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

You are a professional robber planning to rob houses along a street. Each house has a certain amount of money stashed. All houses at this place are **arranged in a circle.** That means the first house is the neighbor of the last one. Meanwhile, adjacent houses have a security system 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 = [2,3,2]

Output: 3

Explanation: You cannot rob house 1 (money = 2) and then rob house 3 (money = 2), because they are adjacent houses.

**Example 2:**

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 3:**

Input: nums = [1,2,3]

Output: 3

**Constraints:**

1 <= nums.length <= 100

0 <= nums[i] <= 1000

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

**Solution 1: Native DFS (10 min, TLE 62/75)**

class Solution {

    public int rob(int[] nums) {

        int n = nums.length;

        // Test out by: nums = {1}, expected = 1, output = 0

        if(n < 2) {

            return nums[0];

        }

        // Rob house from 0 to n - 2

        int count1 = helper(nums, 0, n - 2, 0);

        // Rob house from 1 to n - 1

        int count2 = helper(nums, 1, n - 1, 1);

        return Math.max(count1, count2);

    }

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

        if(index < start || index > end) {

            return 0;

        }

        // Rob the room

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

        // Not rob the room

        int not\_rob = helper(nums, start, end, 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) {

        int n = nums.length;

        // Test out by: nums = {1}, expected = 1, output = 0

        if(n < 2) {

            return nums[0];

        }

        // Be careful, we need 2 memoization array here, these two case

        // should calculate fully separated, since for method1, it only

        // working against index 0 to n - 2, for method2, it only working

        // aginst index 1 to n - 1, if mix two memo together, we not able

        // to identify two cases are independent, since index = 0 and

        // index = n - 1 as concatenate indexes (in circular case here)

        // are stored in same memoization array as memo[0] and memo[n - 1],

        // but since rule as index = 0 and index = n - 1 should not happen

        // in circular case, we cannot store memo[0] and memo[n - 1] in

        // same memoization array, instead, we create two memoization arrays,

        // one for memo[0] to memo[n - 2], the other for memo[1] to memo[n - 1]

        // e.g it can be test out by [2, 1, 1, 2] expected as max = 3 not 4

        Integer[] memo1 = new Integer[n];

        Integer[] memo2 = new Integer[n];

        // Rob house from 0 to n - 2

        int count1 = helper(nums, 0, n - 2, 0, memo1);

        // Rob house from 1 to n - 1

        int count2 = helper(nums, 1, n - 1, 1, memo2);

        return Math.max(count1, count2);

    }

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

        if(index < start || index > end) {

            return 0;

        }

        if(memo[index] != null) {

            return memo[index];

        }

        // Rob the room

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

        // Not rob the room

        int not\_rob = helper(nums, start, end, 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)**

**Its similar than L198. House Robber DP Solution, we just have to add a range [start, end] on L198 template, and L198 became an auxiliary 'helper(...)' method, calling twice**

class Solution {

    public int rob(int[] nums) {

        int n = nums.length;

        if(n < 2) {

            return nums[0];

        }

        int count1 = helper(nums, 0, n - 2);

        int count2 = helper(nums, 1, n - 1);

        return Math.max(count1, count2);

    }

    private int helper(int[] nums, int start, int end) {

        // Test out by: [4,1,2], expect = 4, output = 2

        // we need to handle end - start == 1 case, have to

        // find larger one by Math.max(nums[start], nums[end])

        //if(end - start < 2) {

        //    return nums[end];

        //}

        if(end - start == 1) {

            return Math.max(nums[start], nums[end]);

        }

        if(end == start) {

            return nums[end];

        }

        int[] dp = new int[nums.length];

        dp[end] = nums[end];

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

        for(int i = end - 2; i >= start; i--) {

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

        }

        return dp[start];

    }

}

Time Complexity: O(N)

Space Complexity: O(N)

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

class Solution {

    public int rob(int[] nums) {

        int n = nums.length;

        if(n < 2) {

            return nums[0];

        }

        int count1 = helper(nums, 0, n - 2);

        int count2 = helper(nums, 1, n - 1);

        return Math.max(count1, count2);

    }

    private int helper(int[] nums, int start, int end) {

        // Test out by: [4,1,2], expect = 4, output = 2

        // we need to handle end - start == 1 case, have to

        // find larger one by Math.max(nums[start], nums[end])

        //if(end - start < 2) {

        //    return nums[end];

        //}

        if(end - start == 1) {

            return Math.max(nums[start], nums[end]);

        }

        if(end == start) {

            return nums[end];

        }

        // ... cur  prev  prev2

        //      ^    ^    ^

        // cur prev prev2

        int prev2 = nums[end];

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

        int cur = prev;

        for(int i = end - 2; i >= start; i--) {

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

            prev2 = prev;

            prev = cur;

        }

        return cur;

    }

}

Time Complexity: O(N)

Space Complexity: O(1)

**Attempt 2: 2025-06-04**

**Solution 1: Native DFS (10 min, TLE 62/75)**

**Style 1: int return but similar to** [L2320.Count Number of Ways to Place Houses (Ref.L70,L198)](note://WEBe7b6043ebce73029e2e8890edb8e11af) **which has 'prevRobbed'**

class Solution {

    public int rob(int[] nums) {

        if(nums.length == 1) {

            return nums[0];

        }

        // Case 1: Rob houses 0 to n-2 (exclude last house)

        int case1 = helper(nums, 0, nums.length - 2, 0, false);

        // Case 2: Rob houses 1 to n-1 (exclude first house)

        int case2 = helper(nums, 1, nums.length - 1, 0, false);

        return Math.max(case1, case2);

    }

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

        // Base case: reached end of considered houses

        if (index > end) {

            return currentSum;

        }

        // Option 1: Skip current house

        int skip = helper(nums, index + 1, end, currentSum, false);

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

        int rob = 0;

        if (!prevRobbed) {

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

        }

        return Math.max(skip, rob);

    }

}

Time Complexity: O(2^n)

Space Complexity: O(n)

**Style 2: int return without 'prevRobbed'**

class Solution {

    public int rob(int[] nums) {

        if(nums.length == 1) {

            return nums[0];

        }

        // Case 1: Rob houses 0 to n-2 (exclude last house)

        int case1 = helper(nums, 0, nums.length - 2);

        // Case 2: Rob houses 1 to n-1 (exclude first house)

        int case2 = helper(nums, 1, nums.length - 1);

        return Math.max(case1, case2);

    }

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

        // Base case: reached end of considered houses

        if(index > end) {

            return 0;

        }

        // Option 1: Skip current house

        int skip = helper(nums, index + 1, end);

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

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

        return Math.max(skip, rob);

    }

}

Time Complexity: O(2^n)

Space Complexity: O(n)

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

**Style 1: with 'prevRobbed'**

class Solution {

    public int rob(int[] nums) {

        if(nums.length == 1) {

            return nums[0];

        }

        // Case 1: Rob houses 0 to n-2 (exclude last house)

         Integer[][] memo1 = new Integer[nums.length][2];

        int case1 = helper(nums, 0, nums.length - 2, 0, memo1);

        // Case 2: Rob houses 1 to n-1 (exclude first house)

         Integer[][] memo2 = new Integer[nums.length][2];

        int case2 = helper(nums, 1, nums.length - 1, 0, memo2);

        return Math.max(case1, case2);

    }

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

        // Base case: reached end of considered houses

        if (index > end) {

            return 0;

        }

        if(memo[index][prevRobbed] != null) {

            return memo[index][prevRobbed];

        }

        // Option 1: Skip current house

        int skip = helper(nums, index + 1, end, 0, memo);

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

        int rob = 0;

        if (prevRobbed == 0) {

            rob = helper(nums, index + 1, end, 1, memo) + nums[index];

        }

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

    }

}

Time Complexity: O(n)

Space Complexity: O(n)

**Style 2: without 'prevRobbed'**

class Solution {

    public int rob(int[] nums) {

        if(nums.length == 1) {

            return nums[0];

        }

        // Case 1: Rob houses 0 to n-2 (exclude last house)

        Integer[] memo1 = new Integer[nums.length];

        int case1 = helper(nums, 0, nums.length - 2, memo1);

        // Case 2: Rob houses 1 to n-1 (exclude first house)

        Integer[] memo2 = new Integer[nums.length];

        int case2 = helper(nums, 1, nums.length - 1, memo2);

        return Math.max(case1, case2);

    }

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

        // Base case: reached end of considered houses

        if(index > end) {

            return 0;

        }

        if(memo[index] != null) {

            return memo[index];

        }

        // Option 1: Skip current house

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

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

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

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

    }

}

Time Complexity: O(n)

Space Complexity: O(n)

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

class Solution {

    public int rob(int[] nums) {

        if(nums.length == 1) {

            return nums[0];

        }

        int case1 = helper(nums, 0, nums.length - 2);

        int case2 = helper(nums, 1, nums.length - 1);

        return Math.max(case1, case2);

    }

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

        if(index == end) {

            return nums[end];

        }

        int[] dp = new int[nums.length];

        dp[end] = nums[end];

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

        for(int i = nums.length - 3; i >= 0; i--) {

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

        }

        return dp[index];

    }

}

Time Complexity: O(n)

Space Complexity: O(n)

**Refer to Deepseek**

**Standard DP (Without Space Optimization)**

class Solution {

public int rob(int[] nums) {

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

// Case 1: Rob houses 0 to n-2

int case1 = robHelper(nums, 0, nums.length - 2);

// Case 2: Rob houses 1 to n-1

int case2 = robHelper(nums, 1, nums.length - 1);

return Math.max(case1, case2);

}

private int robHelper(int[] nums, int start, int end) {

if (start == end) return nums[start];

int[] dp = new int[nums.length];

dp[start] = nums[start];

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

for (int i = start + 2; i <= end; i++) {

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

}

return dp[end];

}

}

**Compare to above Standard DP (Without Space Optimization) solution, the DP solution i am writing based on previous DFS solution, is my DP solution logically more like deduced from the DFS solution than your DP solution?**

Your observation is correct! Let me analyze the relationship between these solutions:

**Your DP Solution vs. My DP Solution**

**Your DP Solution (Reverse Order)**:

for(int i = nums.length - 3; i >= 0; i--) {

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

}

**Direct Mapping to DFS**: This perfectly mirrors your recursive DFS logic:

dp[i + 1] corresponds to helper(nums, index + 1, end) (skip)

dp[i + 2] + nums[i] corresponds to nums[index] + helper(nums, index + 2, end) (rob)

**Direction**: Processes houses from right to left, exactly matching the DFS's "look ahead" approach

**My DP Solution (Forward Order)**:

for (int i = start + 2; i <= end; i++) {

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

}

**Traditional DP**: More conventional left-to-right processing

**Same Logic**: Still represents skip/rob choices but looks backward instead of forward

**Key Comparison**

|  |  |  |
| --- | --- | --- |
| Aspect | Your DP Solution | My DP Solution |
| **Direction** | Right-to-left (matches DFS) | Left-to-right (traditional) |
| **Base Case** | Starts from end of array | Starts from beginning |
| **DFS Similarity** | Direct 1:1 mapping | Logical equivalent |
| **Intuitiveness** | More natural for DFS conversion | More conventional |

**Why Yours is More DFS-Like**

**Recursive to Iterative**:

Your DFS makes decisions looking forward (index+1, index+2)

Your DP solution does the same with dp[i+1] and dp[i+2]

**Natural Conversion**:

Each dp[i] in your solution directly represents "maximum from this point onward"

Exactly what the DFS computes at each recursive call

**Base Case Alignment**:

Your DP initializes from the end (like DFS base case index > end)

My DP initializes from the start

**Which is Better?**

**Your Version**: More intuitive if coming from DFS, clearer mapping to recursive logic

**My Version**: Slightly better for left-to-right processing (more conventional in DP problems)

**Performance**: Identical O(n) time and space complexity

Both are correct - yours is indeed a more direct translation of the DFS approach! The choice depends on whether you prefer thinking forward (DFS-like) or backward (traditional DP).

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

class Solution {

    public int rob(int[] nums) {

        if(nums.length == 1) {

            return nums[0];

        }

        int case1 = helper(nums, 0, nums.length - 2);

        int case2 = helper(nums, 1, nums.length - 1);

        return Math.max(case1, case2);

    }

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

        if(index == end) {

            return nums[end];

        }

        int a = nums[end]; // dp[i + 2]

        int b = Math.max(nums[end - 1], nums[end]); // dp[i]

        //            c      b     a

        //   new\_c  new\_b  new\_a

        for(int i = end - 2; i >= index; i--) {

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

            a = b;

            b = c;

        }

        return b;

    }

}

Time Complexity: O(n)

Space Complexity: O(1)

**Refer to**

[L198.House Robber (Ref.L213,L337,L2320,L2560)](note://WEB3a5c4401c63ae93266b0b603cb20076c)

[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)