Example: House Robber

This is the first of 6 articles where we will use a framework to work through example DP problems. The framework provides a blueprint to solve DP problems, but when you are just starting to learn DP, deriving some of the logic yourself may be difficult. The objective of these articles is to talk through how to use the framework to work through each problem, and our goal is that, by the end of this, you will be able to independently tackle most DP problems using this framework.

In this article, we will be looking at the [House Robber](https://leetcode.com/problems/house-robber/) problem.

|  |
| --- |
| 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 |

In an earlier section of this explore card, we talked about how House Robber fits the characteristics of a DP problem. It's asking for the maximum of something, and our current decisions will affect which options are available for our future decisions. Let's see how we can use the framework to develop an algorithm for this problem.

1. A **function or array** that answers the problem for a given state.

First, we need to decide on **state variables**. As a reminder, state variables should be fully capable of describing a scenario. Imagine if you had this scenario in real life - you're a robber and you have a lineup of houses. If you are at one of the houses, the only variable you would need to describe your situation is an integer - the index of the house you are currently at. Therefore, the only state variable is an integer, say i, that indicates the index of a house.

If the problem had an added constraint such as "you are only allowed to rob up to k houses", then k would be another necessary state variable. This is because being at, say house 4 with 3 robberies left is different than being at house 4 with 5 robberies left.

You may be wondering - why don't we include a state variable that is a boolean indicating if we robbed the previous house or not? We certainly could include this state variable, but we can develop our recurrence relation in a way that makes it unnecessary. Building an intuition for this is difficult at first, but it becomes easier with practice.

The problem is asking for "the maximum amount of money you can rob". Therefore, we would use either a function dp(i) that returns the maximum amount of money you can rob up to and including house i, or an array dp where dp[i] represents the maximum amount of money you can rob up to and including house i.

This means that after all the subproblems have been solved dp[i] and dp(i) both return the answer to the original problem for the subarray of nums that spans 0 to i inclusive. To solve the original problem, we will just need to return dp[nums.length - 1] or dp(nums.length - 1), depending if we do bottom-up or top-down.

1. A **recurrence relation** to transition between states.

For this part, let's assume we are using a top-down (recursive function) approach. Note that the top-down approach is closer to our natural way of thinking and it is generally easier to think of the recurrence relation if we start with a top-down approach.

Next, we need to find a recurrence relation, which is typically the hardest part of the problem. For any recurrence relation, **a good place to start is to think about a general state** (in this case, let's say we're at the house at index i), and use information from the problem description to think about how other states relate to the current one.

If we are at some house, logically, we have 2 options: we can choose to rob this house, or we can choose to not rob this house.

1. If we decide not to rob the house, then we don't gain any money. Whatever money we had from the previous house is how much money we will have at this house - which is dp(i - 1).
2. If we decide to rob the house, then we gain nums[i] money. However, this is only possible if we did not rob the previous house. This means the money we had when arriving at this house is the money we had from the previous house without robbing it, which would be however much money we had 2 houses ago, dp(i - 2). After robbing the current house, we will have dp(i - 2) + nums[i] money.

From these two options, we always want to pick the one that gives us maximum profits. Putting it together, we have our recurrence relation:

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

1. **Base cases.**

The last thing we need is base cases so that our recurrence relation knows when to stop. The base cases are often found from clues in the problem description or found using logical thinking. In this problem, if there is only one house, then the most money we can make is by robbing the house (the alternative is to not rob the house). If there are only two houses, then the most money we can make is by robbing the house with more money (since we have to choose between them). Therefore, our base cases are:

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

# Top-down Implementation

Now that we have established all 3 parts of the framework, let's put it together for the final result. Remember: we need to memoize the function!

|  |
| --- |
| class Solution {  *map*<int, int> mp; *vector*<int> temp;  public:  int dp(int i) {  if (i == 0) return temp[0];  if (i == 1) return *max*(temp[0], temp[1]);  if (mp.*find*(i)==mp.*end*()) {  mp[i] = *max*(dp(i - 1), dp(i - 2) + temp[i]);  }  return mp[i];  }  int rob(*vector*<int>& nums) {  temp = nums;  return dp(nums.*size*() - 1);  }  };  int main() {  *vector*<int> nums = { 2,7,9,3,1 };  Solution aSolution;  *cout* << aSolution.rob(nums) << *endl*;  return 0;  } |

# Bottom-up Implementation

Here's the bottom-up approach: everything is the same, except that we use an array instead of a hash map and we iterate using a for-loop instead of using recursion.

|  |
| --- |
| class Solution {  public:  int rob(*vector*<int>& nums) {  if (nums.*size*() == 1) return nums[0];  int\* dp = new int[nums.*size*()];  // Base cases  dp[0] = nums[0];  dp[1] = *max*(nums[0], nums[1]);  for (int i = 2; i < nums.*size*(); i++) {  dp[i] = *max*(dp[i - 1], dp[i - 2] + nums[i]); // Recurrence relation  }  return dp[nums.*size*() - 1];  }  };  int main() {  *vector*<int> nums = { 2,7,9,3,1 };  Solution aSolution;  *cout* << aSolution.rob(nums) << *endl*;  return 0;  } |

For both implementations, the time and space complexity is *O*(*n*). We'll talk about time and space complexity of DP algorithms in depth at the end of this chapter. Here's an animation that shows the algorithm in action:



