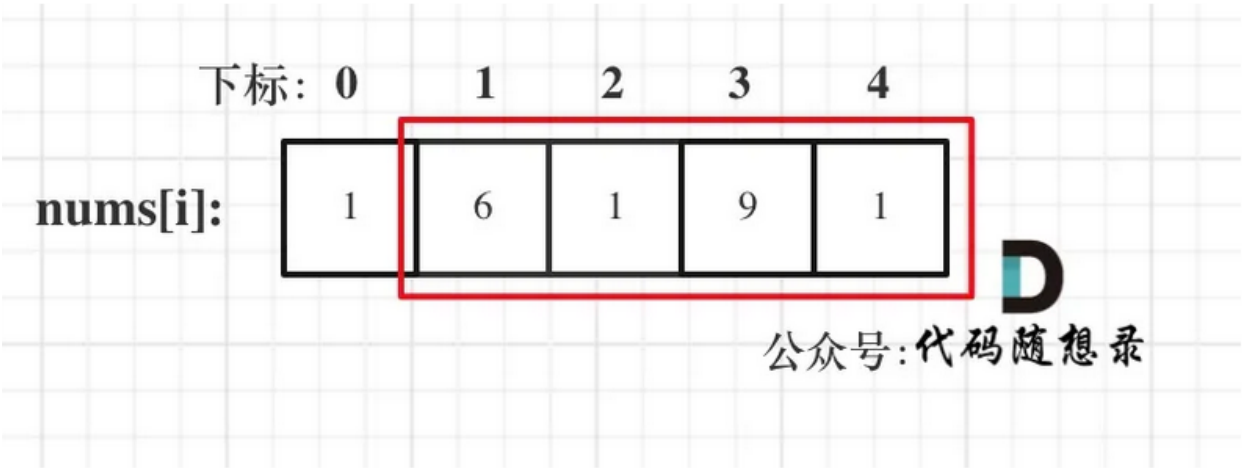
> 情况一：考虑不包含首尾元素



> 情况二:考虑包含首元素，不包含尾元素



> 情况三:考虑包含尾元素,不包含首元素



由于情况二和情况三都包含了情况一，所以只要考虑情况二和情况三即可

综上所述该道题实际上就是对上一篇的题目的分类讨论

```
class Solution {
public:
    int rob(vector<int>& nums) {
  //要考虑两个情况1、尾元素一定不偷2、首元素一定不偷
```

```
5   if(nums.size()==1) return nums[0];
6   int result1=robArray(nums,0,nums.size()-2);
7   int result2=robArray(nums,1,nums.size()-1);
8   return max(result1,result2);
9       }
10   //119题打家劫舍的逻辑,这道题非常秒的地方就是将没学过的知识进行转化
11   int robArray(vector<int>&nums,int start,int end){
12   if(start==end) return nums[start];
13   vector<int> dp(nums.size()+1,0);
14   dp[start]=nums[start];
15   dp[start+1]=max(nums[start],nums[start+1]);
16   for(int i=start+2;i<=end;i++){
17   dp[i]=max(dp[i-1],dp[i-2]+nums[i]);
18   }
19   return dp[end];
20   }
21  };
```

337.打家劫舍 III

题目链接：https://leetcode-cn.com/problems/house-robber-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。

这道题就是在原有打家劫舍的基础上，把屋子的排序建立在二叉树上的一种动态规划
这个时候二叉树的形式以数组的形式存在比较好处理。
本题一定是要用后序遍历的方法,关键问题就是父节点是否要抢如果要抢那么两个孩子就不要抢则考虑要抢孩子的孩子；如果不要抢则考虑抢左右孩子

```
1  /**
2   * Definition for a binary tree node.
3   * struct TreeNode {
4   *     int val;
```

```
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
 * };
 */
class Solution {
public:
    int rob(TreeNode* root) {
 return robFunction(root);
    }
 int robFunction(TreeNode* root){
 //如果向下探寻的没有隔一层那么就只要一个递归结束条件，
 //如果向下探寻隔一层那么递归结束条件就要两个递归结束条件
 if(root==NULL) return 0;
 if(root->left==NULL&&root->right==NULL) return root->val;
 //如果考虑要偷,则考虑孩子的孩子
 int val=root->val;
 //要事先考虑孩子是否存在
 if(root->left) val+=robFunction(root->left->left)+robFunction(root->left->right);
 if(root->right) val+=robFunction(root->right->left)+robFunction(root->right-right);
 //考虑不偷
 return max(val,robFunction(root->left)+robFunction(root->right));
 }
};
```

记忆化递推

> 使用一个map把计算过的结果保存一下,这样如果计算过孙子了，那么计算孩子的时候可以复用孙子节点的结果。

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
```

```
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
 * };
 */
class Solution {
public:
 unordered_map<TreeNode*,int> MAP;
    int rob(TreeNode* root) {
 return robTree(root);
    }
 int robTree(TreeNode* root){
 if(root==NULL) return 0;
 if(root->left==NULL&&root->right==NULL) return root->val;
 if(MAP.find(root)!=MAP.end()) return MAP[root];
 //考虑偷,则考虑左右两个孩子的孩子
 int val=root->val;
 if(root->left)val+=robTree(root->left->left)+robTree(root->left->right);
 if(root->right)val+=robTree(root->right->left)+robTree(root->right->right);
 return max(val,robTree(root->left)+robTree(root->right));
 }
};
```

动态规划方法

这道题算是树形dp的入门题目，因为是在书上进行状态转移，我们在讲解二叉树的时候说过递归三部曲,那么下面我以**递归三部曲为框架**，其中**融合动归五部曲内容来进行讲解**

1、确定递归函数的参数和返回值

这里我们要求一个节点偷与不偷的两个状态所得到的金钱，那么返回值就是一个长度为2的数组

```
vector<int> robTree(TreeNode* cur);
```

所以dp数组以及下标的含义:下标为0记录不偷该节点所得到的最大金钱，下标为1记录偷该节点所得到的的最大金钱。

2、递归终止条件(初始化)

```
if(cur==NULL) return vector<int>{0,0};
```

3、确定遍历顺序

首先确定是**后序遍历算法,**

```
vector<int> left=robTree(cur->left);
vector<int> right=robTree(cur->right);
```

4、单层逻辑

```
//不偷
int nonRob=max(left[0],left[1])+max(right[0],right[1]);
//偷
int Rob=left[0]+right[0]+cur->val;
return vector<int>{nonRob,Rob};
```

```cpp
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
 * };
 */
class Solution {
public:
    int rob(TreeNode* root) {
 vector<int> final=robTree(root);
 return max(final[0],final[1]);
    }
 vector<int> robTree(TreeNode* root){
 if(root==NULL) return vector<int>{0,0};
 //左右子树下的节点价值
 vector<int> left=robTree(root->left);
 vector<int> right=robTree(root->right);
 //偷
 int rob=root->val+left[0]+right[0];
 //不偷
 int nonRob=max(left[0],left[1])+max(right[0],right[1]);
 return vector<int>{nonRob,rob};
 }
};
```

总结:树形dp就是在树上进行递归公式的推导,只不过我们习惯了在一维数组或者二维数组上推导公式，一下子换成了树，就要对树的遍历方式足够了解