# 213. House Robber II

**Medium** Array Dynamic Programming Leetcode Link

## Problem Description

In this problem, we imagine a situation where a professional thief is planning to rob houses. The houses are laid out in a circle, and each house contains a certain amount of money that can be stolen. However, if the robber tries to rob two adjacent houses, an alarm will be triggered and the police will be notified. The objective is to determine the maximum amount of money that can be stolen without triggering any alarms.

Constraints include:

- The arrangement of the houses in a circle, which means that the first and last houses are considered adjacent.
- One cannot rob two adjacent houses on the same night.

Given the circular arrangement, we need to find a way to calculate the maximum loot without robbing adjacent houses and also accounting for the circular layout.

## Intuition

The problem can be seen as a variation of the classic dynamic programming problem "House Robber", with the additional constraint imposed by the circular arrangement of the houses.

The intuition for solving this problem is to use dynamic programming to make decisions at each step. We need to figure out, with each house, whether it's more profitable to rob it or skip it. However, the circular arrangement makes it tricky; we cannot simply start from one end and go to the other, as we may miss out on the optimal solution that involves wrapping around the end.

To resolve this issue with circular arrangement, we make use of the following observations:

- If we rob the first house, we cannot rob the last house due to the circular layout.
- If we ignore the first house, we can treat the problem as a non-circular arrangement and apply standard dynamic programming as in the original "House Robber" problem.

So our approach is:

1. Calculate the maximum money that can be robbed bypassing the first house (making the problem linear instead of circular).
2. Calculate the maximum money that can be robbed if we skip the last house for the same reason.

After calculating these two scenarios, the maximum of both will give us the solution to the original circular problem. The function `_rob` implements this approach of robbing or skipping a house. It iterates through the given list of house values and, at each step, decides whether to rob the current house or not. The choice is made based on the maximum money that can be gained considering the previous decisions.

The `rob` function then calls the helper function `_rob` twice – once for the array of houses excluding the first house and once for the array excluding the last house – and returns the maximum of the two results. This accommodates the circular constraint by considering both possibilities – starting from the first house or skipping it.

In summary, the solution splits the circular problem into two linear sub-problems and applies dynamic programming to each. The maximum of the solutions to these sub-problems is the answer to the original problem.

## Solution Approach

The solution provided uses dynamic programming, a method for solving complex problems by breaking them down into simpler sub-problems. It is particularly well-suited for optimization problems like ours, where we seek the maximum amount of money that can be robbed.

Dynamic programming often involves creating an array to store the sub-problems' solutions; however, in this implementation, only two variables are maintained, reducing the space complexity. The algorithm employs a "rob-or-skip" strategy, where we look at each house and decide whether to rob it or skip it based on which action would yield a higher profit. Here's how the variables `f` and `g` correspond to this strategy:

- `f`: The maximum amount of money that can be robbed up to the current house without robbing it (essentially, the skip option).
- `g`: The maximum amount of money that can be robbed including the current house (which is the sum of the money in the current house and the maximum amount robbed up to the previous house where we did not rob, to ensure no two adjacent houses are robbed).

The main function `rob` distinguishes between two cases due to the circular layout of houses. If there's only one house, it simply returns its value, as there's no constraint to consider. In other cases, the function utilizes a helper function `_rob`. The `_rob` function implements the dynamic programming approach and is applied to two slices of the original list: one excluding the first house and one excluding the last house.

Let's understand how `_rob` works:

1. Initialize `f`, `g` to `0`. These are the maximum amounts with the aforementioned conditions.
2. Loop through each value `x` in the `nums` list, where `x` represents the amount of money in the current house:
   - Calculate the new value of `f` as the maximum between the old `f` and `g`. This choice represents skipping the current house, so we take the best previous outcome.
   - Update `g` to be the sum of the old `f` (which represents the total money robbed up to the previous house without robbing it) and `x` (the amount in the current house). This represents robbing the current house.
3. After the loop, the function returns the maximum amount between `f` and `g` which represents the maximum money the robber can achieve by either robbing or not robbing the last house in the list.

Back in the `rob` function, we then return the maximum value obtained from calling `_rob` with either of the list slices, thus concluding the approach and ensuring we consider the circular property of the problem.

The time complexity of the solution is O(n), where n is the number of houses, because it processes each house exactly once. The space complexity is O(1), because it uses a constant amount of extra space, despite the input size.

## Example Walkthrough

Let's use a small example to illustrate the solution approach. Suppose we have a circular arrangement of houses with the following amounts of money: `[2, 3, 2]`.

The thief has the following options:

1. Rob the first house (with `2`), and then can only rob the third house (with `2`), totaling `4`.
2. Skip the first house, rob the second house (with `3`), and then they have to skip the third house because they can’t rob adjacent houses.

Applying the two scenarios from our solution approach:

1. If we ignore the first house, we treat the remaining houses `[3, 2]` as a linear problem. Using our dynamic programming approach:

   - Start with `f` and `g` both at `0`.
   - For the first house (with `3`), we update `g` to `0 + 3` (since `f` is still `0`). Now `f` is `0` and `g` is `3`.
   - For the second house (with `2`), we update `f` to the maximum of previous `f` and `g`, which is `3`. We then set `g` to the previous `f` value plus the current house's value, which is `0 + 2`. Now `f` is `3` and `g` is `2`.
   - The maximum of `f` and `g` is `3`, which is the most we can rob from houses `[3, 2]`.
2. If we ignore the last house, we treat the remaining houses `[2, 3]` as a linear problem. Again, we apply dynamic programming:

   - Start with `f` and `g` both at `0`.
   - For the first house (with `2`), we update `g` to `0 + 2`. Now `f` is `0` and `g` is `2`.
   - For the second house (with `3`), we update `f` to the maximum of previous `f` and `g`, which is `2`. We then set `g` to the previous `f` value plus the current house's value, which is `0 + 3`. Now `f` is `2` and `g` is `3`.
   - The maximum of `f` and `g` is `3`, the most we can rob from houses `[2, 3]`.

After comparing the two scenarios, we see the maximum money the thief can rob without triggering any alarms is `3`. Thus, the robber robs the second house with 3 in both cases, and the solution to the problem is 3. This demonstrates how the circular constraint is handled by considering the problem in a linear fashion for two slices of the array.

## Python Solution

```python
class Solution:
    def rob(self, nums: List[int]) -> int:
        def _rob_subsequence(sub_nums):
            """
            Calculate the maximum amount of money that can be
            robbed from a subsequence of houses (sub_nums).
            """
            prev_max = curr_max = 0
            for value in sub_nums:
                # Update the prev_max and curr_max to include
                # the current house in the robbery plan or not.
                prev_max, curr_max = curr_max, max(curr_max, prev_max + value)
            # The current maximum will be the solution for this subsequence.
            return curr_max

        # If there's only one house, return its value.
        if len(nums) == 1:
            return nums[0]

        # Rob the houses by considering two scenarios:
        # 1. Excluding the first house.
        # 2. Excluding the last house.
        # Take the maximum from these two options.
        return max(_rob_subsequence(nums[1:]), _rob_subsequence(nums[:-1]))

```

## Java Solution

```java
class Solution {
    // The main function to compute the maximum amount of money that can be robbed.
    public int rob(int[] nums) {
        int houseCount = nums.length;

        // If there's only one house, the maximum money is what's in that house.
        if (houseCount == 1) {
            return nums[0];
        }

        // Otherwise, find the maximum of two scenarios: excluding the first house or excluding the last house.
        return Math.max(robMaxMoney(nums, 0, houseCount - 2), robMaxMoney(nums, 1, houseCount - 1));
    }

    // Helper function to calculate the max money that can be robbed in a specific range of houses.
    private int robMaxMoney(int[] nums, int left, int right) {
        // Initialize two variables to keep track of the two previous maximum values.
        int inclPrev = 0; // This will hold the maximum amount including the previous house.
        int exclPrev = 0; // This will hold the maximum amount excluding the previous house.

        // Iterate through the range of houses.
        for (int i = left; i <= right; ++i) {
            // Calculate the new maximum including the current house.
            int inclCurr = exclPrev + nums[i];

            // Calculate the new maximum excluding the current house by taking the maximum between inclPrev and exclPrev.
            exclPrev = Math.max(inclPrev, exclPrev);

            // Update inclPrev to be used in the next iteration.
            inclPrev = inclCurr;
        }

        // Return the maximum money that can be robbed within the specified range.
        // Must consider the final value of inclPrev and exclPrev, since one of them will have the maximum.
        return Math.max(inclPrev, exclPrev);
    }
}

```

## C++ Solution

```cpp
class Solution {
public:
    // Main function to calculate the maximum amount of money that can be robbed.
    int rob(vector<int>& nums) {
        int n = nums.size();
        // If there's only one house, return its value.
        if (n == 1) {
            return nums[0];
        }
        // Compare and return the maximum amount between robbing the first house or the second one.
        return max(robRange(nums, 0, n - 2), robRange(nums, 1, n - 1));
    }

```

```
    // Helper function to calculate max amount from a range of houses.
    int robRange(vector<int>& nums, int left, int right) {
        int inclusive = 0; // This will store the max amount including the current house.
        int exclusive = 0; // This will store the max amount excluding the current house.

        // Iterate from the left to the right indices of the house range.
        for (; left <= right; ++left) {
            // Compute the new exclusive amount, which is the max of the previous inclusive and exclusive amounts.
            int newExclusive = max(inclusive, exclusive);
            // Update inclusive to be the sum of the old exclusive and current house value.
            inclusive = exclusive + nums[left];
            // Update exclusive to the newly computed value.
            exclusive = newExclusive;
        }
        // Return the maximum of inclusive and exclusive amounts as the result.
        return max(inclusive, exclusive);
    }
};

```

## Typescript Solution

```
function rob(nums: number[]): number {
    // Determine the length of the array.
    const numLength = nums.length;

    // Edge case: If there's only one house, return its value.
    if (numLength === 1) {
        return nums[0];
    }

    // Helper function to rob a range of houses from index l to r, inclusive.
    const robRange = (start: number, end: number): number => {
        let robPrevious = 0; // Initialize to store the maximum amount that can be robbed up to the previous house
        let robCurrent = 0;  // Initialize to store the maximum amount that can be robbed up to the current house

        // Iterate from the start index to the end index, inclusive.
        for (; start <= end; ++start) {
            let robNext = Math.max(robPrevious, robCurrent); // The next 'previous' is the max of current and previous robbed amounts
            robCurrent = robPrevious + nums[start];          // Update the current to the sum of previous robbed amount and current h
            robPrevious = robNext;                           // Move the 'next' value to 'previous' for the next iteration.
        }

        // Return the maximum amount robbed between the last two houses.
        return Math.max(robPrevious, robCurrent);
    };

    // The maximum amount robbed will be the maximum of either robbing from the first house to the second-to-last house,
    // or from the second house to the last one since we cannot rob adjacent houses.
    return Math.max(robRange(0, numLength - 2), robRange(1, numLength - 1));
}

```

## Time and Space Complexity

The given Python code defines a method `rob` that solves the house robber problem for a neighborhood arranged in a circle, meaning the first and last houses cannot be robbed together.

### Time Complexity

The time complexity of the helper function `_rob` is O(n), where n is the number of elements in `nums`. It iterates through the list once and performs a constant time operation of `max(f, g)` and an addition operation `f + x` for each element.

The `rob` function calls the `_rob` function twice: once with `nums[1:]` and once with `nums[:-1]`. Since slicing a list is O(n) and the `_rob` function is also O(n), the total time complexity for each call is O(2n). Because the two calls do not depend on each other, the overall time complexity of `rob` remains O(n).

Final time complexity: `O(n)`

### Space Complexity

The helper function `_rob` uses O(1) extra space, as it only keeps track of the two variables `f` and `g`.

However, the main function `rob` creates two slices of the `nums` input list: `nums[1:]` and `nums[:-1]`. Each of these slices is of size `n-1`, where `n` is the length of the original `nums`. The space taken up by these lists is significant.

Final space complexity: `O(n)`, accounting for the space used by the list slices.