<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: 2025-06-01**

**Native DFS void return wrong slution**

**nums = [1,2,3,1]**

**Output = 3**

**Expected = 4**

class Solution {

int maxAmount = 0;

public int rob(int[] nums) {

helper(nums, 0, false);

return maxAmount;

}

private void helper(int[] nums, int index, boolean prevRobbed) {

if(index >= nums.length) {

return;

}

int curMax = 0;

// Do not rob current house

helper(nums, index + 1, false);

// Rob current house

if(!prevRobbed) {

maxAmount = Math.max(maxAmount, curMax + nums[index]);

helper(nums, index + 1, true);

}

}

}

**Solution 1: Native DFS (30 min, TLE 55/70)**

**Style 1 - 1: void return, similar to** [L2320.Count Number of Ways to Place Houses (Ref.L70,L198)](note://WEBe7b6043ebce73029e2e8890edb8e11af) **style**

class Solution {

    int maxAmount = 0;

    public int rob(int[] nums) {

        helper(nums, 0, false, 0);

        return maxAmount;

    }

    private void helper(int[] nums, int index, boolean prevRobbed, int curMax) {

        if(index >= nums.length) {

            maxAmount = Math.max(maxAmount, curMax);

            return;

        }

        // Do not rob current house

        helper(nums, index + 1, false, curMax);

        // Rob current house

        if(!prevRobbed) {

            helper(nums, index + 1, true, curMax + nums[index]);

        }

    }

}

Time Complexity: O(2^n)

Space Complexity: O(n)

**Refer to Deepseek**

Your solution has a few issues that prevent it from correctly calculating the maximum amount. Let me explain the problems and how to fix them:

**Issues in Your Current Solution**

**curMax is always 0**: **You're not tracking the accumulated sum of robbed houses.**

**No sum accumulation**: **You're not passing the running total through recursive calls.**

**Base case doesn't update max**: **Your base case just returns without updating the maximum.**

**Fixed Solution**

Here's the corrected version that properly tracks the accumulated sum:

class Solution {

private int maxAmount = 0;

public int rob(int[] nums) {

helper(nums, 0, false, 0);

return maxAmount;

}

private void helper(int[] nums, int index, boolean prevRobbed, int currentSum) {

if (index >= nums.length) {

maxAmount = Math.max(maxAmount, currentSum);

return;

}

// Option 1: Don't rob current house

helper(nums, index + 1, false, currentSum);

// Option 2: Rob current house (only if previous wasn't robbed)

if (!prevRobbed) {

helper(nums, index + 1, true, currentSum + nums[index]);

}

}

}

**Key Fixes**

**Added currentSum parameter**:

Tracks the total amount robbed so far in the current path

Passed through recursive calls

**Proper base case**:

Updates maxAmount when reaching the end of the array

**Correct recursive calls**:

When robbing, adds current house's value to currentSum

When skipping, keeps currentSum unchanged

**How It Works for nums = [1,2,3,1]**

The algorithm explores all valid combinations:

Rob 0 → Skip 1 → Rob 2 → Skip 3: 1 + 3 = 4

Rob 0 → Skip 1 → Skip 2 → Rob 3: 1 + 1 = 2

Skip 0 → Rob 1 → Skip 2 → Rob 3: 2 + 1 = 3

Skip 0 → Skip 1 → Rob 2 → Skip 3: 3

And other combinations...

The maximum found is 4 (correct answer).

**Style 1 - 2: void return (don't need boolean variable 'prevRobbed')**

class Solution {

    int maxAmount = 0;

    public int rob(int[] nums) {

        helper(nums, 0, 0);

        return maxAmount;

    }

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

        if(index >= nums.length) {

            maxAmount = Math.max(maxAmount, curMax);

            return;

        }

        // Do not rob current house

        helper(nums, index + 1, curMax);

        // Rob current house

        helper(nums, index + 2, curMax + nums[index]);

    }

}

Time Complexity: O(2^n)

Space Complexity: O(n)

**Refer to Deepseek**

Here's the **void return style DFS solution** for the House Robber problem, where we track the maximum amount robbed using a class variable instead of returning values from the recursive function:

**Void-Style DFS Solution**

class Solution {

private int maxRobbed = 0;

public int rob(int[] nums) {

dfs(nums, 0, 0);

return maxRobbed;

}

private void dfs(int[] nums, int index, int currentSum) {

// Base case: no more houses to consider

if (index >= nums.length) {

maxRobbed = Math.max(maxRobbed, currentSum);

return;

}

// Option 1: Rob current house (add to sum and skip next)

dfs(nums, index + 2, currentSum + nums[index]);

// Option 2: Skip current house (keep sum and move to next)

dfs(nums, index + 1, currentSum);

}

}

**Key Differences from Return-Style DFS**

**Class Variable**:

maxRobbed tracks the global maximum instead of returning values.

**Accumulated Sum**:

currentSum carries the running total of robbed money.

**Base Case Update**:

When we reach the end of the array, we update maxRobbed.

**How It Works**

**Rob Current House**:

Add nums[index] to currentSum.

Jump to index + 2 (skip adjacent house).

**Skip Current House**:

Keep currentSum unchanged.

Move to index + 1.

**Example Walkthrough (nums = [1, 2, 3])**

|  |  |  |
| --- | --- | --- |
| Path | currentSum | Updates  maxRobbed |
| Rob 0 → Rob 2 | 1 + 3 = 4 | 4 |
| Rob 0 → Skip 2 | 1 | (no update) |
| Skip 0 → Rob 1 → Skip 3 | 2 | 2 |
| Skip 0 → Skip 1 → Rob 2 | 3 | 4 |

Final maxRobbed = 4 (correct answer).

**Optimization Note**

This void-style DFS is **less efficient** than the return-style version because:

It doesn't prune redundant paths early.

It explores all possible combinations (O(2^n) time).

For better performance, use **memoization** or switch to the **DP solution** for large inputs.

This version is primarily for understanding the brute-force decision space!

**Style 2: int return (which capable to convert into Memoization & DP solution)**

class Solution {

    public int rob(int[] nums) {

        return helper(nums, 0);

    }

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

        if(index >= nums.length) {

            return 0;

        }

        // Do not rob current house

        int notRob = helper(nums, index + 1);

        // Rob current house

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

        return Math.max(notRob, rob);

    }

}

Time Complexity: O(2^n)

Space Complexity: O(n)

**Refer to Deepseek**

Here's an intuitive **pure DFS solution** for the House Robber problem. While this approach is less efficient (O(2^n) time complexity), it helps build a foundational understanding of the problem before optimizing with DP.

**Pure DFS Approach**

**Problem Breakdown**:

For each house, we have two choices:

Rob it (can't rob the next house)

Skip it (can consider the next house)

We need to explore all possible valid combinations of these choices.

**DFS Strategy**:

At each step, recursively explore both choices (rob/skip).

Track the maximum amount robbed so far.

**Solution Code (Pure DFS)**

class Solution {

public int rob(int[] nums) {

return dfs(nums, 0);

}

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

if (index >= nums.length) {

return 0;

}

// Option 1: Rob current house and skip next

int robCurrent = nums[index] + dfs(nums, index + 2);

// Option 2: Skip current house and consider next

int skipCurrent = dfs(nums, index + 1);

// Return the maximum of the two options

return Math.max(robCurrent, skipCurrent);

}

}

**Explanation**

**Base Case**:

If index exceeds the array bounds (index >= nums.length), return 0 (no more houses to rob).

**Recursive Cases**:

**Rob Current House**:

Add nums[index] to the total.

Move to index + 2 (skip the next house).

**Skip Current House**:

Move to index + 1 (consider the next house).

**Result**:

The function returns the maximum value between robbing or skipping the current house.

**Why This Works**

The DFS explores all possible valid paths (no two adjacent houses robbed).

It's essentially a **binary decision tree** where each level represents a house, and branches represent rob/skip choices.

**Limitations**

**Time Complexity**: O(2^n) - Exponential due to repeated calculations (e.g., dfs(nums, 3) might be called multiple times).

**Space Complexity**: O(n) - Recursion stack depth.

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

**Style 1 - 1 and Style 1 - 2 cannot convert into Memoization**

The void-return DFS solution you provided **cannot be directly memoized** in its current form because of how it tracks results. Here's why and how to adapt it for memoization:

**Why It Can't Be Directly Memoized**

**Side Effects**: It uses maxAmount as a global variable that gets modified during traversal.

**No Return Values**: Memoization typically works by caching return values of pure functions.

**Current Sum Tracking**: The curMax parameter makes each call unique, preventing effective caching.

**Style 3: int return memo**

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];

        }

        // Do not rob current house

        int notRob = helper(nums, index + 1, memo);

        // Rob current house

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

        return memo[index] = Math.max(notRob, rob);

    }

}

Time Complexity: O(n)

Space Complexity: O(n)

**Refer to Deepseek**

To fix the inefficiency, we can memoize results of subproblems:

class Solution {

private Integer[] memo;

public int rob(int[] nums) {

memo = new Integer[nums.length];

return dfs(nums, 0);

}

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

if (index >= nums.length) return 0;

if (memo[index] != null) return memo[index];

int robCurrent = nums[index] + dfs(nums, index + 2);

int skipCurrent = dfs(nums, index + 1);

return memo[index] = Math.max(robCurrent, skipCurrent);

}

}

**Time Complexity**: O(n) - Each subproblem is solved once.

**Space Complexity**: O(n) - For memoization array and recursion stack.

**Solution 3: DP (10 min)**

**Style 1: Mapping logic for Native DFS**

class Solution {

    public int rob(int[] nums) {

        int len = nums.length;

        if(len == 1) {

            return nums[0];

        }

        // dp[i] represents the maximum amount that can be robbed up to house i.

        // so dp[len - 1] has i = len - 1, it mapping to nums[i] as nums[len - 1]

        int[] dp = new int[len];

        dp[len - 1] = nums[len - 1];

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

        for(int i = len - 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)

**Style 2: Reverse loop direction of Style 1**

class Solution {

    public int rob(int[] nums) {

        int len = nums.length;

        if(len == 1) {

            return nums[0];

        }

        // dp[i] represents the maximum amount that can be robbed up to house i.

        int[] dp = new int[len];

        dp[0] = nums[0];

        dp[1] = Math.max(nums[0], nums[1]);

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

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

        }

        return dp[len - 1];

    }

}

Time Complexity: O(n)

Space Complexity: O(n)

**Solution 4: Space Optimized DP (10 min)**

**Style 1: Mapping logic for Native DFS**

class Solution {

    public int rob(int[] nums) {

        int len = nums.length;

        if(len == 1) {

            return nums[0];

        }

        int a = nums[len - 1];

        int b = Math.max(nums[len - 1], nums[len - 2]);

        //          c     b    a

        //  new\_c new\_b new\_a

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

            int c = Math.max(a + nums[i], b);

            a = b;

            b = c;

        }

        return b;

    }

}

Time Complexity: O(n)

Space Complexity: O(1)

**Style 2: Reverse loop direction of Style 1**

class Solution {

    public int rob(int[] nums) {

        int len = nums.length;

        if(len == 1) {

            return nums[0];

        }

        int a = nums[0];

        int b = Math.max(nums[0], nums[1]);

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

            int c = Math.max(a + nums[i], b);

            a = b;

            b = c;

        }

        return b;

    }

}

Time Complexity: O(n)

Space Complexity: O(1)

**Style 2 refer to Deepseek for both DP and Space Optimized DP solutions**

Here's a clean and efficient Java solution for LeetCode 198 (House Robber) using dynamic programming:

**Problem Analysis**

You're a robber planning to rob houses along a street where each house has a certain amount of money. The constraint is that you cannot rob two adjacent houses. Determine the maximum amount you can rob tonight without alerting the police.

**Approach**

We'll use dynamic programming to solve this problem:

**State Definition**: dp[i] represents the maximum amount that can be robbed up to house i.

**Transition**:

Rob house i: nums[i] + dp[i-2] (can't rob house i-1)

Skip house i: dp[i-1]

**Base Cases**:

dp[0] = nums[0] (only one house)

dp[1] = max(nums[0], nums[1]) (choose between first two houses)

**Solution Code**

class Solution {

public int rob(int[] nums) {

if (nums == null || nums.length == 0) return 0;

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

int n = nums.length;

int[] dp = new int[n];

dp[0] = nums[0];

dp[1] = Math.max(nums[0], nums[1]);

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

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

}

return dp[n-1];

}

}

**Space Optimization**

We can reduce space complexity from O(n) to O(1) by only keeping track of the last two values:

class Solution {

public int rob(int[] nums) {

if (nums == null || nums.length == 0) return 0;

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

int prev2 = nums[0];

int prev1 = Math.max(nums[0], nums[1]);

for (int i = 2; i < nums.length; i++) {

int current = Math.max(prev1, nums[i] + prev2);

prev2 = prev1;

prev1 = current;

}

return prev1;

}

}

**Explanation**

**Initialization**:

Handle edge cases (empty array or single house)

Initialize dp[0] and dp[1] with base cases

**DP Transition**:

For each house i, choose between:

Robbing it (add to dp[i-2])

Skipping it (take dp[i-1])

**Result**:

The last element dp[n-1] contains the maximum amount

**Complexity**

**Time**: O(n) - Single pass through the array

**Space**: O(1) optimized version - Only two variables needed

This solution efficiently computes the maximum robbery amount while respecting the adjacency constraint.

**Style 1 refer to below for both DP and Space Optimized DP solutions**

**DP**

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

// 这里需要两个起始参数来启动推导，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];

}

}

**Space Optimized DP**

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;

}

}

**Refer to**

[L213.House Robber II (Ref.L198)](note://A5D6119BDEB54084A860F30A71A0A8C2)

[L337.House Robber III (Ref.L198,L213)](note://WEBdfa028bc04290ef6bce0fdf40c470cfe)

[L2320.Count Number of Ways to Place Houses (Ref.L70,L198)](note://WEBe7b6043ebce73029e2e8890edb8e11af)

[L2560.House Robber IV (Ref.L11,L198)](note://WEB4bc49b4b30992ee8ff1d6f950fa72a1f)