**Example 198. House Robber**

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.

2. 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]) .

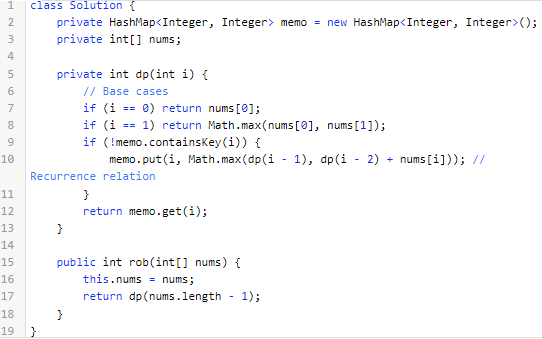
3. **Base cases**

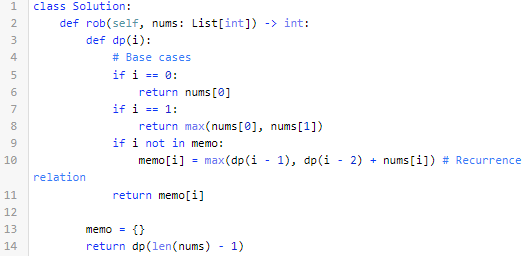
The last thing we need is base cases so that our recurrence relation knows when to stop. The base cases are often found from clues in the problem description or found using logical thinking. In this problem, if there is only one house, then the most money we can make is by robbing the house (the alternative 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

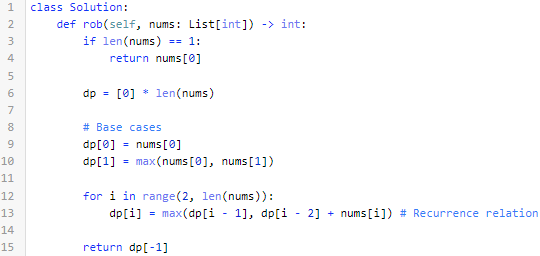
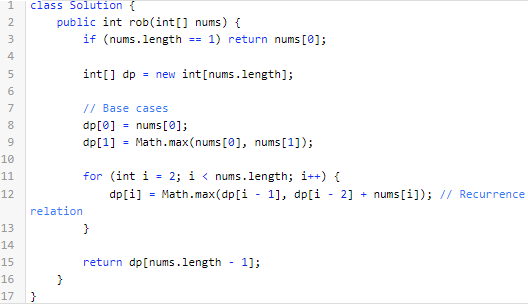
Now that we have established all 3 parts of the framework, let's put it together for the final result. Remember: we need to memoize the function!





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.



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:

**746. Min Cost Climbing Stairs**

Let dp(i) be the minimum cost necessary to reach step i.

We can arrive at step i from either step i - 1 or step i - 2. Choose whichever one is cheaper.

Since we can start from either step 0 or step 1, the cost to reach these steps is 0.

**1137. N-th Tribonacci Number**

Let dp(i) represent the *ith* tribonacci number.

Use the equation given in the problem description.

Use the base cases given in the problem description.

**740. Delete and Earn**

**- Sort** nums and **count** how many times each number occurs in nums.

- Let dp(i) be the maximum number of points you can earn between i and the end of the sorted nums array.

- When we are at index i we have 2 options:

1. Take all numbers that match nums[i] and skip nums[i] + 1.
2. Do not take nums[i] and move to the first occurrence of nums[i] + 1.

Choose whichever option yields the most points.

**Bonus Hint:** When is the first option guaranteed to be better than the second option?

- If we have reached the end of the nums array  (i = nums.length) then return 0 because we cannot gain any more points.