198. House Robber

Medium | Array | Dynamic Programming | Leetcode Link

Problem Description

In this problem, you take on the role of a professional burglar who is aiming to rob houses lined up along a street. Every house has some money, but there's a twist: the houses are equipped with connected security systems. If you rob two adjacent houses on the same night, the security system will detect the break-ins and alert the police. Your goal is to figure out the maximum amount of money you can steal tonight without triggering any alarms.

So, you're presented with an array of integers `nums`, where `nums[i]` represents the amount of money stashed in the `i`-th house. The challenge is to find the optimal combination of houses to rob that will maximize your total loot, keeping in mind that you cannot rob adjacent houses.

It's a classic optimization problem that requires you to look for the best strategy to maximize your gains without breaking the constraint (the alarm system in this case).

Intuition

To get to the heart of this problem, let's discuss the intuition behind the solution.

The main idea here is dynamic programming, which involves breaking the problem down into smaller subproblems and building up a solution from the answers to those subproblems.

Now, consider the following choices at each house: either you rob it or you don't. If you decide to rob house `i`, you cannot rob house `i-1`, but you are free to rob house `i-2` and before. If you decide not to rob house `i`, your best robbery amount up to house `i` is the same as if you were standing at house `i-1`.

Let's define two variables: `f` and `g`. We'll use `f` to track the maximum amount we can rob up to the current house if we don't rob this house, and `g` to track the maximum amount if we do rob it.

We update `f` like this: We take the max between the previous `f` (not robbing the previous house) and `g` (robbing the previous house) because for the current house, we are not robbing it, so we are free to choose the maximum loot collected from the previous two states.

On the other hand, we update `g` by adding the current house's money to the previous `f`, because if we decide to rob this house, we can't rob the previous one, so we add to `f` (the max money robbed without including the previous house).

Lastly, we have to return the maximum amount we can rob including or excluding the last house, which means we return the max between `f` and `g`.

The elegance of this solution lies in its simplicity and the fact that it takes advantage of overlapping subproblems, typical in dynamic programming solutions, and calculates the result in a bottom-up manner.

Solution Approach

The solution uses a dynamic programming approach, which is a method for solving complex problems by breaking them down into simpler subproblems. It computes solutions to these subproblems just once and stores their results—typically in an array—to avoid the need for redundant calculations. However, in this particular problem, we don't even need an array; we only use two variables to store the results since we move through the `nums` array sequentially and only require information from the previous step.

Here's a step-by-step explanation of the code that was provided:

1. Initialize two variables, `f` and `g`. `f` represents the maximum amount of money that can be robbed up to the current house without robbing it, and `g` represents the maximum amount with robbing it.

```
f = g = 0
```

2. Iterate over each amount `x` in the `nums` array representing the money at each house.

```
for x in nums:
```

3. Update the values of `f` and `g` for each house:

   - `f` is updated to the maximum of the previous `f` or `g`. This makes `f` the maximum amount of money we could have if we chose not to rob this house (`i`). We can use the maximum from the previous state since robbing the current house is out of the question.

```
f = max(f, g)
```

   - `g` is updated by adding the current house's stash `x` to the previous non-robbed maximum `f`. This is because if we rob the current house, we must skip the previous one, hence we add to the previous `f`.

```
g = f + x
```

   Within the for loop, this updates `f` and `g` at each house while maintaining the condition that we never rob two adjacent houses.

4. After the loop ends, the maximum amount of money that can be robbed will be the maximum of `f` and `g`.

```
return max(f, g)
```

It's important to note that `f` and `g` are being used to keep the space complexity of the solution at O(1), meaning it doesn't require any extra space proportional to the input size. This algorithm overall has O(n) time complexity since it requires one pass through the `nums` array, where `n` is the number of elements in the array.

In conclusion, the implementation keeps track of two possibilities at each house (to rob or not to rob) and makes decisions that ensure the highest amount of money robbed without ever triggering the alarm system.

Example Walkthrough

To illustrate the solution approach, let's use a small example where the `nums` array representing money at each house is `[2, 7, 9, 3, 1]`.

Here's the step-by-step walkthrough:

1. Initialize `f` and `g` to `0`. These variables will hold the maximum money that can be robbed so far without and with robbing the current house, respectively.
2. Start iterating over the array `nums`.
   - For the first house (`nums[0] = 2`), we can only rob it since there's no previous house. Therefore, `f` remains `0` as we are not robbing this house, and `g` is updated to `f + nums[0]`, which is also `2`.
3. Move to the second house (`nums[1] = 7`):
   - Update `f` to `max(f, g)`, which was from the previous iteration. Here `f` becomes `max(0, 2)` so `f` is now `2`.
   - Update `g` to `f + nums[1]`, which is `2 + 7`, so `g` becomes `9`.
4. At the third house (`nums[2] = 9`):
   - Update `f` to `max(2, 9)`, so `f` is now `9`.
   - Update `g` to `f (previous f before updating which is 2) + nums[2]`, which is `2 + 9`, so `g` is `11`.
5. At the fourth house (`nums[3] = 3`):
   - Update `f` to `max(9, 11)`, so `f` is now `11`.
   - Update `g` to `f (previous f before updating which is 9) + nums[3]`, which is `9 + 3`, so `g` is `12`.
6. Finally, at the fifth house (`nums[4] = 1`):
   - Update `f` to `max(11, 12)`, so `f` is now `12`.
   - Update `g` to `f (previous f before updating which is 11) + nums[4]`, which is `11 + 1`, so `g` becomes `12`.
7. The loop has finished, and we now take the maximum of `f` and `g`. Both `f` and `g` are `12`, so the maximum money that can be robbed is `12`.

If we map out the houses we chose to rob based on `g`'s updates, we robbed the second and fourth houses - a total of `7 + 3 = 10`, and not the third and fifth as the walk through suggests. Let's correct the steps for the fourth and fifth houses.

For the fourth house, since we can't rob it (because we robbed the third house):

- `f` remains `11` since we're not adding the value of the current house.
- `g` doesn't change since we can't rob this house directly after the third house.

For the fifth house:

- `f` remains `11`, and
- `g` is updated with `11 + 1` (the previous `f` plus the current house's value), resulting in `g` being `12`.

In this case, we robbed the third and fifth houses (2nd house was not actually robbed), getting a total of `9 + 1 = 10`. Hence, the final maximum amount we could rob is `12`, updating our final `g` with the amount from the fifth house and maintaining the rule of not robbing two adjacent houses.

Python Solution

```python
from typing import List

class Solution:
    def rob(self, nums: List[int]) -> int:
        # Initialize two variables to store the maximum amount of money that can be
        # robbed without adjacent thefts:
        # prev_no_rob denotes the max amount considering the previous house wasn't robbed
        # prev_robbed denotes the max amount considering the previous house was robbed
        prev_no_rob, prev_robbed = 0, 0

        # Iterate over all the houses
        for current_house_value in nums:
            # Calculate the new value for prev_no_rob by choosing the max between
            # prev_no_rob and prev_robbed. This is because if the current house is not
            # being robbed, then the maximum amount is the best of either not robbing
            # the previous house or robbing it.
            # The value for prev_robbed is updated to the sum of prev_no_rob and
            # the value of the current house, as we can only rob the current house if
            # the previous one was not robbed.
            temp = max(prev_no_rob, prev_robbed)
            prev_robbed = prev_no_rob + current_house_value
            prev_no_rob = temp

        # Return the max amount that can be robbed - either the amount if the last house
        # was not robbed or if it was.
        return max(prev_no_rob, prev_robbed)

```

Java Solution

```java
class Solution {
    public int rob(int[] nums) {
        // f represents the max profit we can get from the previous house
        int prevNoRob = 0;
        // g represents the max profit we can get if we rob the current house
        int prevRob = 0;

        // Iterate over all the houses in the array
        for (int currentHouseValue : nums) {
            // Store max profit of robbing/not robbing the previous house
            int tempPrevNoRob = Math.max(prevNoRob, prevRob);

            // If we rob the current house, we cannot rob the previous one
            // hence our current profit is previous house's no-rob profit + current house value
            prevRob = prevNoRob + currentHouseValue;

            // Update the previous no-rob profit to be the best of robbing or not robbing the last house
            prevNoRob = tempPrevNoRob;
        }

        // Return the max profit we can get from the last house,
        // regardless of whether we rob it or not
        return Math.max(prevNoRob, prevRob);
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <algorithm> // For std::max

class Solution {
public:
    // Function to calculate the maximum amount of money that can be robbed
    int rob(vector<int>& nums) {
        // Initialize previous and beforePrevious to store the max rob amounts
        // at two adjacent houses we may potentially skip or rob
        int previous = 0, beforePrevious = 0;

        // Loop through each house represented in the nums vector
        for (int num : nums) {
            // Temporarily hold the maximum amount robbed so far
            int temp = max(previous, beforePrevious);

            // Update beforePrevious to the new amount robbed including the current house
            beforePrevious = previous + num;

            // Update previous to the amount stored in temp
            previous = temp;
        }

        // Return the maximum amount that can be robbed without alerting the police
        // which is the maximum of previous and beforePrevious after considering all houses
        return max(previous, beforePrevious);
    }
};

```

Typescript Solution

```typescript
// Function to determine the maximum amount of money that can be robbed
// without robbing two adjacent houses.
function rob(nums: number[]): number {
    // Initialize two variables:
    // robPrevious - the maximum amount robbed up to the previous house
    // robCurrent - the maximum amount robbed up to the current house
    let [robPrevious, robCurrent] = [0, 0];

    for (const currentHouseValue of nums) {
        // Compute the new maximum excluding and including the current house:
        // The new robPrevious becomes the maximum of either
        // robPrevious or robCurrent from the previous iteration.
        // robCurrent is updated with the sum of robPrevious (excludes the previous house)
        // and the value of the current house, representing the choice to rob this house.
        [robPrevious, robCurrent] = [Math.max(robPrevious, robCurrent), robPrevious + currentHouseValue];
    }

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

```

Time and Space Complexity

Time Complexity

The given code consists of a single for-loop that iterates through every element in the input list `nums`. Inside this loop, the calculations involve simple arithmetic operations and a single `max` function call which both have a constant time complexity (`O(1)`). Since these operations are performed once for each element in the list, the time complexity is directly proportional to the length of the list. Thus, the time complexity of the code is `O(n)`, where `n` is the number of elements in `nums`.

Space Complexity

The algorithm uses two variables (`f` and `g`) to keep track of the calculations at each step. No additional data structures that grow with the input size are used. This means that the space complexity is not dependent on the size of the input and is, therefore, constant. Hence, the space complexity of the code is `O(1)`.