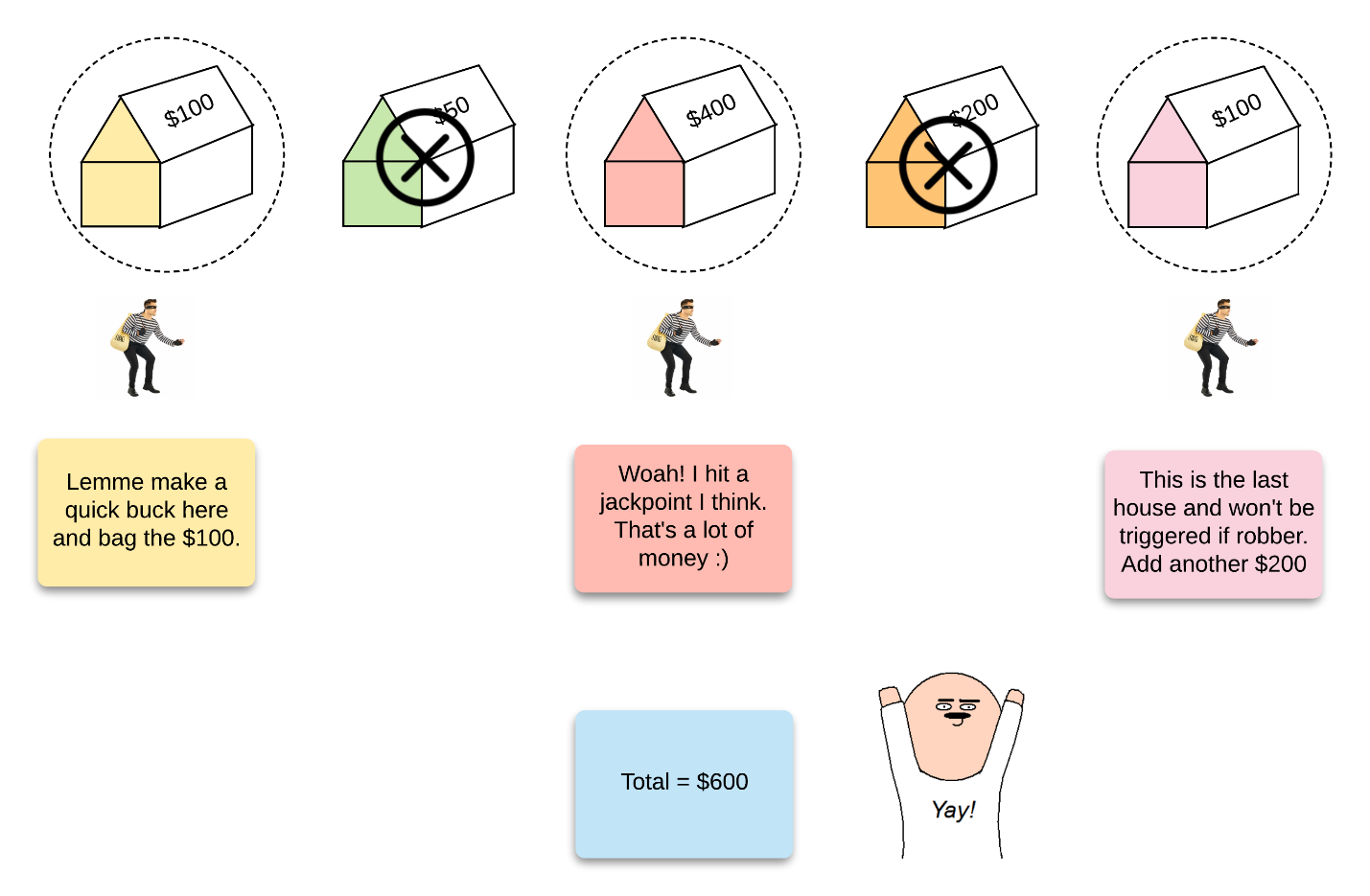
**Overview**

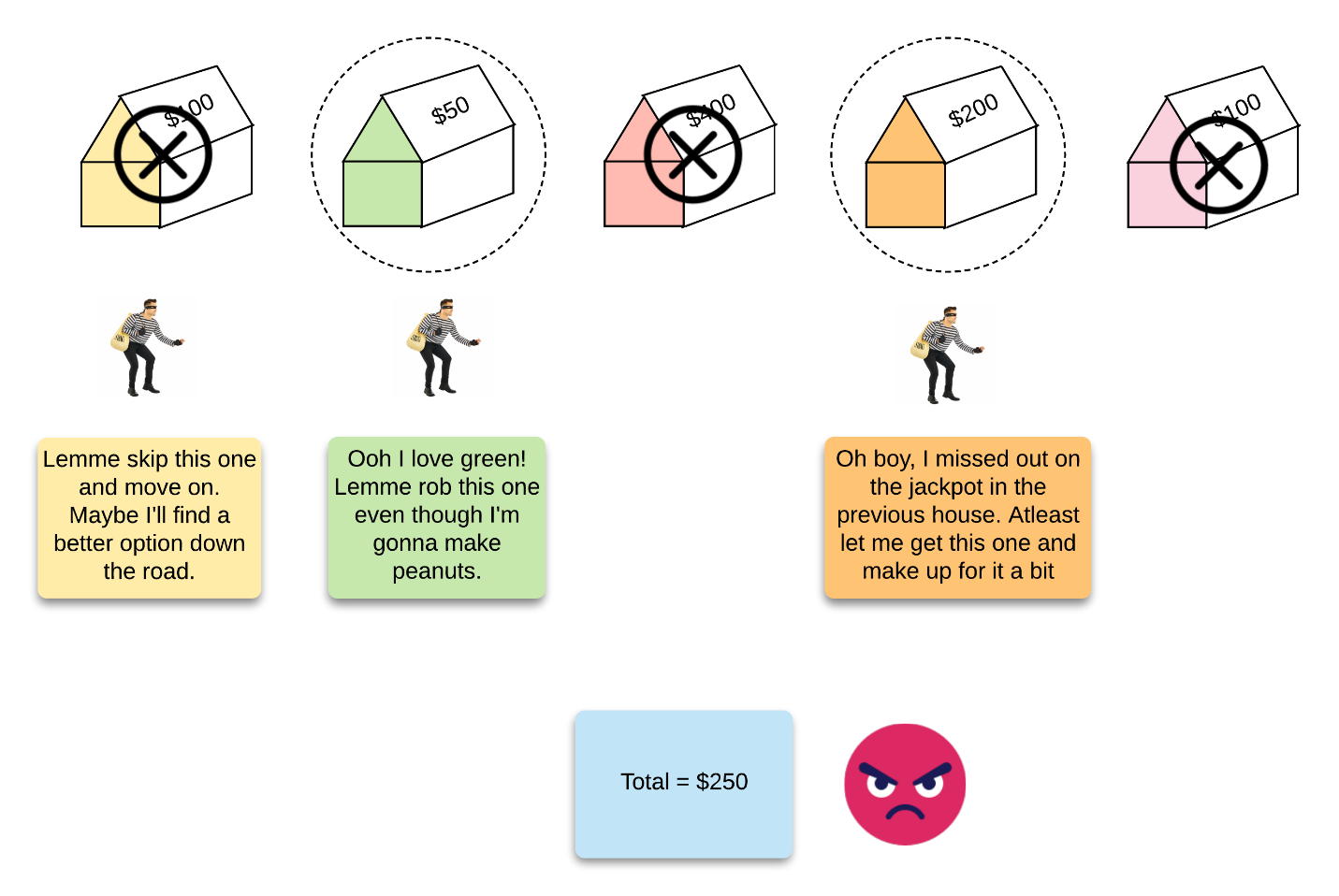
Our professional robber is in for a treat! They have a buffet of houses available for them to rob. Technically, they can rob any of the houses on the street. However, the alarm companies are smart enough to catch any robber red-handed under certain conditions. If there were no alarms in the houses, obviously the robber would rob every house and make the most money.

However, now they have a lot of choices to make. Let's look at a few houses on the street, the different choices our robber can make, and how those choices will affect the heist.



*Figure 1. An example showing the robber making the optimal choices to obtain the most money.*

In this example, the robber ended up making 600 which is the maximum they can make for this series of houses. Let's take another set of choices that the robber can make where they will not make the maximum amount.

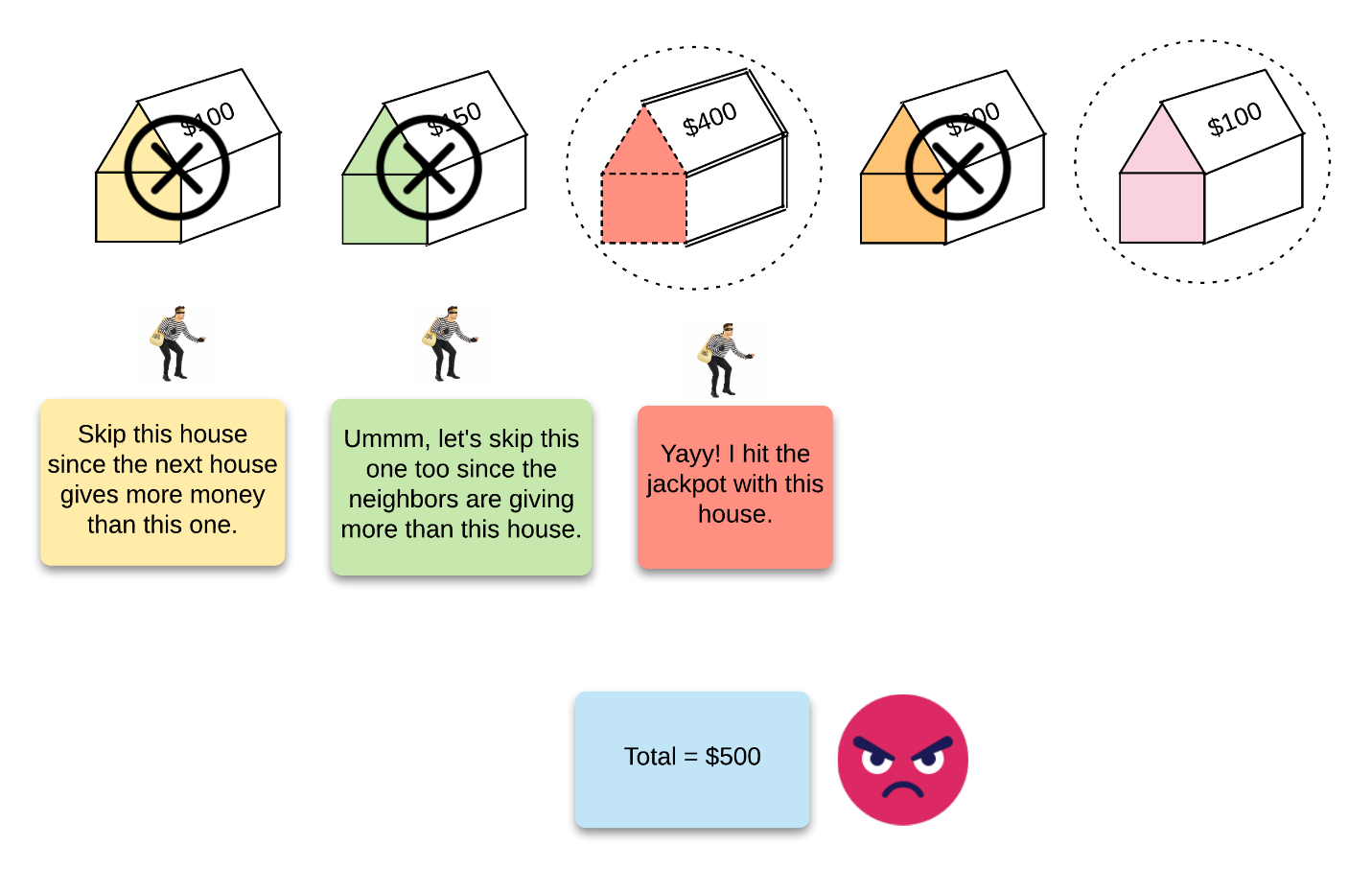


*Figure 2. The same example showing the robber making sub-optimal choices.*

In this case, the series of choices that the robber made turned out to be less than optimal and they ended up making a paltry 250.

A series of choices essentially gives us a subset of houses from the original list. We need to make these choices in such a way that the overall profit is maximized.

There is no *greedy* way of deciding if the robber should rob a house or not. The best greedy strategy may be to check the neighboring houses and only rob a house if it gives them more money than the neighbors combined. That might be a sound greedy strategy. However, by doing so, the robber may miss out on making the maximum profit. Let's look at an example for that.



*Figure 3. Depicting the failure of a greedy strategy on the same example.*

In this example, you could argue that after the robber decided to rob the 3rd house, they could go back and rob the first one as well. In this case, that will give us the optimal answer. However, that decision is still **local** in that we just consider 3 houses at a time.

What we need is to try all the possibilities and see which one gives us (the robber) the optimal loot.

**Approach 1: Recursion with Memoization**

**Intuition**

As we mentioned above, the easiest approach here is to try *all* possible combinations of house choices and then use the combination that gives the maximum amount of money to the robber. We do this because there is no plausible greedy strategy that we can use to decide if the robber should rob a particular house or not.

We rely on our good friend recursion whenever we have *choices* involved in solving a problem. Technically, a robber can come back and rob a house that they previously rejected. However, since we are trying *all* options, we will not go back and rob an unrobbed house since that scenario will be covered in a different recursive path.

The basic choice that we make is whether to rob the current house or not. If the robber decides to rob the current house, they have to skip the next house. Otherwise, they can evaluate the same choice on the next house i.e. to rob or not to rob.

**Subproblems**

To approach a problem recursively, we need to make sure that it can be broken down into sub-problems. Additionally, we need to ensure that the optimal solution to these sub-problems can be used to form the solution to the main problem. Let's see how we can divide this problem into smaller recursive problems.

Let's say that we have a function called robFrom which we will use to solve this problem. The only input this function takes is an index, position. This position essentially represents a *suffix in the array which we, the robber, have yet to scan*. Essentially, the position indicates that the robber has yet to scan houses [position,⋯ ,N][\text{position}, \cdots, N][position,⋯,*N*] where NN*N* represents the total number of houses.

Naturally, the answer to our problem would be the function call robFrom(0) which means scan all the houses and return the maximum profit. Now let's think about robFrom(i) for a moment. This simply represents a sub-array or a suffix from the main array. We can think about this as a smaller max-profit problem in itself, can't we?

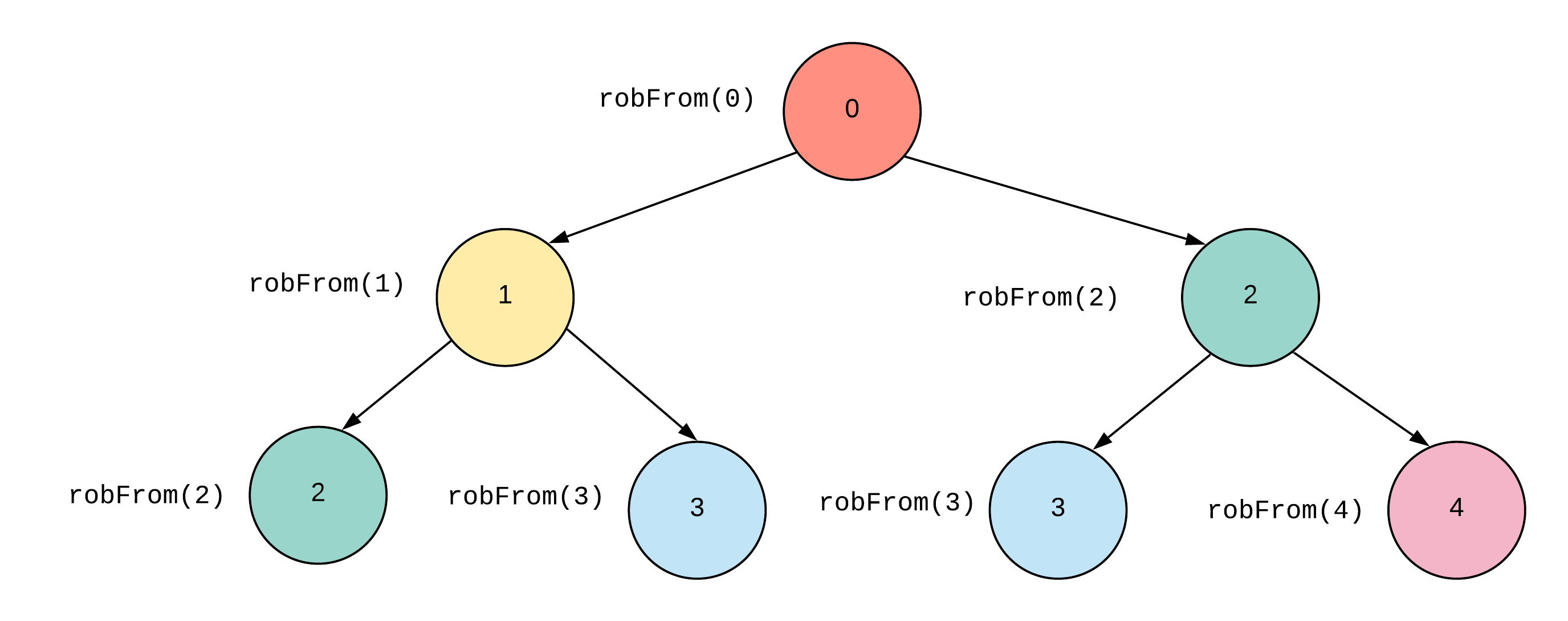
A suffix of the initial set of houses simply means a smaller set of houses that the robber has to consider. We will be working with the assumption that in the function call **robFrom(i)**, the robber has to maximize their profit from i..N houses.

At each step, the robber has two options. If he chooses to rob the current house, he will have to skip the next house on the list by moving two steps forward. If he chooses not to rob the current house, he can simply move on to the next house in the list. Let's see this mathematically.

robFrom(i)=max(robFrom(i+1),robFrom(i+2)+nums(i))\text{robFrom}(i) = \text{max}(\text{robFrom}(i + 1), \text{robFrom}(i + 2) + \text{nums}(i))robFrom(*i*)=max(robFrom(*i*+1),robFrom(*i*+2)+nums(*i*))

**Recursion tree and memoization**

Now that we have an idea about the recursive structure of our problem, let's look at the recursion tree which will be formed. This is important so that we can determine if we have repeating sub-problems, in which case we can use memoization or caching to reduce the overall solution complexity.



*Figure 4. Overlapping sub-problems in the recursion tree.*

As we can see in the recursion tree above, we have the repeating sub-problems (nodes) marked in the same color. A repeating node in the tree represents an entire subtree calculation that has to be repeated which is computationally expensive. Hence, we cache the already computed results so that we don't need to re-calculate the maximum profit for previously seen sub-problems.

Let's formalize this idea concretely in the algorithm section below.

**Algorithm**

1. We define a function called robFrom() which takes the index of the house that the robber currently has to examine. Also, this function takes the nums array which is required during the calculations.
2. At each step of our recursive call, the robber has two options. He can either rob the current house or not.
   1. If the robber chooses to rob the current house, then he have to skip the next house i.e robFrom(i + 2, nums). The answer here would be whatever is returned by robFrom(i + 2, nums) in addition to the amount that the robber will get by robbing the current house i.e. nums[i].
   2. Otherwise, he can simply move on to the house next door and return whatever profit he will make in the sub-problem i.e. robFrom(i + 1, nums).
3. We need to find, cache, and return the maximum of these two options at each step.
4. robFrom(0, nums) will give us the answer to the entire problem.