Solution

Approach 1: Dynamic Programming

**Template**

This is a classical dynamic programming problem.

Here is a template one could use:

* Define the base cases for which the answer is obvious.
* Develop the strategy to compute more complex case from more simple one.
* Link the answer to base cases with this strategy.

**Example**

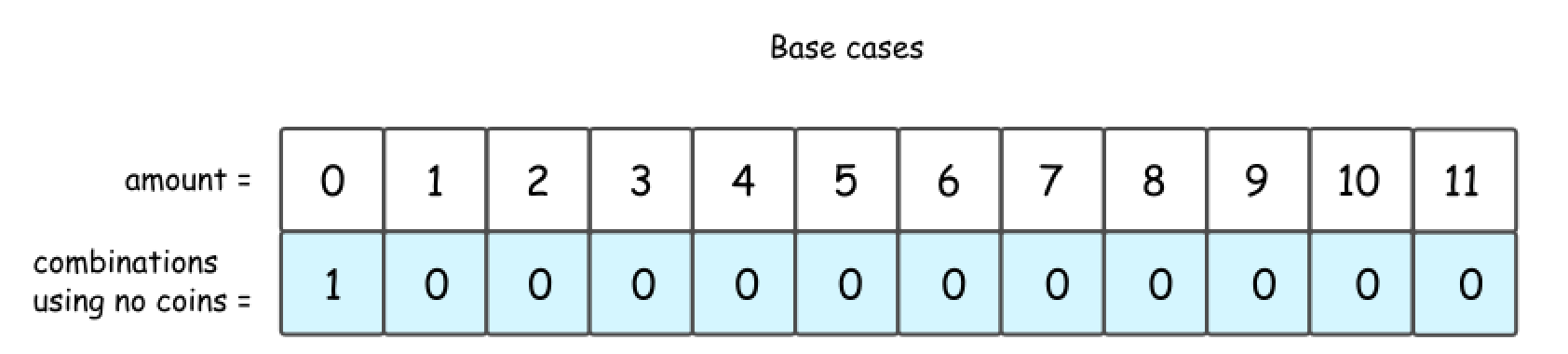
Let's pic up an example: amount = 11, available coins - 2 cent, 5 cent and 10 cent. Note, that coins are unlimited.



**Base Cases: No Coins or Amount = 0**

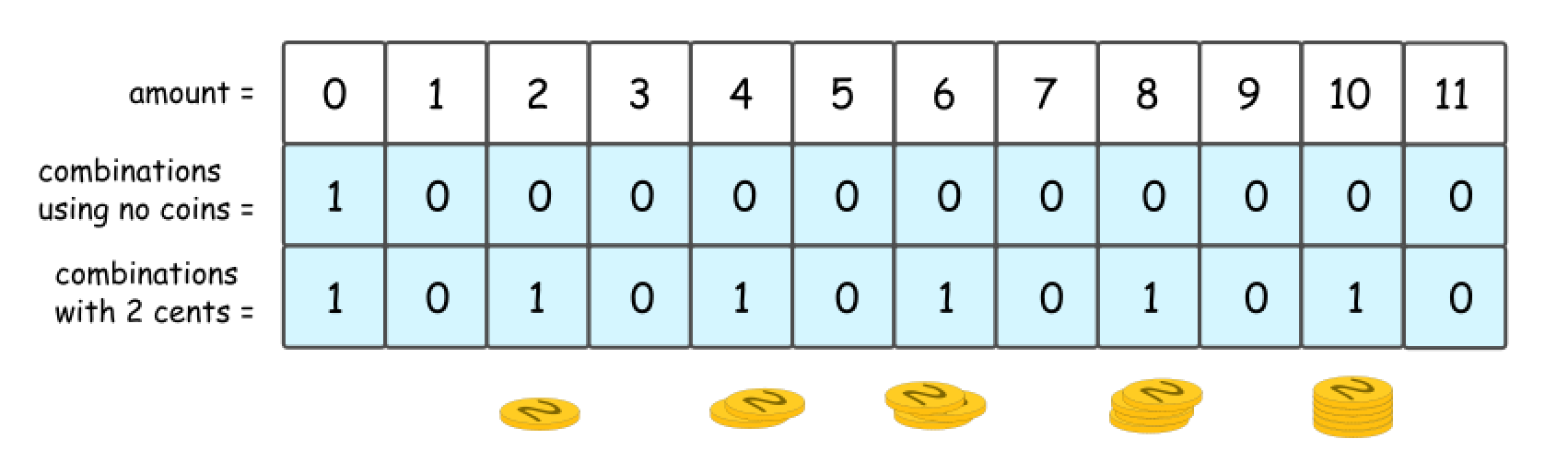
If the total amount of money is zero, there is only one combination: to take zero coins.

Another base case is no coins: zero combinations for amount > 0 and one combination for amount == 0.



**2 Cent Coins**

Let's do one step further and consider the situation with one kind of available coins: 2 cent.



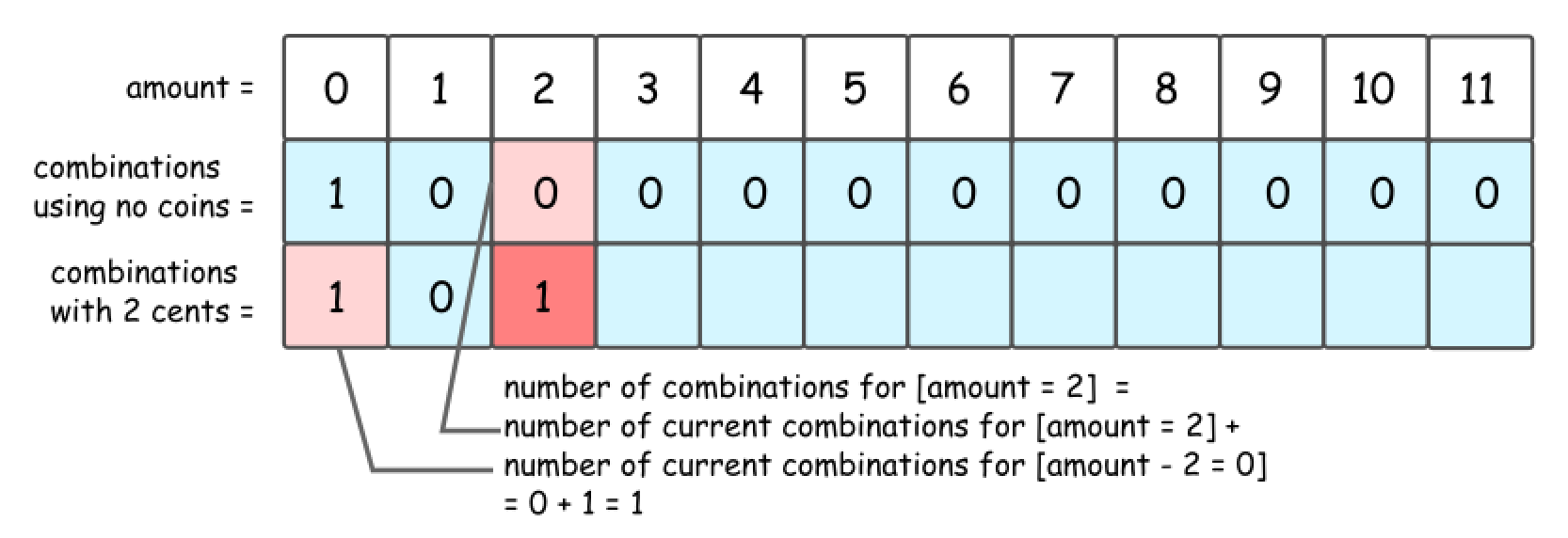
It's quite evident that there could be 1 or 0 combinations here, 1 combination for even amount and 0 combinations for the odd one.

The same answer could be received in a recursive way, by computing the number of combinations for all amounts of money, from 0 to 11.

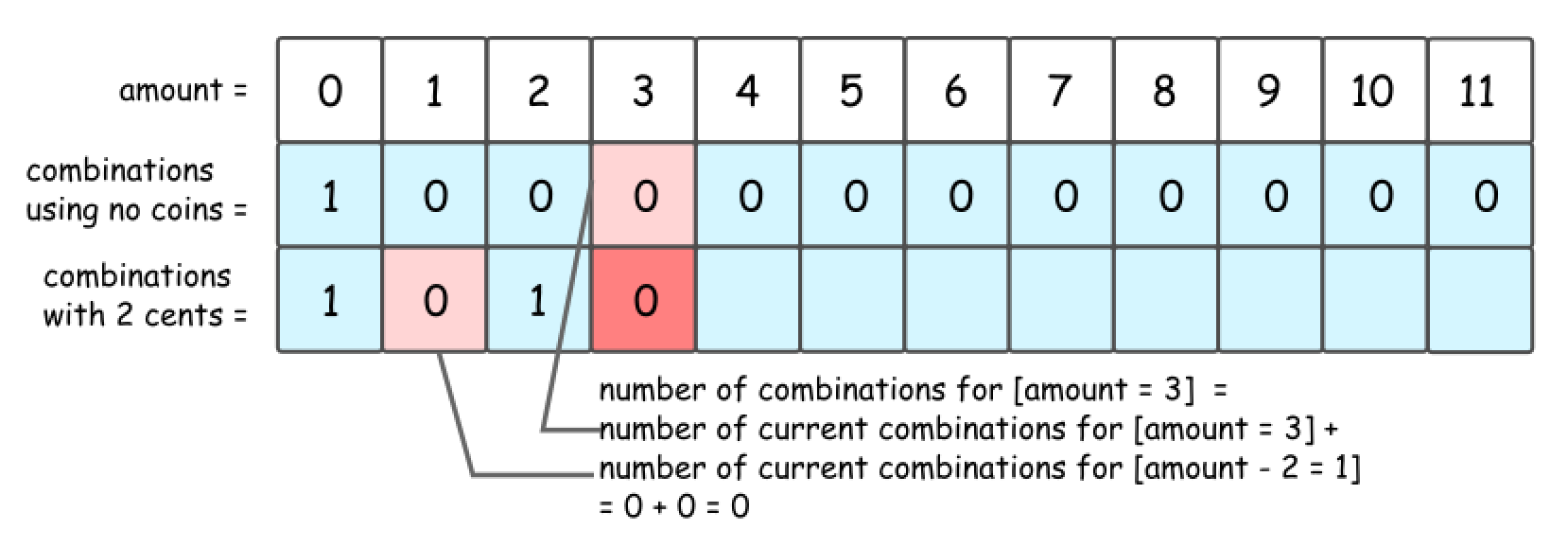
First, that's quite obvious that all amounts less than 2 are *not* impacted by the presence of 2 cent coins. Hence for amount = 0 and for amount = 1 one could reuse the results from the figure 2.

Starting from amount = 2, one could use 2 cent coins in the combinations. Since the amounts are considered gradually from 2 to 11, at each given moment one could be sure to add not more than one coin to the previously known combinations.

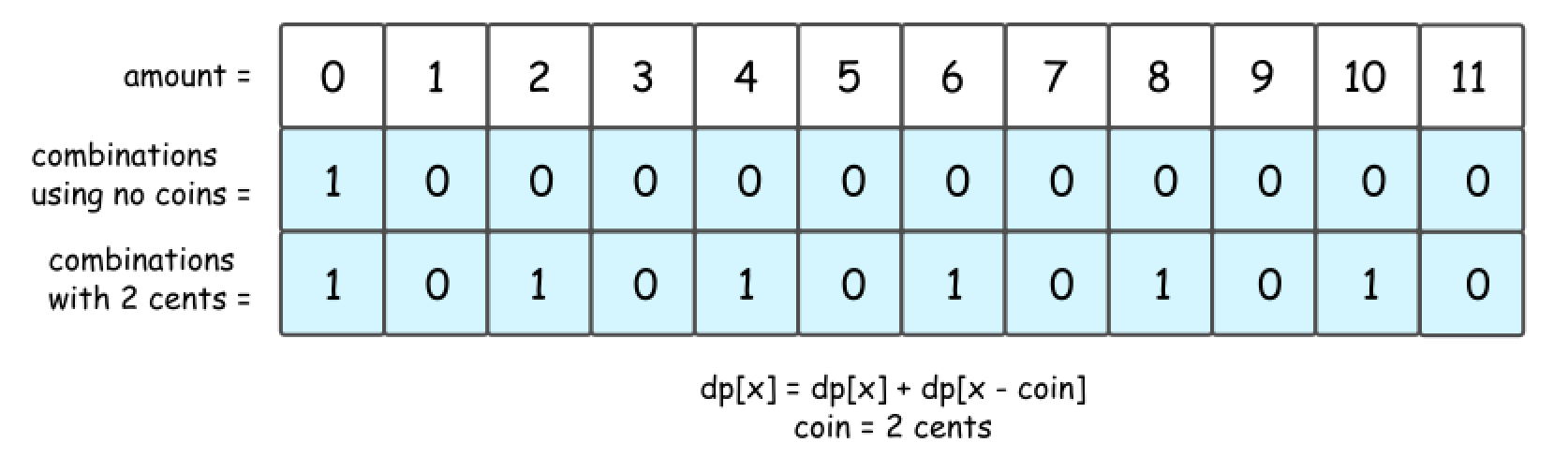
So let's pick up 2 cent coin, and use it to make up amount = 2. The number of combinations with this 2 cent coin is a number combinations for amount = 0, i.e. 1.



Now let's pick up 2 cent coin, and use it to make up amount = 3. The number of combinations with this 2 cent coin is a number combinations for amount = 1, i.e. 0.

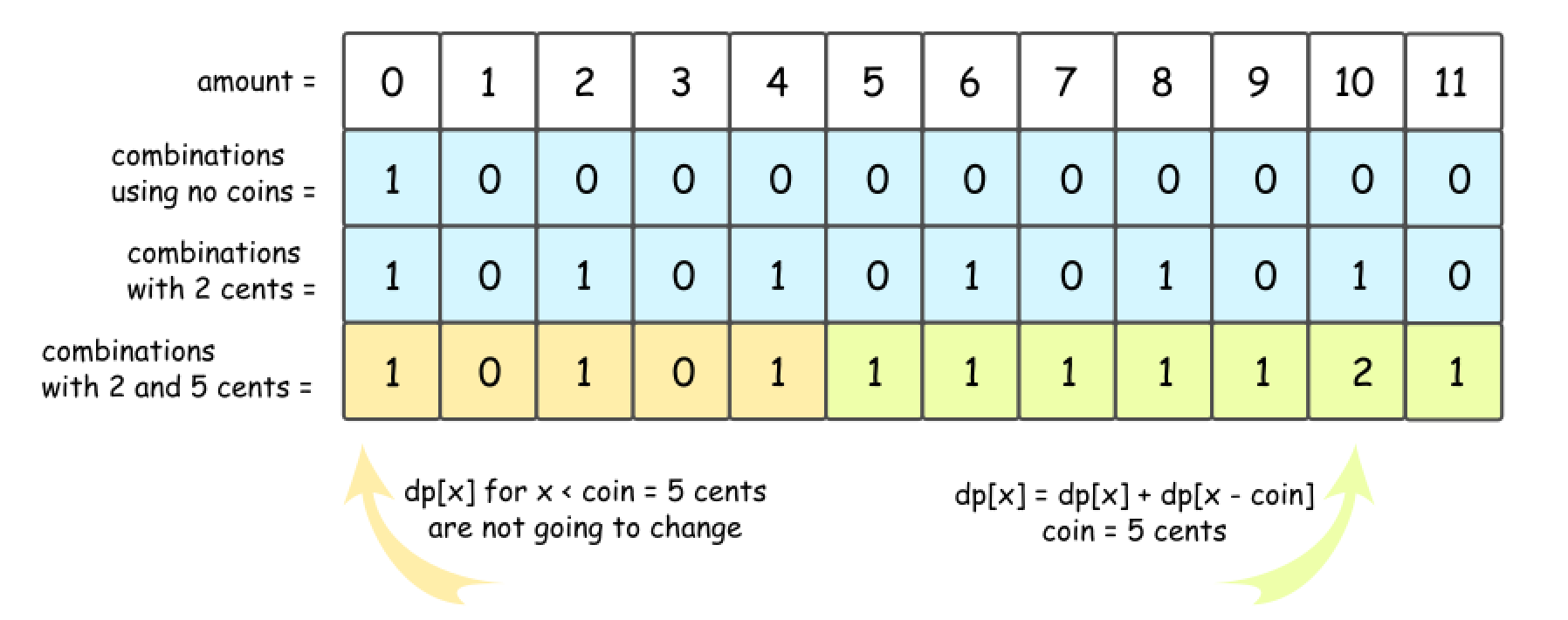


That leads to DP formula for number of combinations to make up the amount = x: dp[x] = dp[x] + dp[x - coin], where coin = 2 cents is a value of coins we're currently adding.

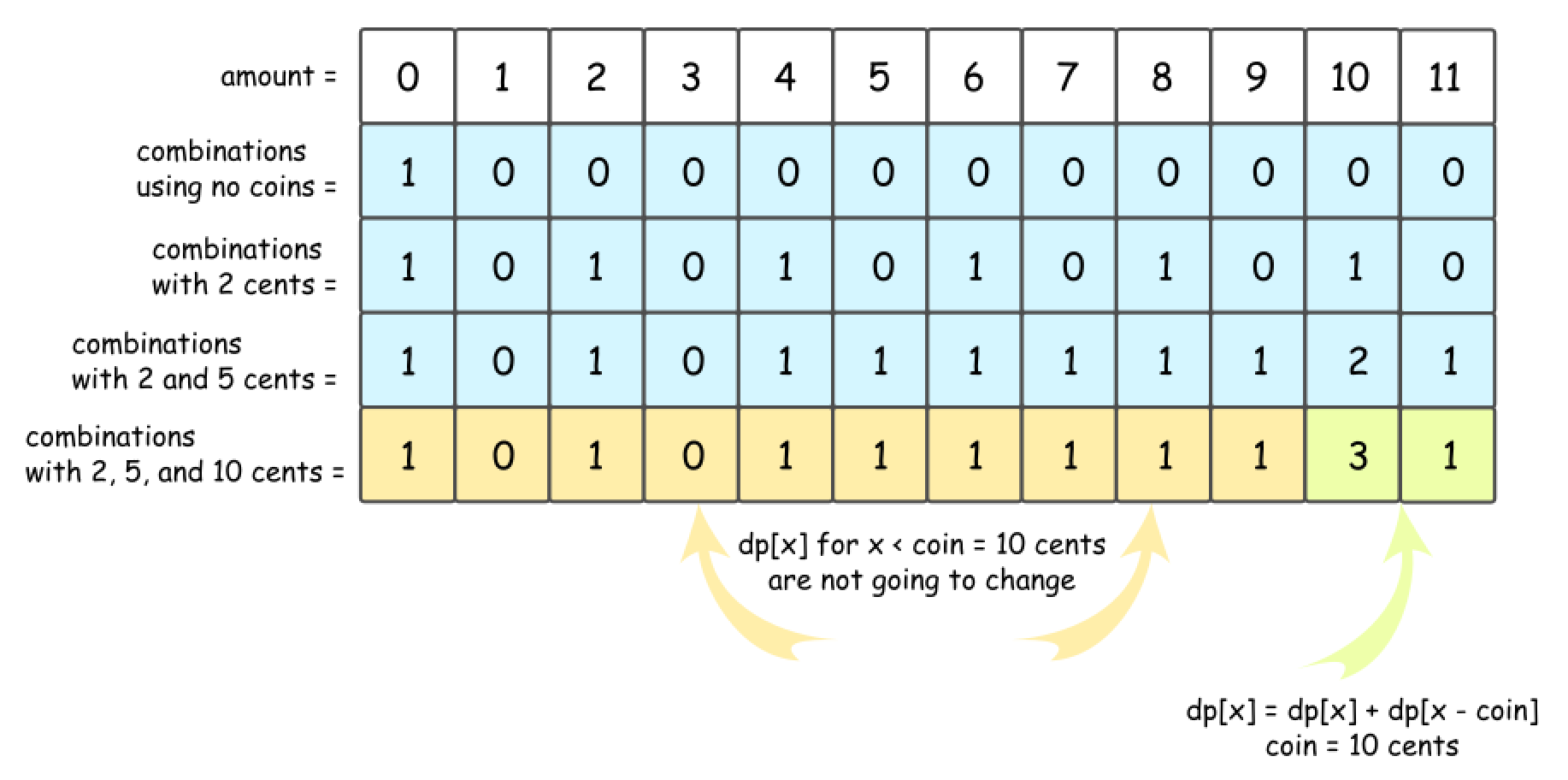


**2 Cent Coins + 5 Cent Coins + 10 Cent Coins**

Now let's add 5 cent coins. The formula is the same, but do not forget to add dp[x], number of combinations with 2 cent coins.



The story is the same for 10 cent coins.



Now the strategy is here:

* Add coins one-by-one, starting from base case "no coins".
* For each added coin, compute recursively the number of combinations for each amount of money from 0 to amount.

**Algorithm**

* Initiate number of combinations array with the base case "no coins": dp[0] = 1, and all the rest = 0.
* Loop over all coins:
  + For each coin, loop over all amounts from 0 to amount:
    - For each amount x, compute the number of combinations: dp[x] += dp[x - coin].
* Return dp[amount].

**Implementation**

**Complexity Analysis**

* Time complexity: \mathcal{O}(N \times \textrm{amount})O(*N*×amount), where N is a length of coins array.
* Space complexity: \mathcal{O}(\textrm{amount})O(amount) to keep dp array.