<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)