<https://leetcode.com/problems/house-robber/description/>

You are a professional robber planning to rob houses along a street. Each house has a certain amount of money stashed, the only constraint stopping you from robbing each of them is that adjacent houses have security systems connected and **it will automatically contact the police if two adjacent houses were broken into on the same night**.

Given an integer array nums representing the amount of money of each house, return *the maximum amount of money you can rob tonight* ***without alerting the police***.

**Example 1:**

Input: nums = [1,2,3,1]

Output: 4

Explanation: Rob house 1 (money = 1) and then rob house 3 (money = 3).

Total amount you can rob = 1 + 3 = 4.

**Example 2:**

Input: nums = [2,7,9,3,1]

Output: 12

Explanation: Rob house 1 (money = 2), rob house 3 (money = 9) and rob house 5 (money = 1).

Total amount you can rob = 2 + 9 + 1 = 12.

**Constraints:**

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

**Attempt 1: 2023-11-15**

**Solution 1: Native DFS (10 min, TLE 55/70)**

class Solution {

public int rob(int[] nums) {

return helper(nums, 0);

}

private int helper(int[] nums, int index) {

if(index >= nums.length) {

return 0;

}

// Rob house

int rob = helper(nums, index + 2) + nums[index];

// Not rob house

int not\_rob = helper(nums, index + 1);

return Math.max(rob, not\_rob);

}

}

Time Complexity: O(2^N)

Space Complexity: O(1)

**Solution 2: DFS + Memoization (10 min)**

class Solution {

public int rob(int[] nums) {

Integer[] memo = new Integer[nums.length + 1];

return helper(nums, 0, memo);

}

private int helper(int[] nums, int index, Integer[] memo) {

if(index >= nums.length) {

return 0;

}

if(memo[index] != null) {

return memo[index];

}

// Rob house

int rob = helper(nums, index + 2, memo) + nums[index];

// Not rob house

int not\_rob = helper(nums, index + 1, memo);

return memo[index] = Math.max(rob, not\_rob);

}

}

Time Complexity: O(N)

Space Complexity: O(N)

**Solution 3: DP Fibonacci Sequence (30 min)**

class Solution {

public int rob(int[] nums) {

if(nums.length < 2) {

return nums[0];

}

int n = nums.length;

// 没必要n + 1因为index就算到达n也是dp[n] = 0，

// 基于DFS中的底是index >= n时return 0

int[] dp = new int[n];

// 标准顶底之术，顶 index = 0，底 index >= n

// 那么要实现正确的推导，和0/1 knapsack的模式不一样

// 这里需要两个启示参数来启动推导，因为第一个参数直接

// 就有两种选择，也就是这里的nums[n - 1]和nums[n - 2]，

// 而dp[i]代表从index = n - 1到index = i的最大值，

// 所以dp[n - 1] = nums[n - 1]因为你可以选择不抢劫

// index = n - 1的房子也可以选择抢劫，不抢劫获得0，抢劫

// 获得nums[n - 1]，最大值就是nums[n - 1]，而对于dp[n - 2]

// 就不同了，你可以选择跳过index = n - 1的房子抢劫index

// = n - 2的房子，获得nums[n - 1]，也可以因为选择抢劫

// index = n - 1的房子而跳过index = n - 2的房子，那么

// 最大的获取值就必须比较nums[n - 1]和nums[n - 2]的大小

// 了，所以dp[n - 2] = Math.max(nums[n - 1], nums[n - 2])

dp[n - 1] = nums[n - 1];

dp[n - 2] = Math.max(nums[n - 1], nums[n - 2]);

// 而从倒数第三个位置index = n - 3开始就可以使用Fibonacci了

for(int i = n - 3; i >= 0; i--) {

dp[i] = Math.max(dp[i + 2] + nums[i], dp[i + 1]);

}

return dp[0];

}

}

Time Complexity: O(N)

Space Complexity: O(N)

**Solution 4: DP Fibonacci Sequence + Space Optimization (30 min)**

class Solution {

public int rob(int[] nums) {

if(nums.length < 2) {

return nums[0];

}

int n = nums.length;

// ... cur prev prev2

// ^ ^ ^

// cur prev prev2

// int cur = 0; --> wrong !!

int prev2 = nums[n - 1];

int prev = Math.max(nums[n - 1], nums[n - 2]);

// 'cur' must assign value after 'prev',

// test out by [1,1], expect 1, output 0

int cur = prev;

for(int i = n - 3; i >= 0; i--) {

cur = Math.max(prev2 + nums[i], prev);

prev2 = prev;

prev = cur;

}

return cur;

}

}

Time Complexity: O(N)

Space Complexity: O(1)

**Refer to**

<https://leetcode.com/problems/house-robber/solutions/1605133/c-discussing-all-solutions-dp-with-constant-space/>

This is a classic 1D-DP problem where at every step we have a choice to make ...So the first and foremost thing in any DP problem is to find the recurrence relation !! At every ith house robber has 2 options: *a) rob current house i.* *b) don't rob current house.*

* In case he is robbing the (i)th house, the money he can get till the i-th house == money robbed till (i-2)th house + money robbed at (i)th house....let's say total money robbed in this case equals to X.
* In case he is not robbing, money robbed till i-th house==money robbed till (i-1)th house...lets say total money robbed in this case equals to Y.
* So , the max money he gets till i-th house is the max(X,Y).

Example of case (a) --> nums={2,3,2} ... Here, the robber will rob the house at index-2 as nums[index-2] + nums[index-0] > nums[index-1]Example of case (b)--> nums={2,7,3} ... here maximum money robbed till index-2 will not be equal to nums[index-2] + nums[index-0]... as nums[index-1] is greater than the sum of money at both those houses ...

*We can achieve the desired solution to this problem via multiple ways, let's start with the simpler ones and then will look forward to optimize the* ***Time and Space Complexities***

* **Simple Recursion**
* Time Complexity : O ( 2^n ) Gives us TLE
* Space Complexity : O( 1 )

class Solution {

public:

int rec(vector<int>& nums,int idx){

if(idx>=nums.size())return 0;

return max(nums[idx]+rec(nums,idx+2),rec(nums,idx+1));

}

int rob(vector<int>& nums) {

return rec(nums,0);

}

};

* **Memoization**
* Time Complexity : O (n)
* Space Complexity : O(n)

class Solution {

public:

int rec(vector<int>& nums,int idx,vector<int>&dp){

if(idx >= nums.size()) return 0;

if(dp[idx] != -1) return dp[idx];

return dp[idx] = max(rec(nums, idx+1, dp), nums[idx] + rec(nums, idx+2, dp));

}

int rob(vector<int>& nums) {

vector<int>dp(nums.size()+1,-1);

return rec(nums,0,dp);

}

};

* **Dynamic Programming**
* Time Complexity : O(n)
* Space Complexity : O(n)

class Solution {

public:

int rob(vector<int>& nums) {

if(nums.size()==1)return nums[0];

vector<int>dp(nums.size());

dp[0]=nums[0];

dp[1]=max(nums[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];

}

};

* **Dynamic Programming** (improved version)
* Time Complexity : O(n)
* Space complexity : O(1)

We can observe that the above dp solution relied only on the previous two indices in dp to compute the value of current dp[i]. So, we dont really need to maintain the whole dp array and can instead just maintain the values of previous index (denoted as **prev** below) and previous-to-previous index (denoted as **prev2**) and we can calculate the value for current index (**cur**) using just these two variables and roll-forward each time.

We can optimize the space now, as we can see we only need to know the answer till (i-1)th idx and (i-2)th idx to have an answer for the (i)th idx. And we don't really care about the whole dp-vector, so there is no point maintaining one... 3 variables will do the job, prev\_ans (for i-1), prev\_ans2(for i-2) and curr\_ans !!

* dp[i - 2] - previous answer(till i-2 idx) -> prev\_ans2
* dp[i - 1] - previous answer(till i-1 idx) -> prev\_ans
* dp[i] - current answer(can be prev\_ans or prev\_ans2+nums[i]) -> curr\_ans
* So we will get rid of the dp-vector and thus our SC will get reduced down to O(1)
* (It is very similar to Fibonacci series Space optimization)

class Solution {

public:

int rob(vector<int>& nums) {

int n = nums.size();

if(n == 1) return nums[0];

int prev\_ans2=nums[0], prev\_ans=max(nums[0],nums[1]),curr\_ans=prev\_ans;

for(int i = 2; i < n; i++){

curr\_ans = max(prev\_ans, prev\_ans2 + nums[i]);

prev\_ans2 = prev\_ans;

prev\_ans = curr\_ans;

}

return curr\_ans;

}

};