Solution Review: Number of Ways to Represent N Dollars

In this lesson, we will see some solutions to the problem from the last challenge.

We'll cover the following

Solution 1: Simple recursion

```
def countways_(bills, amount, maximum):
  if amount == 0:     # base case 1
    return 1
  ways = 0
  # iterate over bills
  for bill in bills:
    # to avoid repetition of similar sequences, use bills smaller than maximum
    if bill <= maximum and amount - bill >= 0:
      # notice how bill becomes maximum in recursive call
```

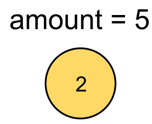
```
      ways += countways_(bills, amount-bill, bill)
  return ways


def countways(bills, amount):
  return countways_(bills, amount, max(bills))

print(countways([1,2,5], 5))
```

Explanation

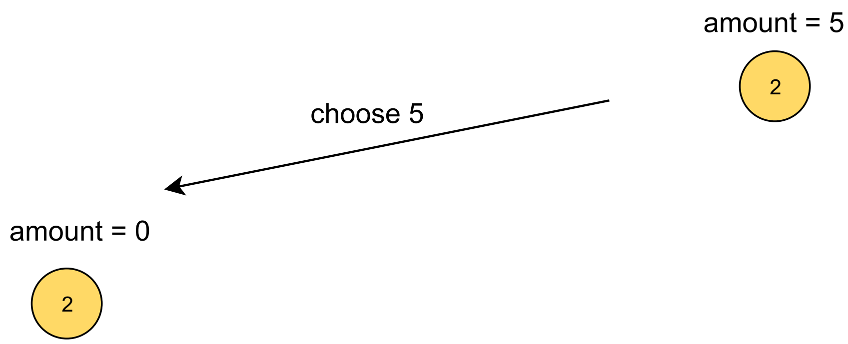
Finding different combinations in a set of things is not difficult at all. In fact, one of the first problems we did in this course was finding different permutations of a string. A slightly challenging bit was how to avoid overcounting due to the same permutations. For example, *$10 + $20* adds to *$30*, so does *$20 + $10*. In the context of permutations, both these sequences would be distinct, but in the case of combinations, these are the same, meaning we must not count them twice. We achieve this by restricting each recursive call to use a subset of bills. The first call can use all *n* bills, the second call can use *n-1* bills excluding the largest one (*lines 8-10*), and so on. This way it is not possible that a recursive call that used a smaller bill would select a larger bill in later recursive calls; thereby avoiding the scenario of cases like *$10 + $20*. Notice that as a base case, we return *1* when `amount` is zero because there is only one way to represent zero irrespective of the number and kinds of bills available. Look at the following visualization of the algorithm.

countways([1,2,5], 5)

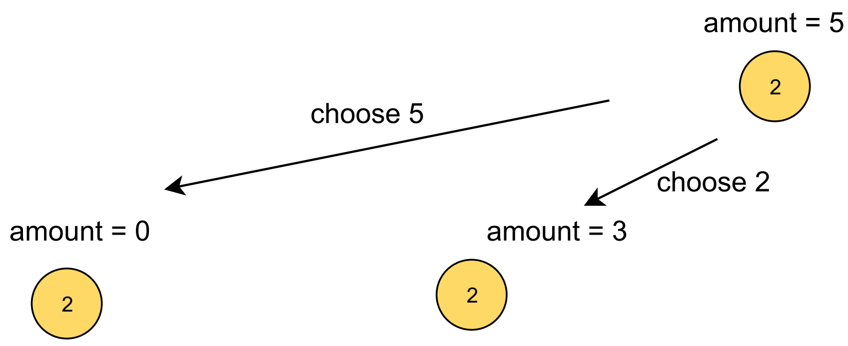
countways([1,2,5], 5)



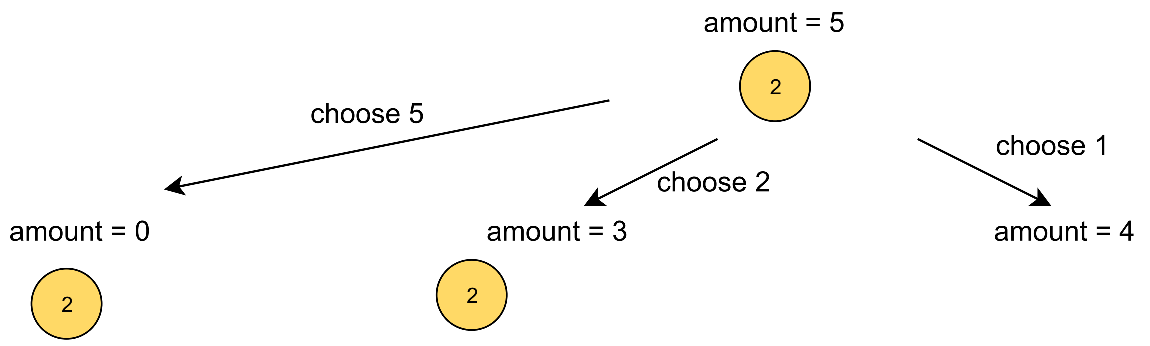
we have to count number of ways to make 5 out of 1,2 and 5



we can choose 5, if we count with 5, we will be left with 0 amount

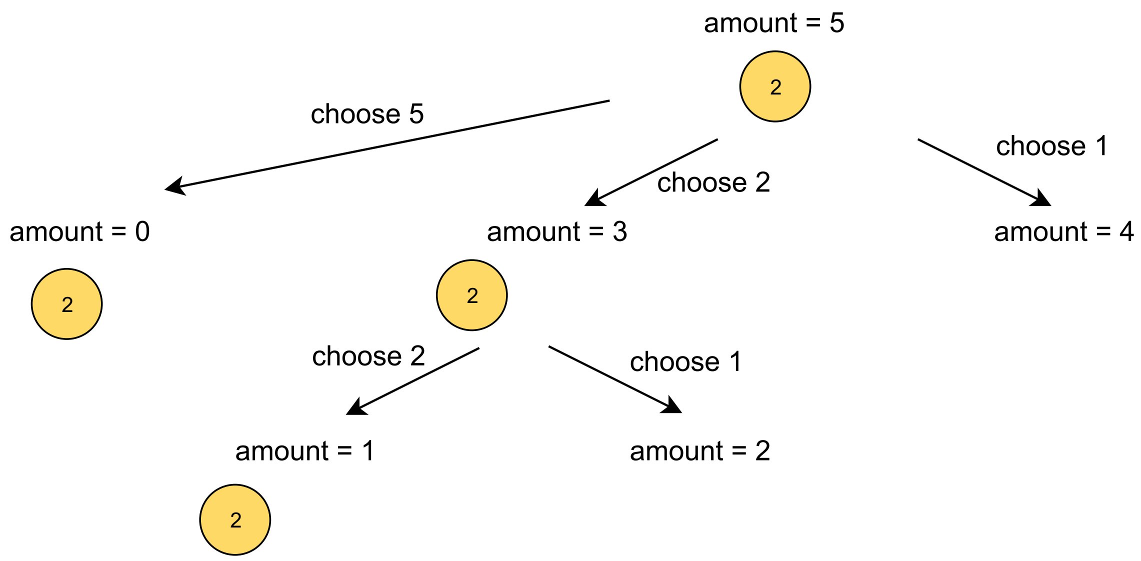


or we can choose 2, thus remaining amount will be 3, also we have reduced our set of possible bills by excluding 5

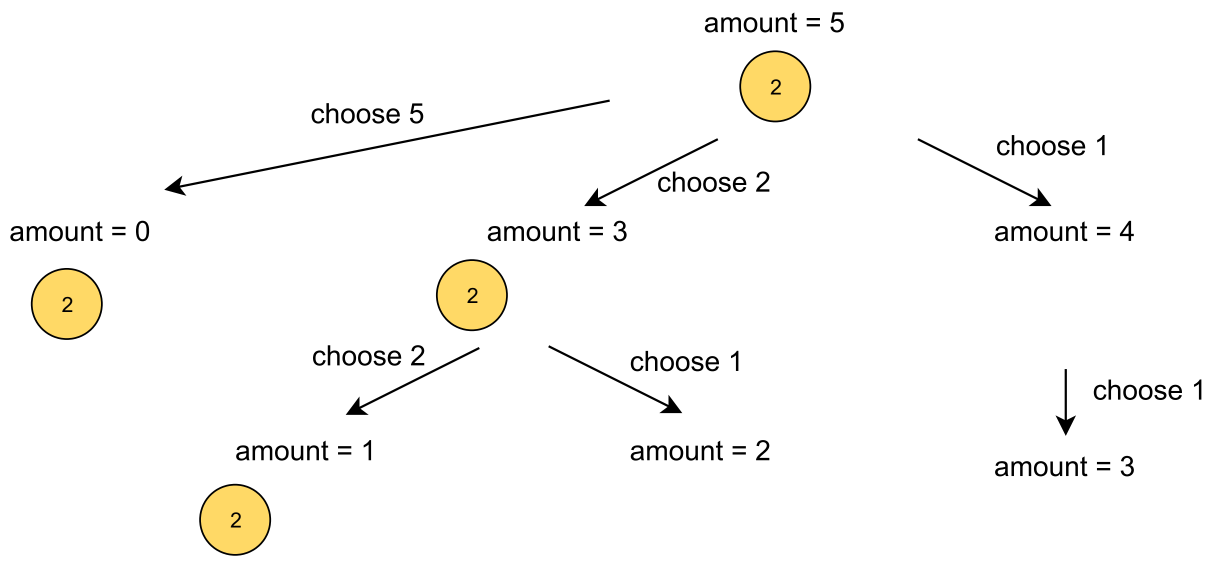


lastly we can choose 1, amount remaining will be 4 and possible set of bills will include 1 only

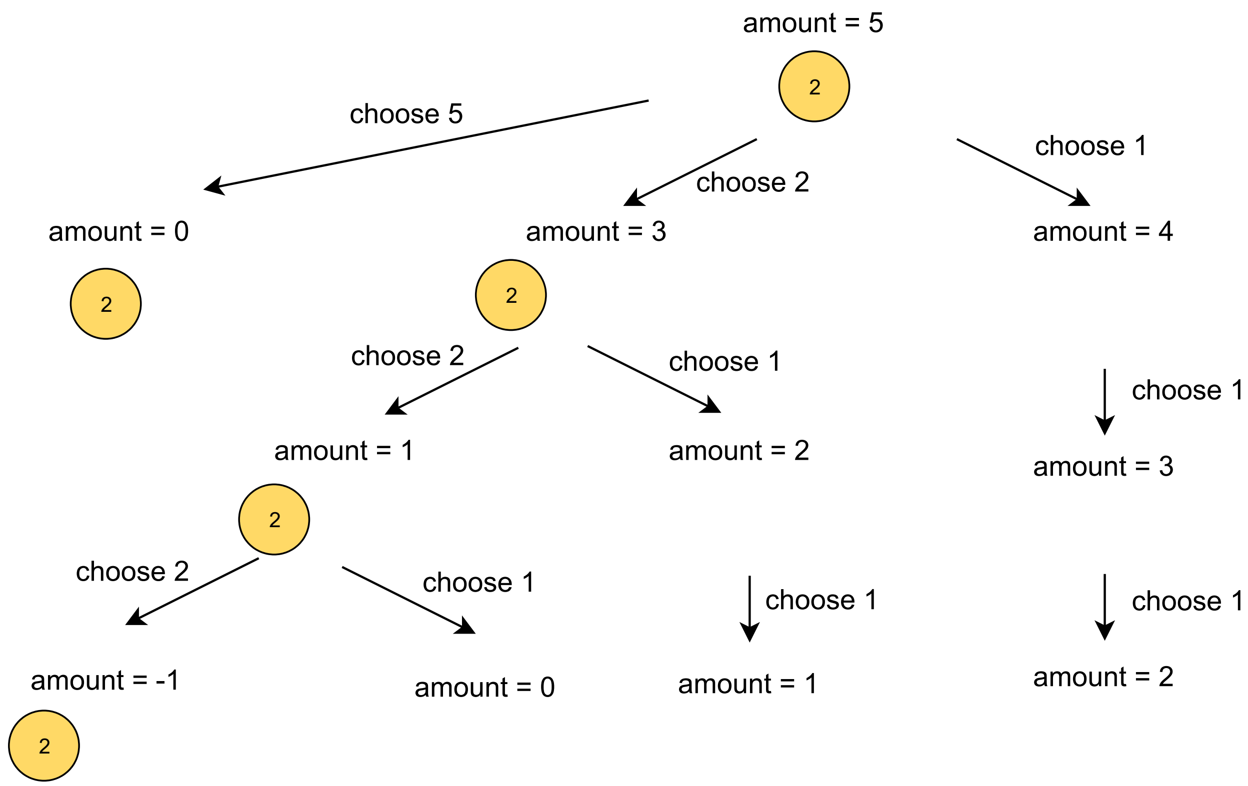
5 of 12



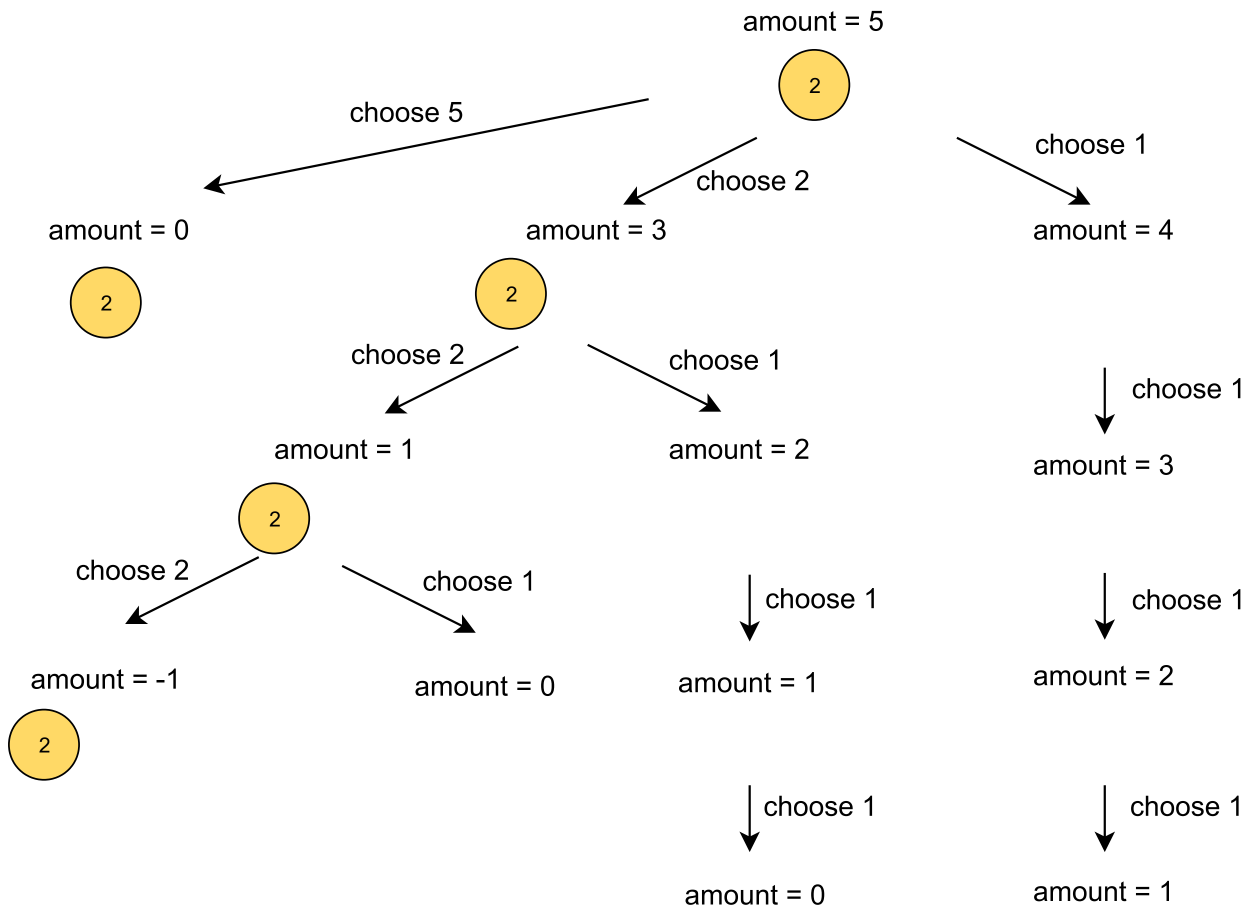
similarly expanding next calls we have



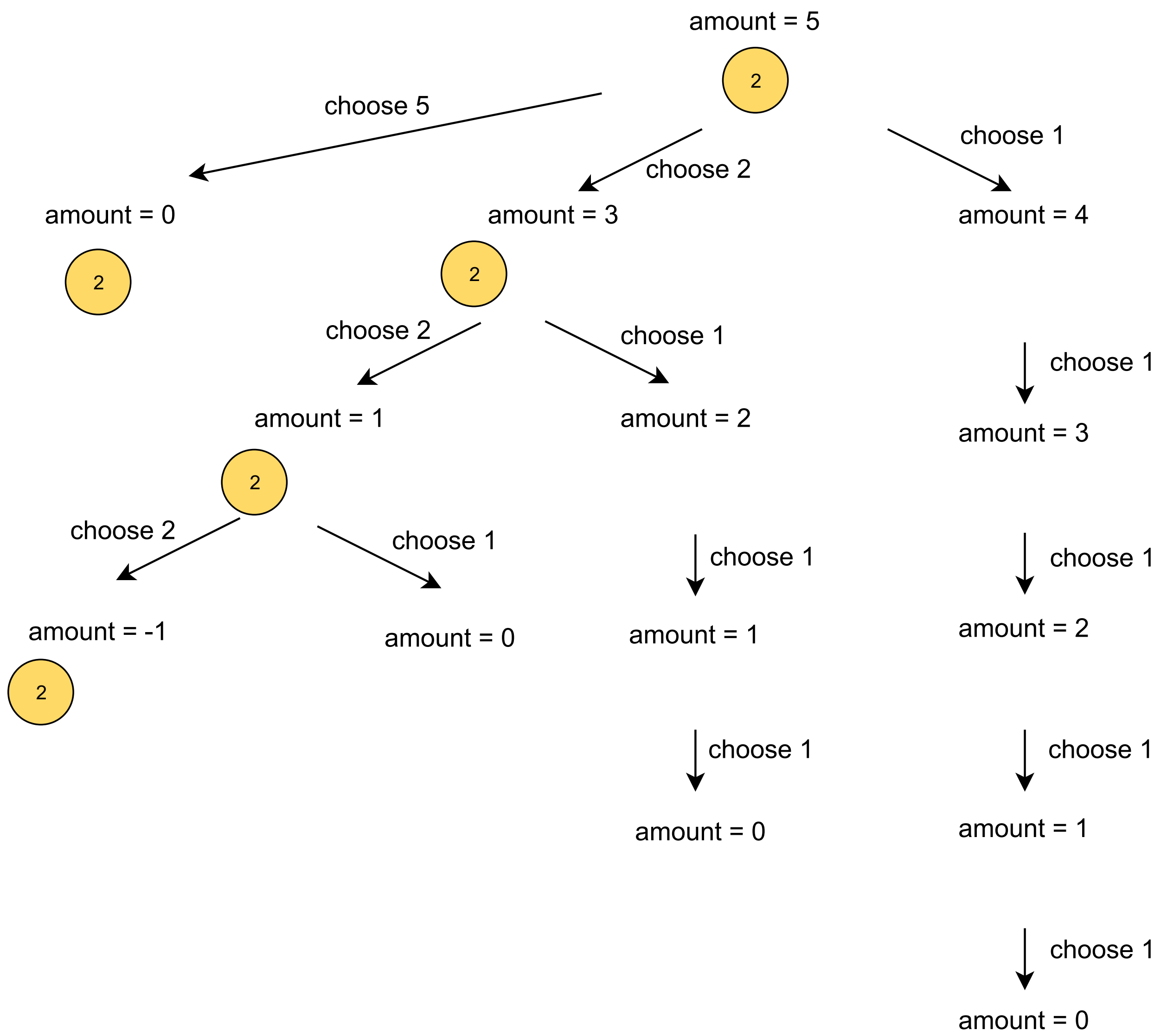
similarly expanding next calls we have



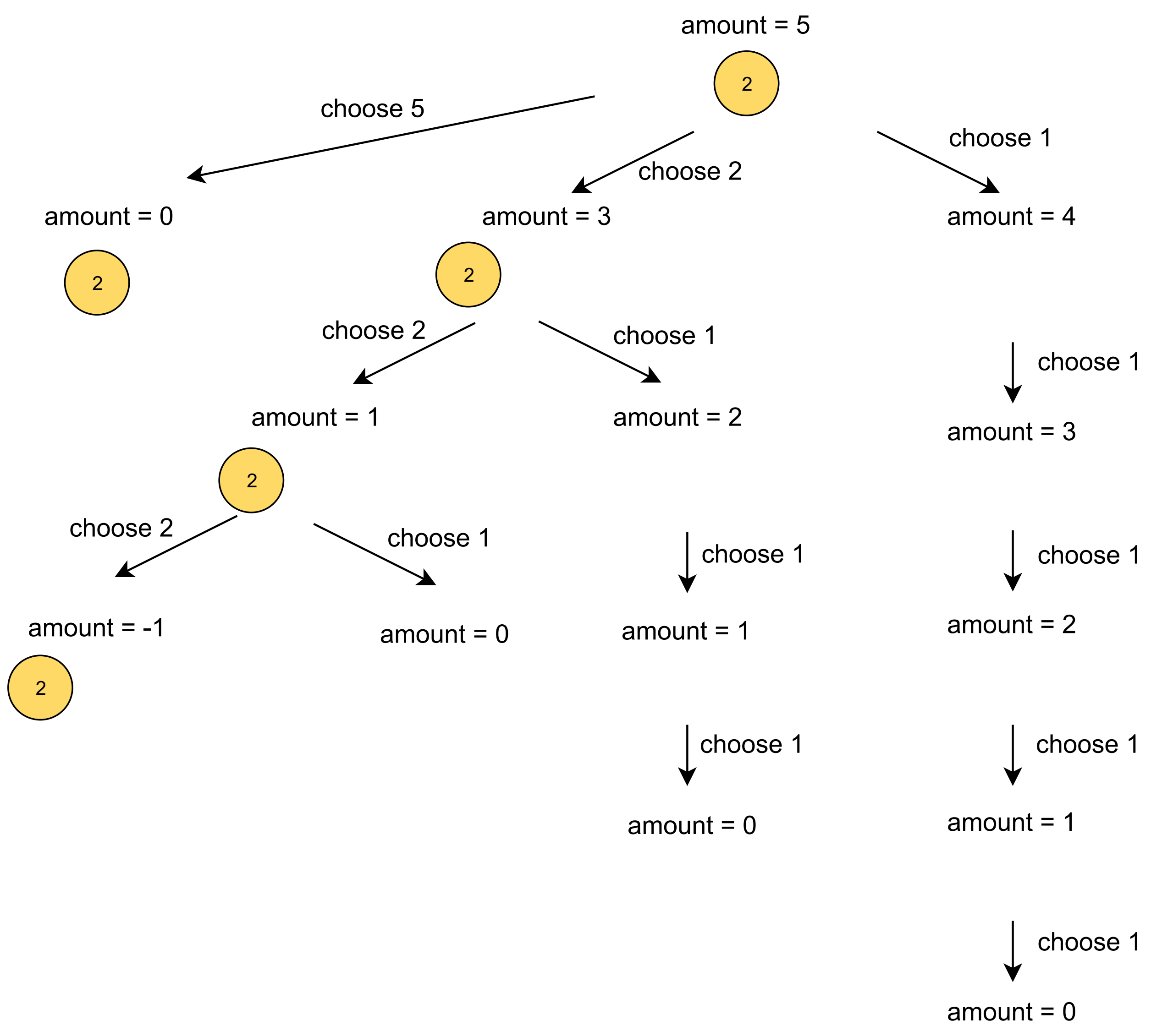
continuing this way we get



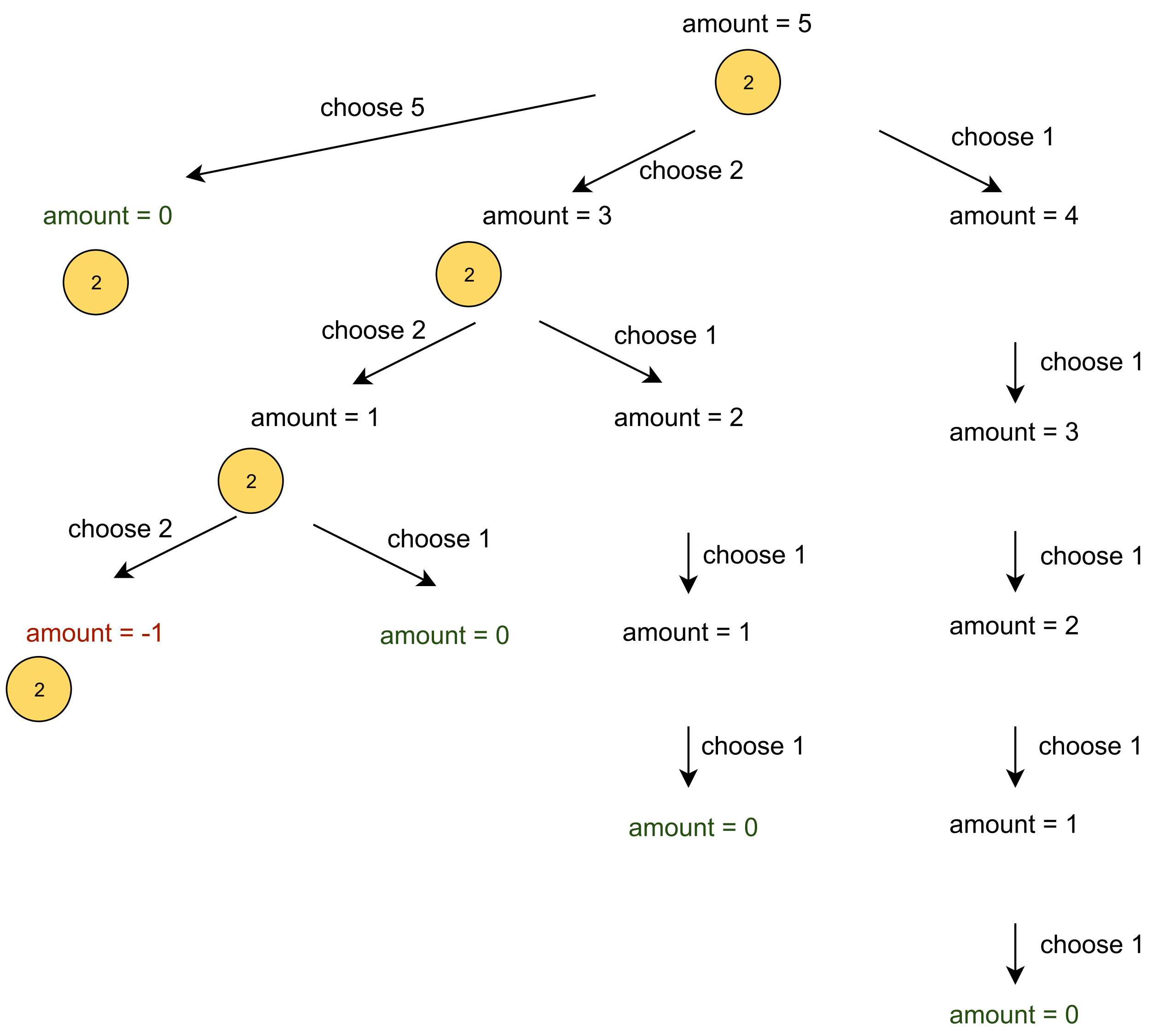
continuing this way we get



Finally we have all the recursive calls evaluated.



count all the leaves (base cases) with amount = 0



There are 4 such leaves, thus our answer evaluates to 4

12 of 12

Another plausible recursive solution is given in the following coding playground. The logic is similar to the first solution, but it is simpler.

Alternate algorithm

```
def countways_(bills, amount, index):
  if amount == 0:     # base case 1
    return 1
  if amount < 0 or index >= len(bills):      # base case 2
    return 0
  # count the number of ways to make amount by including bills[index] and excluding bills[index]
  return countways_(bills, amount - bills[index], index) + countways_(bills, amount, index+1)
```

```
def countways(bills, amount):
  return countways_(bills, amount, 0)


print(countways([1,2,5], 5))
```

Explanation

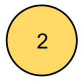
The key part of this solution is the sum of two recursive calls in *line 7*. To count to `amount`, some combinations will either include a specific bill given by `bills[index]` or they won't. So, we can simply count both these possibilities. The first call to `countways_` counts `bills[index]` as a part of the solution, whereas the second call skips over it. Let's see a simple visualization of this algorithm.

countways([1,2,5], 5)

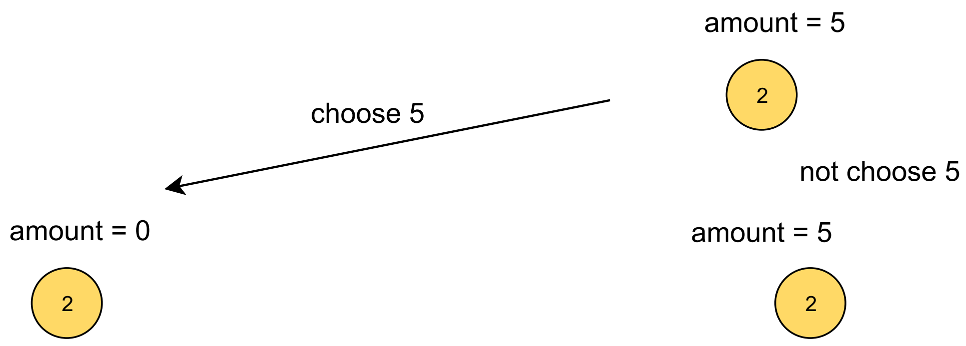
countways([1,2,5], 5)

1 of 9

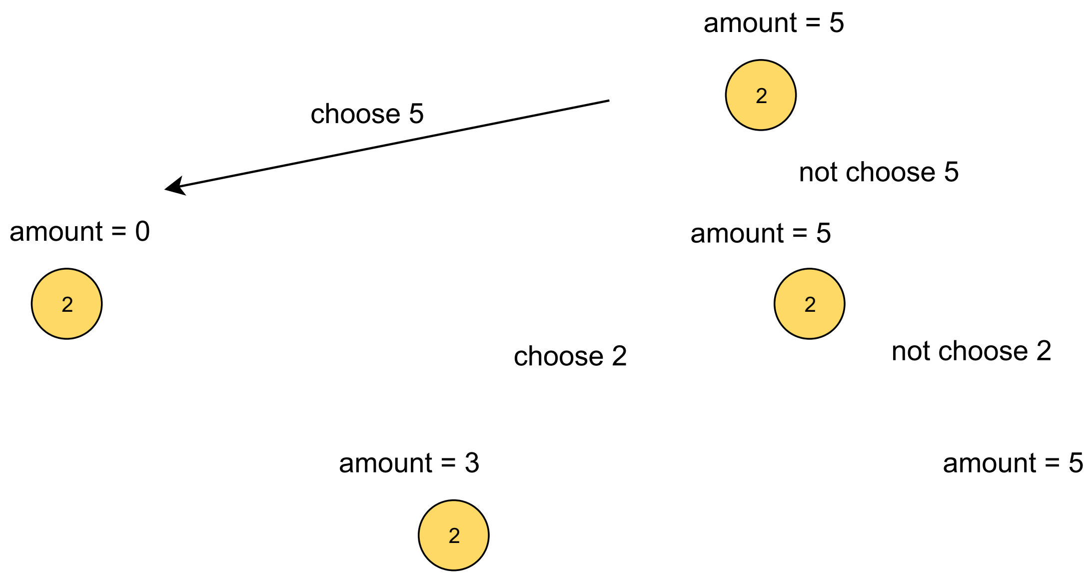
amount = 5



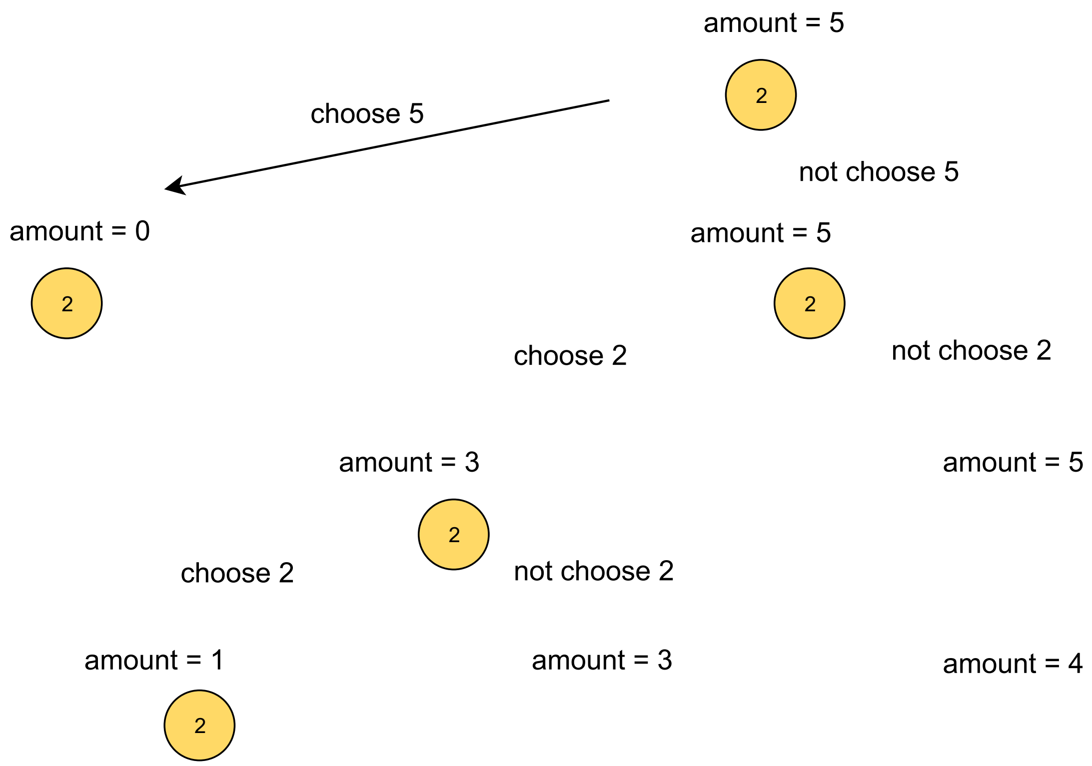
we have to count number of ways to make 5 out of 1,2 and 5



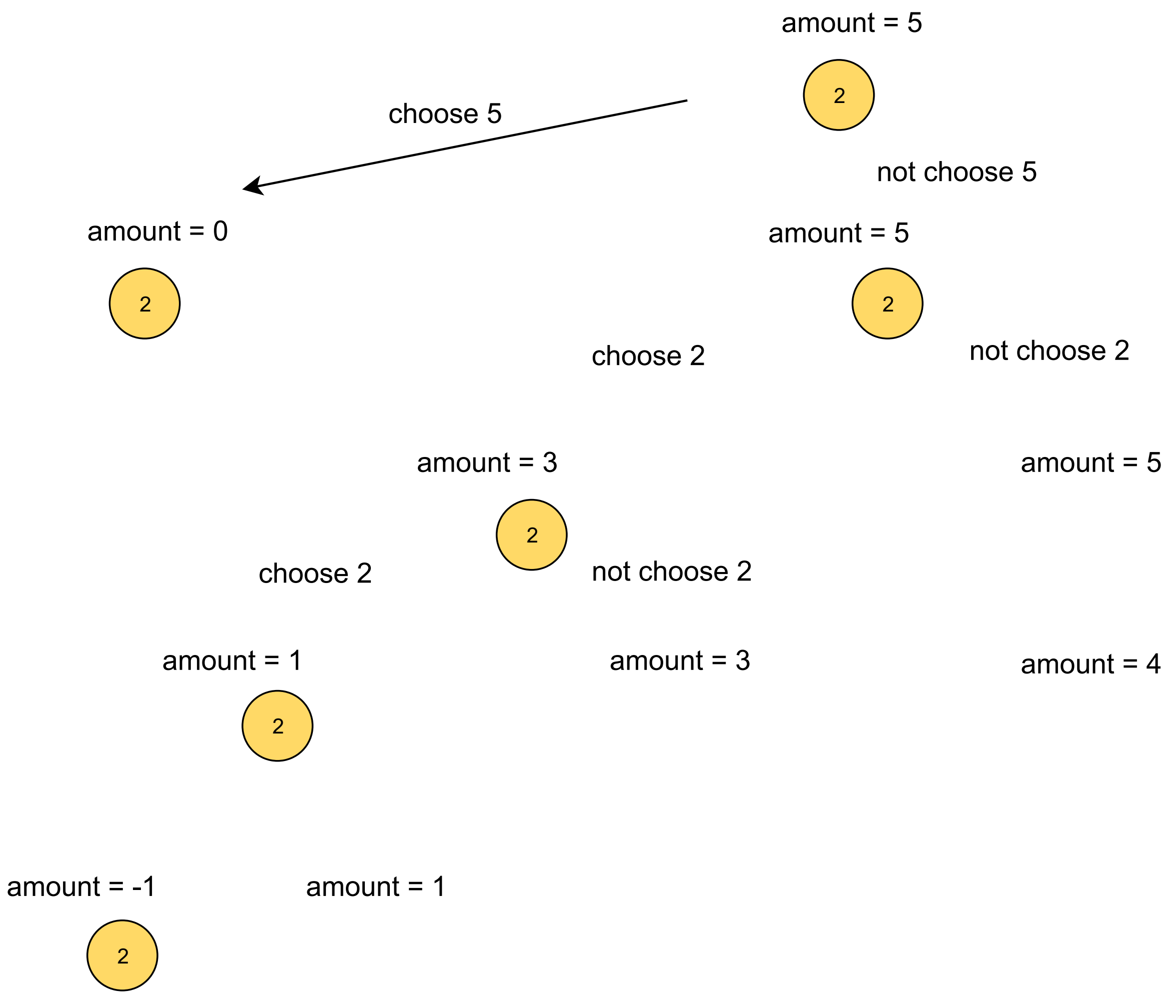
we can either have 5 or we cannot



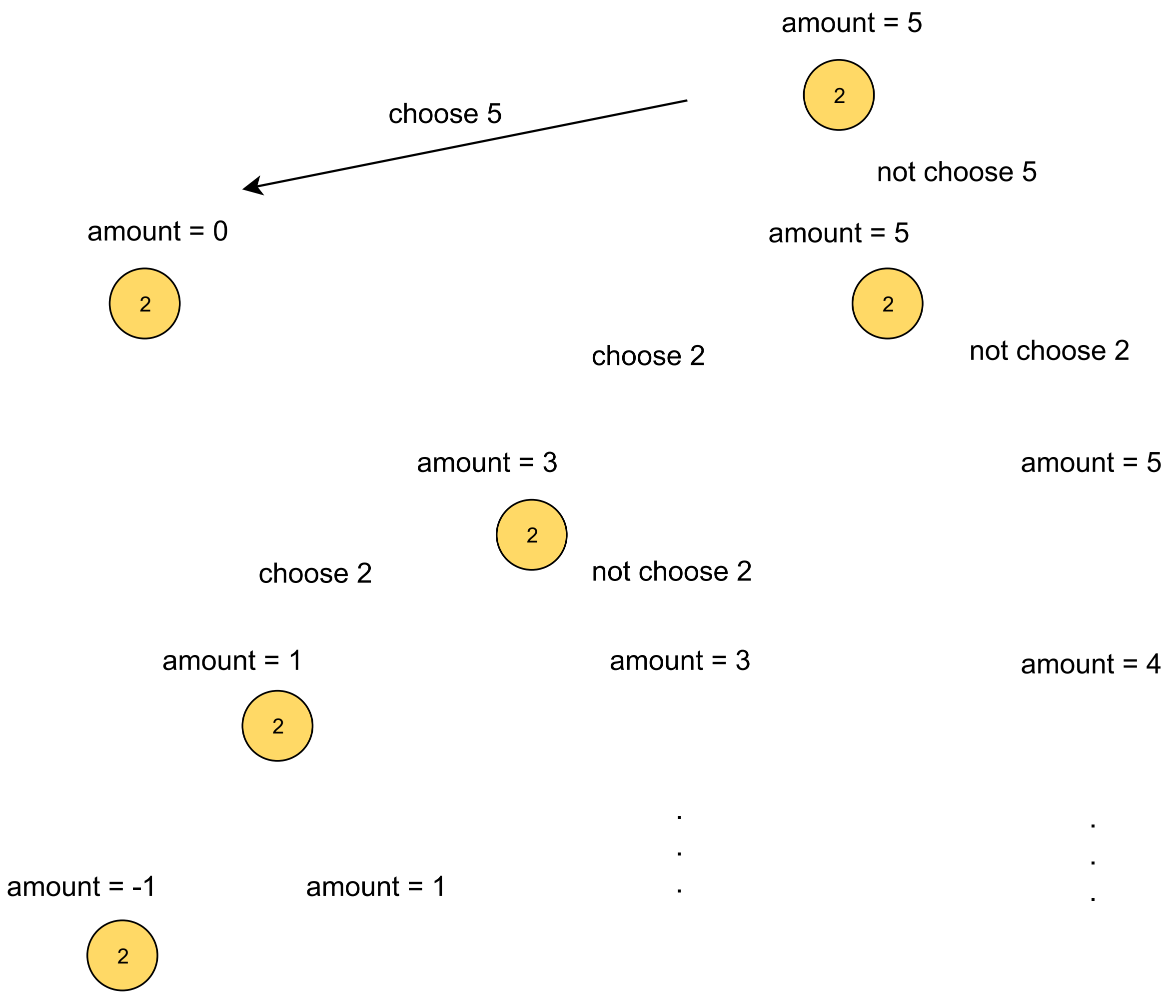
similarly we can either have 2 or we cannot



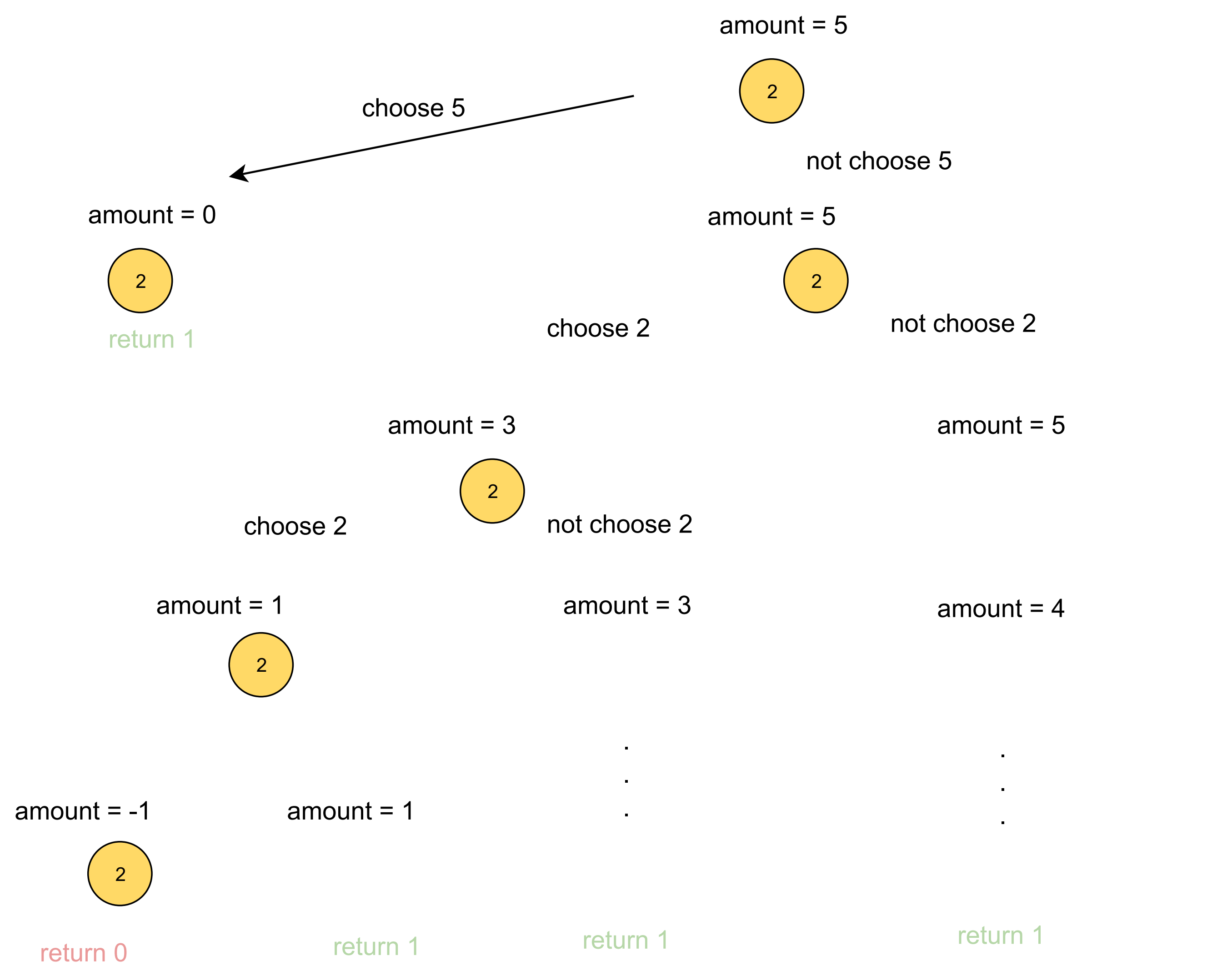
continuing in similar way



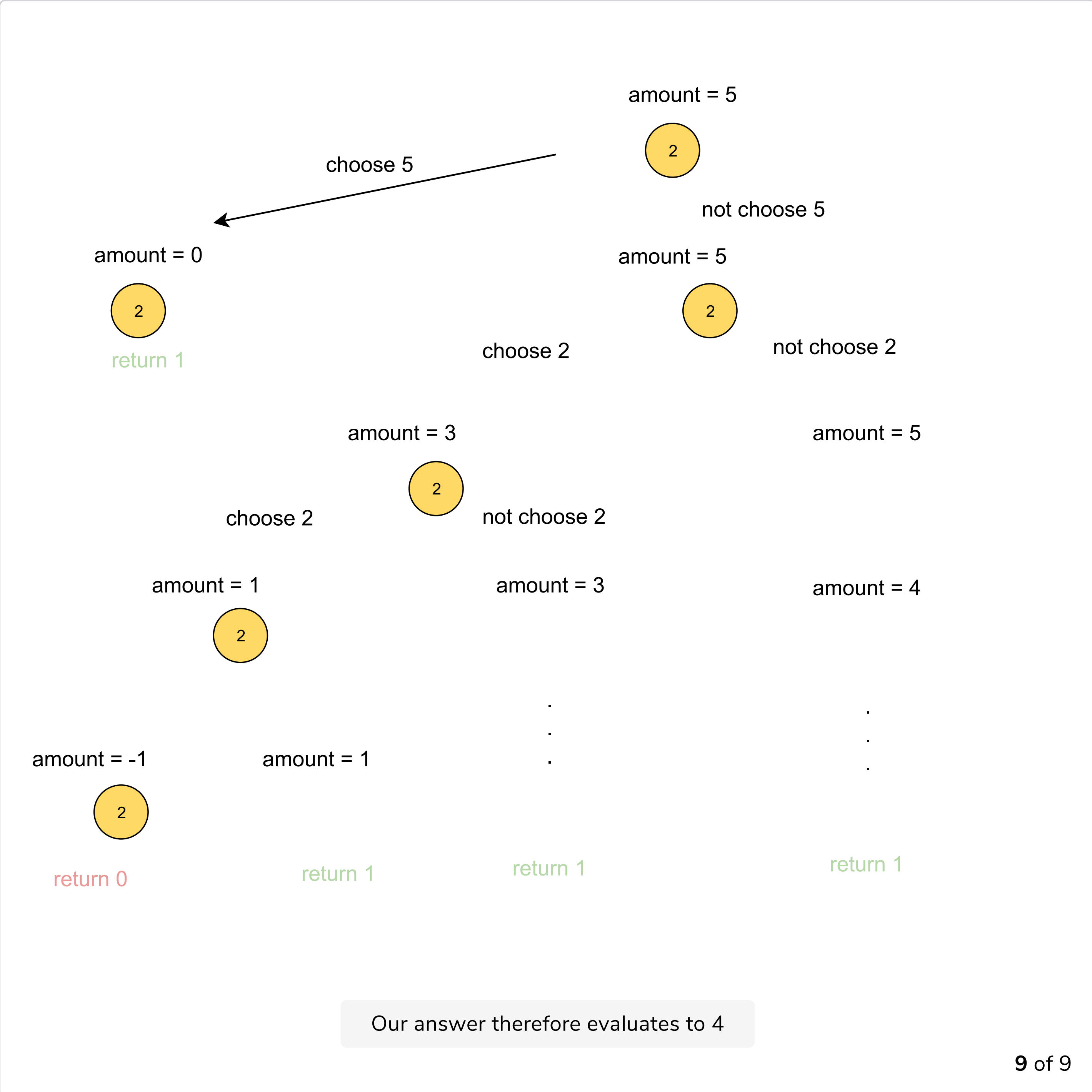
continuing in similar way



the calls with only 1 will evaluate to 1 eventually



thus return from all recursive call where amount would reach 0



Time complexity

If the length of the list `bills` is n, and the amount to be made out of them is `C`, the time complexity of the recursive algorithms would be $\mathbf{O(n^{C})}$. This is because in the first recurrence tree, we can have at most n branches from each node in the tree. The total height of the tree can get as big as C. Thus, the total number of nodes in such a tree is bound by n^{C}; therefore, the time complexity would be $\mathbf{O(n^{C})}$.

Solution 2: Top-down dynamic programming

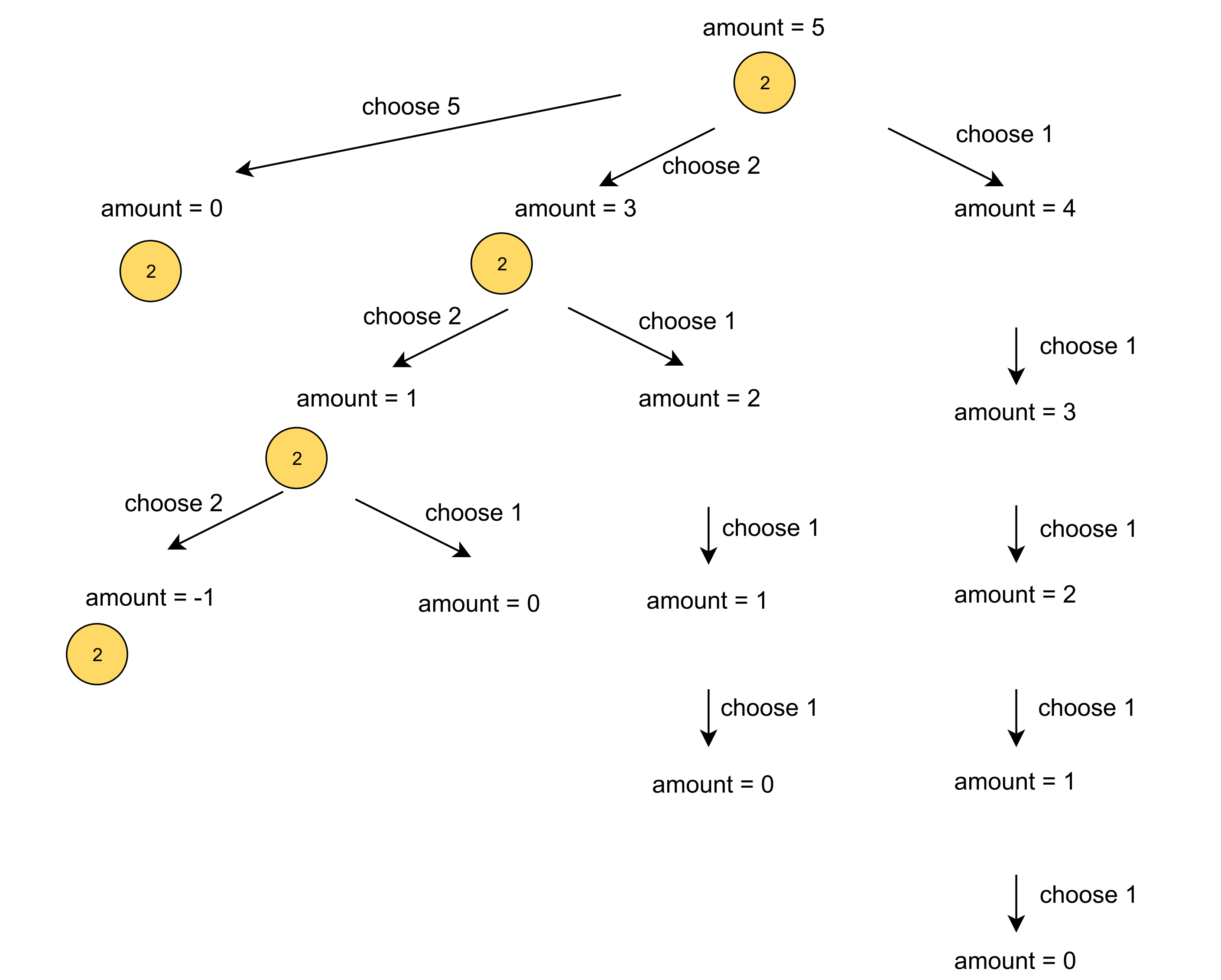
Let's see if this problem satisfies both the properties of dynamic programming before jumping on to the solution.

Optimal substructure

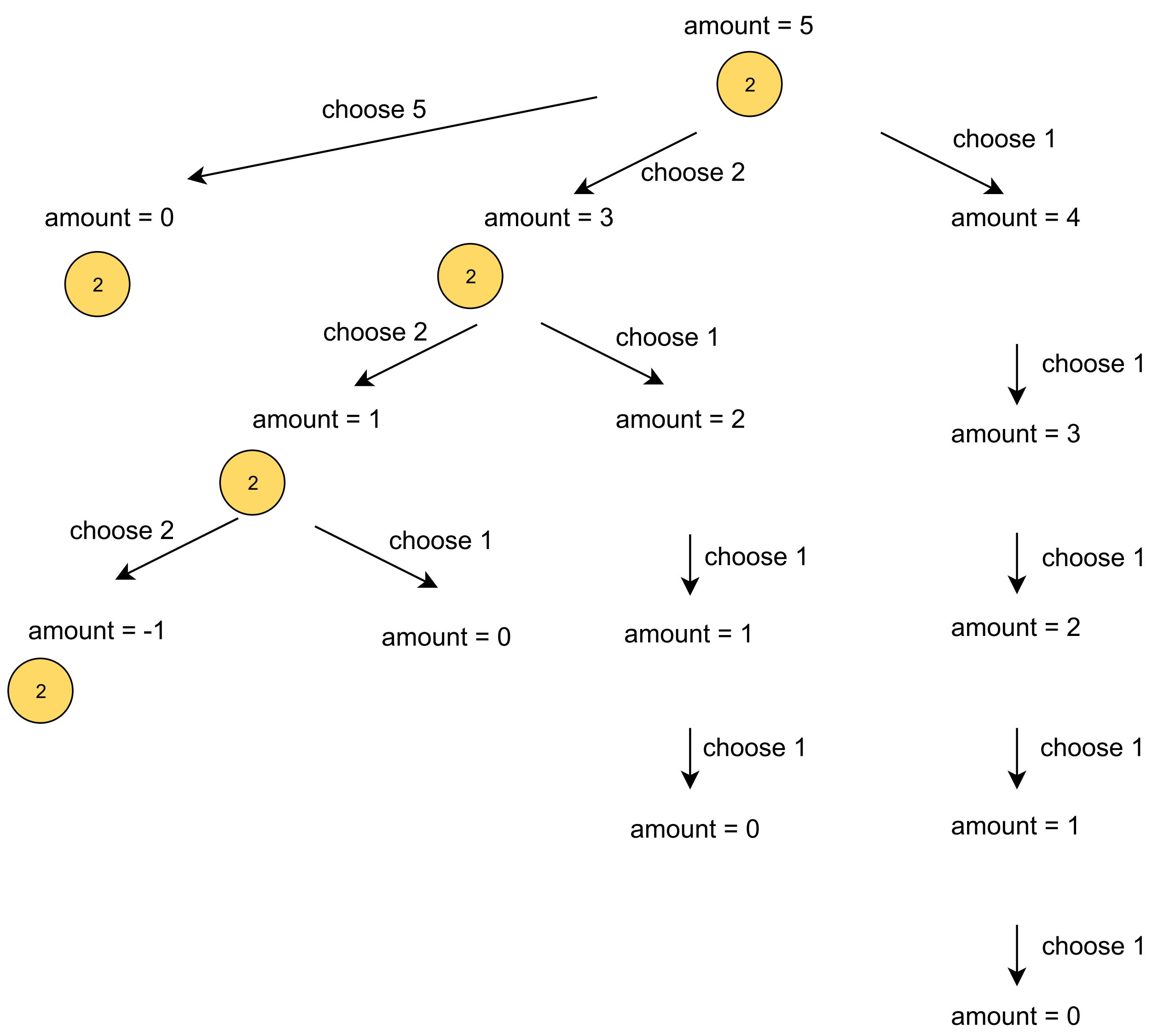
If we wanted to find the solution to the problem of counting an amount, C, given n different bills and we knew the answer to the n subproblems formed by subtracting each of the n bills from the C, we could simply sum up the answer to these subproblems and that would give us answer to our original problem. Thus, this problem obeys optimal substructure property.

Overlapping subproblems

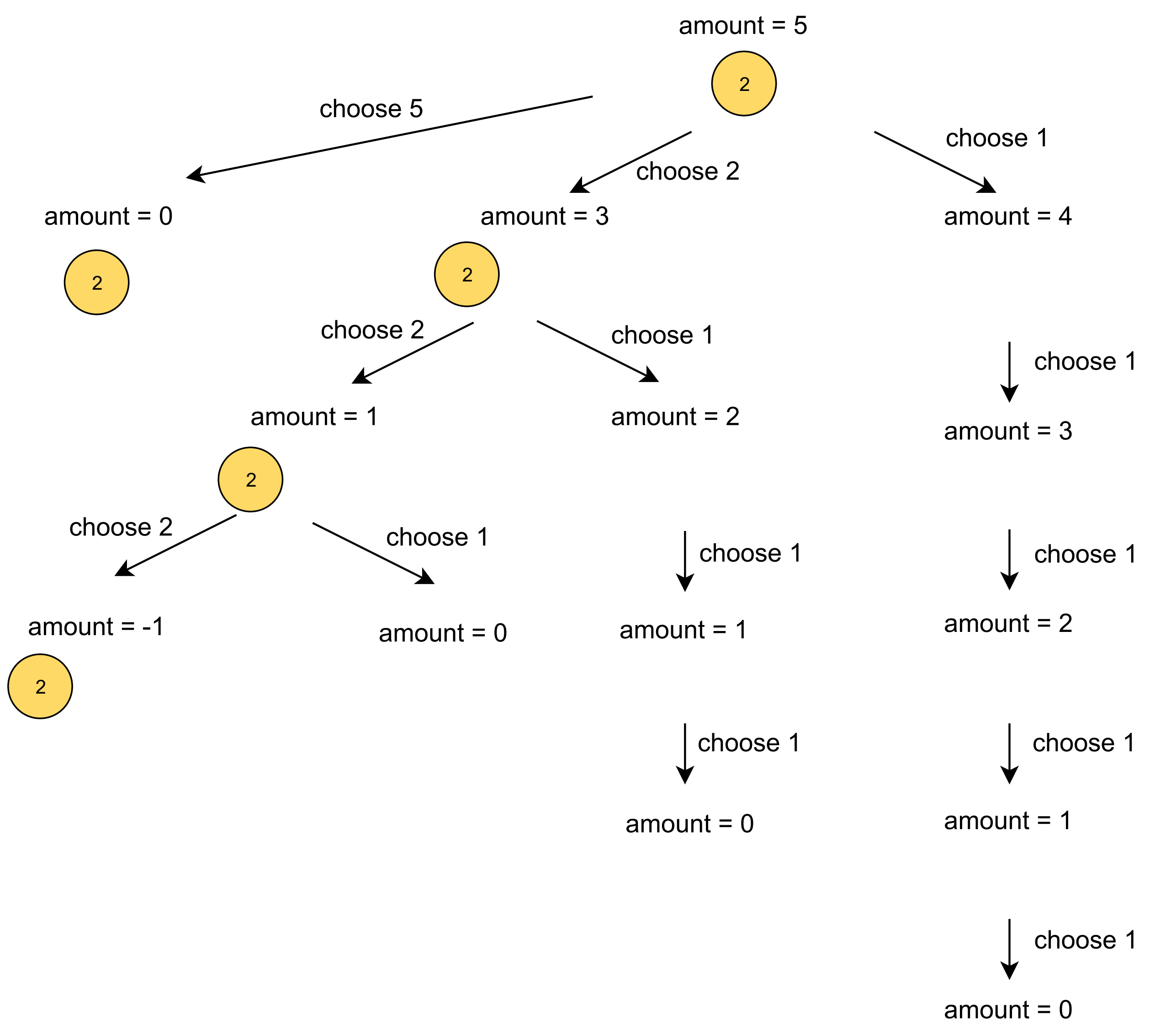
Look at the visualization below to see an overlapping subproblem. We may have a lot more overlapping subproblems in a bigger problem.



computation due to repeating subproblem



same amount and bills



computation due to repeating subproblem

3 of 3

Now that we know this problem obeys both pre-requisites of dynamic programming, let's look at the top-down dynamic programming algorithm.

```
def countways_(bills, amount, maximum, memo):
  if amount == 0:     # base case 1
    return 1
  if amount < 0:      # base case 2
    return 0
  if (amount, maximum) in memo: # checking if memoized
    return memo[(amount, maximum)]
  ways = 0
```

```
  for bill in bills:     # iterate over bills
    # to avoid repetition of similar sequences, use bills smaller than maximum
    if bill <= maximum:

      # notice how maximum becomes bill in recrusive call
      ways += countways_(bills, amount-bill, bill, memo)
  memo[(amount, maximum)] = ways #memoizing
  return ways

def countways(bills, amount):
  memo = {}
  return countways_(bills, amount, max(bills), memo)

print(countways([1,2,5], 5))
```

Explanation

The only addition to this algorithm compared to the simple recursion one is the addition of memoization. We store all the results in `memo` (*line 14*) and then retrieve them as needed (*line 6*). Since we had two defining variables here, amount, and the maximum value, we use them to uniquely identify different subproblems.

We can also write the memoized version of the second algorithm we discussed in solution one. We will memoize using a tuple of `index` and `amount`.

Time and space complexity

How many unique subproblems are possible that we can evaluate? The value of `amount` will not be greater than the one provided at the start of the algorithm, let's call it C. Similarly, since we have n number of different `bills`, the `maximum` variable will thus be bounded by n. Therefore, we will have, at most Cn subproblems. Thus, the time complexity to evaluate these problems would be **O(Cn)** and the space complexity to hold the results of these subproblems would be **O(Cn)** as well.

Solution 3: Bottom-up dynamic programming

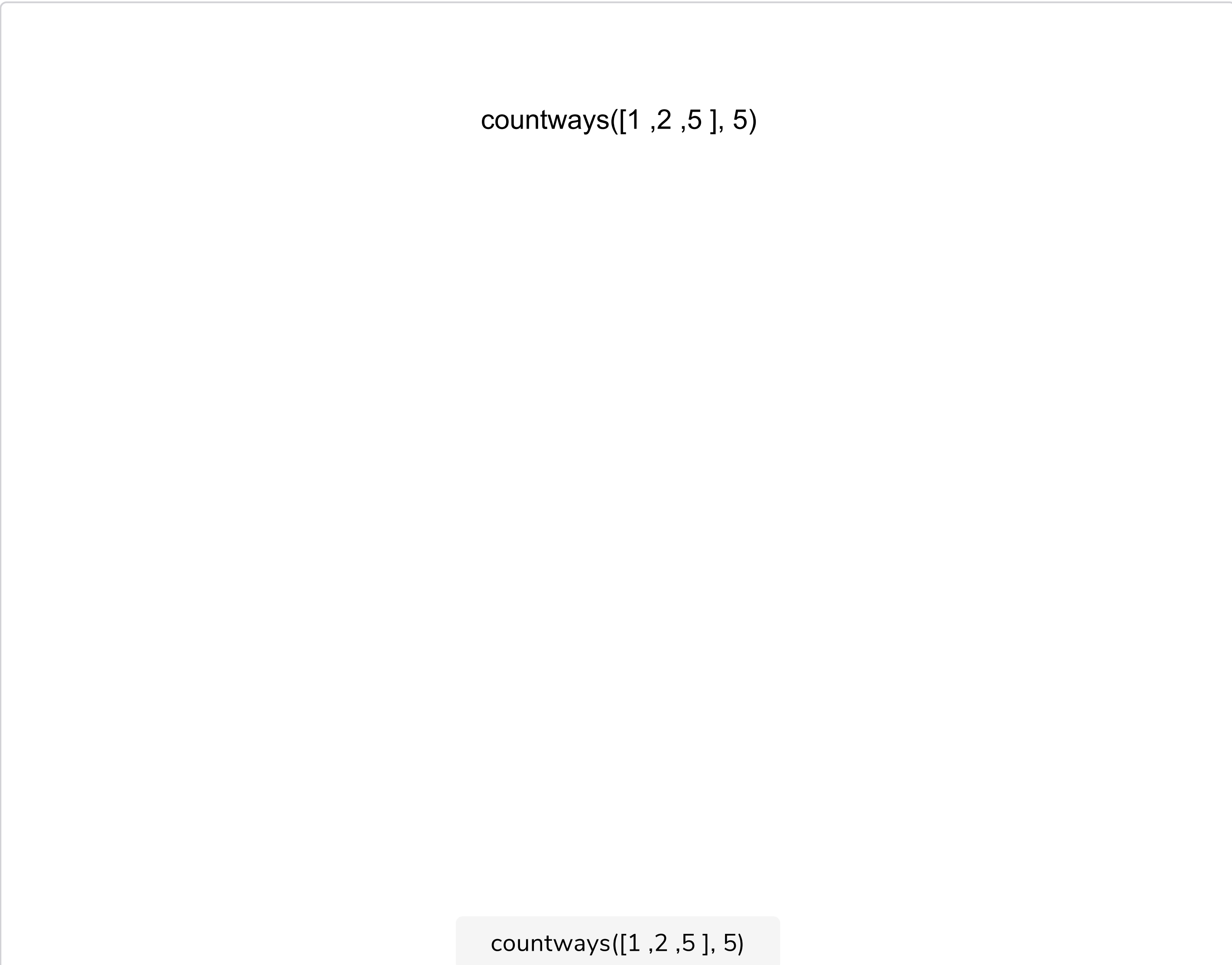
Let's look at a bottom-up algorithm to solve this problem as well.

```
def countways(bills, amount):
  if amount <= 0:
    return 0
  dp = [[1 for _ in range(len(bills))] for _ in range(amount + 1)]
  for amt in range(1, amount+1):
    for j in range(len(bills)):
```

```
      bill = bills[j]
      if amt - bill >= 0:
        x = dp[amt - bill][j]
      else:
        x = 0
      if j >= 1:
        y = dp[amt][j-1]
      else:
        y = 0
      dp[amt][j] = x + y
  return dp[amount][len(bills) - 1]

print(countways([1,2,5], 5))
```

Explanation

This implementation is based on the second algorithm we discussed in solution one. To make up an `amount` using `n` bills, we just need to count the ways in which we can either make the `amount` using the `n`th bill (*lines 8-9*) or without using it (*lines 12-13*). The algorithm is easier to understand with the visualization given below.



Bills

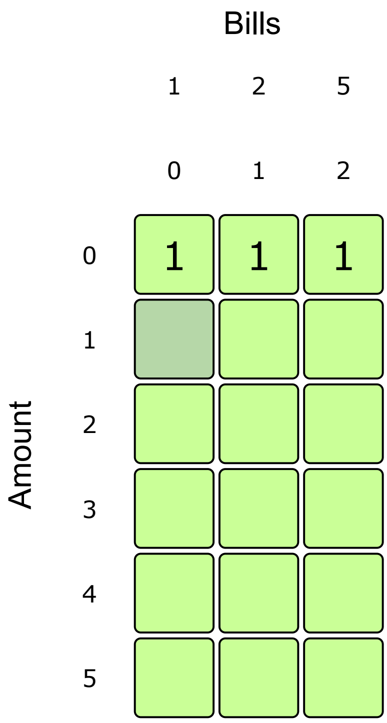
		1	2	5
		0	1	2
Amount	0			
	1			
	2			
	3			
	4			
	5			

make a 2d array of dimensions equal to amount and number of bills

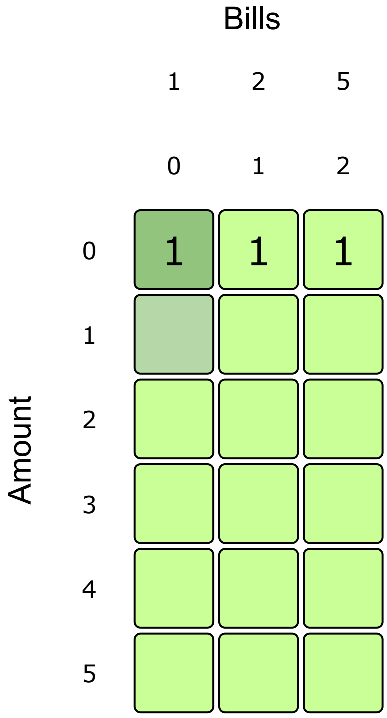
Bills

		1	2	5
		0	1	2
Amount	0	1	1	1
	1			
	2			
	3			
	4			
	5			

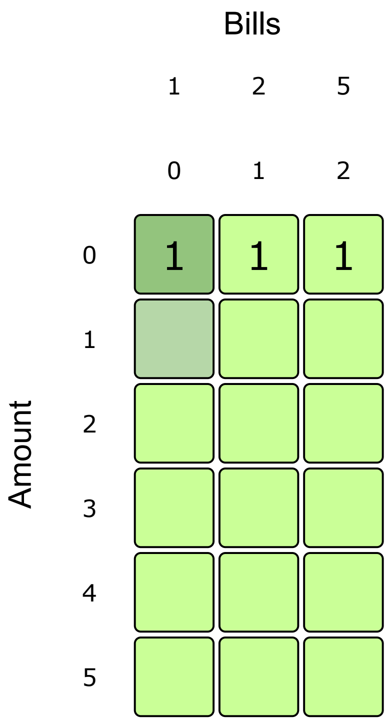
As a base case fill first row up with 1, because 0 amount can be made with any note



how many ways are there to make amount n using d bills?



we need to count number of ways to make the amount by using nth bill



and by not using nth bill, (by not using 1 bill there are 0 ways)

Bills

Amount		1	2	5
		0	1	2
	0	1	1	1
	1	1		
	2			
	3			
	4			
	5			

so we get only 1 here

Bills

Amount	1	2	5
	0	1	2
0	1	1	1
1	1		
2			
3			
4			
5			

moving on

Bills

Amount	1	2	5
	0	1	2
0	1	1	1
1	1		
2			
3			
4			
5			

how many ways are there to make amount 1 using 2 bills (1,2)?

Bills

Amount	1	2	5
	0	1	2
0	1	1	1
1	1		
2			
3			
4			
5			

count the ways to make amount 1 by including bill 2 (which is not possible so 0)

Bills

	1	2	5
Amount	0	1	2
0	1	1	1
1	1		
2			
3			
4			
5			

and by counting the ways to make amount 1 by excluding bill 2

Bills

	1	2	5
Amount	0	1	2
0	1	1	1
1	1	1	
2			
3			
4			
5			

Thus we get 1

Bills

Amount	1	2	5
	0	1	2
0	1	1	1
1	1	1	
2			
3			
4			
5			

moving on

Bills

		1	2	5
		0	1	2
Amount	0	1	1	1
	1	1	1	
	2			
	3			
	4			
	5			

how many ways are there to make amount 1 using 3 bills (1,2,5)?

Bills

		1	2	5
		0	1	2
Amount	0	1	1	1
	1	1	1	
	2			
	3			
	4			
	5			

either by including bill 5 (0 possibilities) or by excluding it

Bills

Amount	1	2	5
	0	1	2
0	1	1	1
1	1	1	1
2			
3			
4			
5			

Thus we get 1

Bills

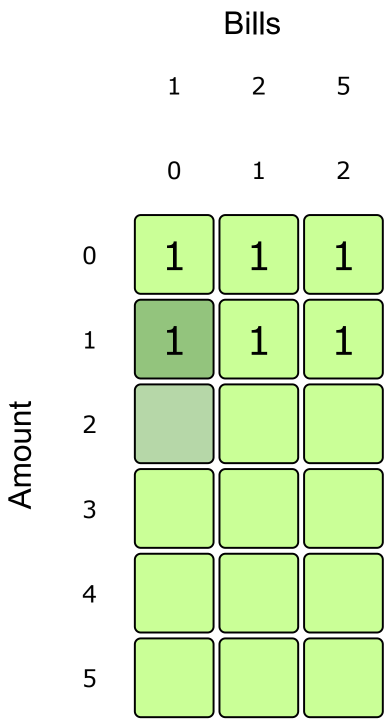
		1	2	5
		0	1	2
Amount	0	1	1	1
	1	1	1	1
	2			
	3			
	4			
	5			

moving on

Bills

		1	2	5
		0	1	2
Amount	0	1	1	1
	1	1	1	1
	2			
	3			
	4			
	5			

how many ways are there to make amount 2 using 1 bill (1)?



either by including bill 1 or by excluding it (0 possibilities)

Bills

		1	2	5
		0	1	2
Amount	0	1	1	1
	1	1	1	1
	2	1		
	3			
	4			
	5			

thus we get 1

Bills

		1	2	5
		0	1	2
Amount	0	1	1	1
	1	1	1	1
	2	1		
	3			
	4			
	5			

moving on

Bills

	1	2	5
Amount	0	1	2
0	1	1	1
1	1	1	1
2	1		
3			
4			
5			

how many ways are there to make amount 2 using 2 bills (1,2)?

Bills

		1	2	5
		0	1	2
Amount	0	1	1	1
	1	1	1	1
	2	1		
	3			
	4			
	5			

Either by including the bill i.e. amount = 0

Bills

		1	2	5
		0	1	2
Amount	0	1	1	1
	1	1	1	1
	2	1		
	3			
	4			
	5			

or by excluding it

Bills

Amount	1	2	5
	0	1	2
0	1	1	1
1	1	1	1
2	1	2	
3			
4			
5			

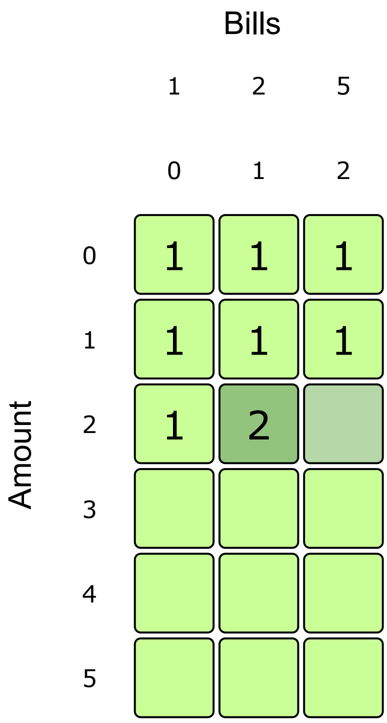
thus we get 2

		Bills		
		1	2	5
Amount		0	1	2
	0	1	1	1
	1	1	1	1
	2	1	2	
	3			
	4			
	5			

moving on

		Bills		
		1	2	5
Amount		0	1	2
	0	1	1	1
	1	1	1	1
	2	1	2	
	3			
	4			
	5			

how many ways are there to make amount 2 using 3 bills (1,2,5)?



either by including bill 5 (0 possibilities) or by excluding it

		Bills		
		1	2	5
		0	1	2
Amount	0	1	1	1
	1	1	1	1
	2	1	2	2
	3			
	4			
	5			

thus we get 2

		Bills		
		1	2	5
		0	1	2
Amount	0	1	1	1
	1	1	1	1
	2	1	2	2
	3			
	4			
	5			

moving on

Bills

		1	2	5
		0	1	2
Amount	0	1	1	1
	1	1	1	1
	2	1	2	2
	3			
	4			
	5			

how many ways are there to make amount 3 using 1 bill (1)?

Bills

Amount	1	2	5
	0	1	2
0	1	1	1
1	1	1	1
2	1	2	2
3			
4			
5			

either by including bill 1 or by excluding it (0 possibilities)

Bills

Amount	1	2	5
	0	1	2
0	1	1	1
1	1	1	1
2	1	2	2
3	1		
4			
5			

Thus we get 1

Bills

Amount	1	2	5
	0	1	2
0	1	1	1
1	1	1	1
2	1	2	2
3	1		
4			
5			

moving on

Bills

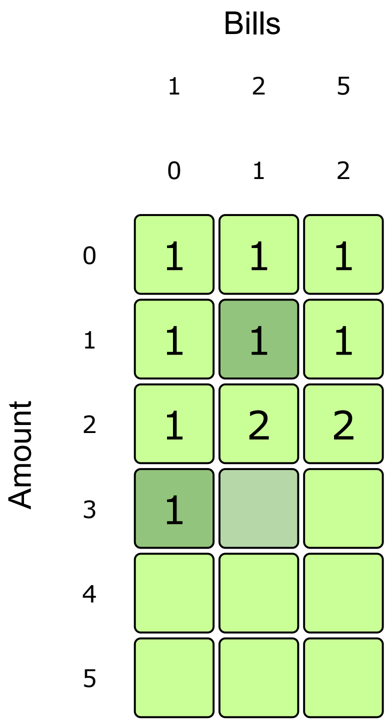
Amount	1	2	5
	0	1	2
0	1	1	1
1	1	1	1
2	1	2	2
3	1		
4			
5			

how many ways are there to make amount 3 using 2 bills (1,2)?

Bills

Amount	1	2	5
	0	1	2
0	1	1	1
1	1	1	1
2	1	2	2
3	1		
4			
5			

Either by including the bill i.e. amount = 1



or by excluding it

Bills

	1	2	5
Amount	0	1	2
0	1	1	1
1	1	1	1
2	1	2	2
3	1	2	
4			
5			

thus we get 2

Bills

	1	2	5
Amount	0	1	2
0	1	1	1
1	1	1	1
2	1	2	2
3	1	2	
4			
5			

moving on

Bills

Amount	1 (0)	2 (1)	5 (2)
0	1	1	1
1	1	1	1
2	1	2	2
3	1	2	
4			
5			

how many ways are there to make amount 3 using 3 bills (1,2,5)?

Bills

Amount	1 (0)	2 (1)	5 (2)
0	1	1	1
1	1	1	1
2	1	2	2
3	1	2	
4			
5			

either by including bill 5 (0 possibilities) or by excluding it

Bills

Amount	1 (0)	2 (1)	5 (2)
0	1	1	1
1	1	1	1
2	1	2	2
3	1	2	2
4			
5			

thus we get 2

Bills

Amount	1	2	5
	0	1	2
0	1	1	1
1	1	1	1
2	1	2	2
3	1	2	2
4			
5			

moving on

Bills

Amount	1	2	5
	0	1	2
0	1	1	1
1	1	1	1
2	1	2	2
3	1	2	2
4			
5			

how many ways are there to make amount 4 using 1 bill (1)?

Bills

Amount	1	2	5
	0	1	2
0	1	1	1
1	1	1	1
2	1	2	2
3	1	2	2
4			
5			

either by including bill 1 or by excluding it (0 possibilities)

Bills

Amount	1	2	5
	0	1	2
0	1	1	1
1	1	1	1
2	1	2	2
3	1	2	2
4	1		
5			

thus we get 1

Bills

		1	2	5
		0	1	2
Amount	0	1	1	1
	1	1	1	1
	2	1	2	2
	3	1	2	2
	4	1		
	5			

moving on

Bills

		1	2	5
		0	1	2
Amount	0	1	1	1
	1	1	1	1
	2	1	2	2
	3	1	2	2
	4	1		
	5			

how many ways are there to make amount 4 using 2 bills (1,2)?

Bills

Amount	1 0	2 1	5 2
0	1	1	1
1	1	1	1
2	1	2	2
3	1	2	2
4	1		
5			

Either by including the bill i.e. amount = 2

Bills

Amount	1	2	5
	0	1	2
0	1	1	1
1	1	1	1
2	1	2	2
3	1	2	2
4	1		
5			

or by excluding it

Bills

Amount	1	2	5
	0	1	2
0	1	1	1
1	1	1	1
2	1	2	2
3	1	2	2
4	1	3	
5			

thus we get 3

Bills

Amount	1	2	5
	0	1	2
0	1	1	1
1	1	1	1
2	1	2	2
3	1	2	2
4	1	3	
5			

moving on

Bills

Amount	1	2	5
	0	1	2
0	1	1	1
1	1	1	1
2	1	2	2
3	1	2	2
4	1	3	
5			

how many ways are there to make amount 4 using 3 bills (1,2,5)?

Bills

Amount	1	2	5
	0	1	2
0	1	1	1
1	1	1	1
2	1	2	2
3	1	2	2
4	1	3	
5			

either by including bill 5 (0 possibilities) or by excluding it

Bills

Amount	1 (0)	2 (1)	5 (2)
0	1	1	1
1	1	1	1
2	1	2	2
3	1	2	2
4	1	3	3
5			

thus we get 3

Bills

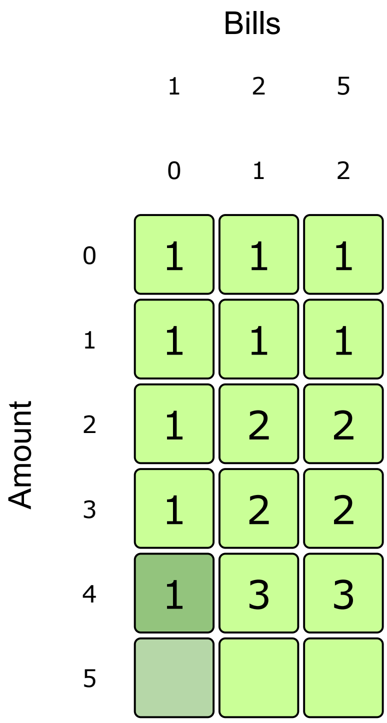
Amount	1	2	5
	0	1	2
0	1	1	1
1	1	1	1
2	1	2	2
3	1	2	2
4	1	3	3
5			

moving on

Bills

Amount	1	2	5
	0	1	2
0	1	1	1
1	1	1	1
2	1	2	2
3	1	2	2
4	1	3	3
5			

how many ways are there to make amount 5 using 1 bill (1)?



either by including bill 1 or by excluding it (0 possibilities)

Bills

Amount	1 (0)	2 (1)	5 (2)
0	1	1	1
1	1	1	1
2	1	2	2
3	1	2	2
4	1	3	3
5	1		

thus we get 1

Bills

Amount	1 (0)	2 (1)	5 (2)
0	1	1	1
1	1	1	1
2	1	2	2
3	1	2	2
4	1	3	3
5	1		

moving on

Bills

Amount	1	2	5
	0	1	2
0	1	1	1
1	1	1	1
2	1	2	2
3	1	2	2
4	1	3	3
5	1		

how many ways are there to make amount 5 using 2 bills (1,2)?

Bills

Amount	1	2	5
	0	1	2
0	1	1	1
1	1	1	1
2	1	2	2
3	1	2	2
4	1	3	3
5	1		

Either by including the bill i.e. amount = 3

Bills

Amount	1	2	5
	0	1	2
0	1	1	1
1	1	1	1
2	1	2	2
3	1	2	2
4	1	3	3
5	1		

or by excluding it

Bills

		1	2	5
		0	1	2
Amount	0	1	1	1
	1	1	1	1
	2	1	2	2
	3	1	2	2
	4	1	3	3
	5	1	3	

thus we get 3

Bills

Amount	1	2	5
	0	1	2
0	1	1	1
1	1	1	1
2	1	2	2
3	1	2	2
4	1	3	3
5	1	3	

moving on

Bills

Amount	1	2	5
	0	1	2
0	1	1	1
1	1	1	1
2	1	2	2
3	1	2	2
4	1	3	3
5	1	3	

how many ways are there to make amount 5 using 3 bills (1,2,5)?

Bills

		1	2	5
		0	1	2
Amount	0	1	1	1
	1	1	1	1
	2	1	2	2
	3	1	2	2
	4	1	3	3
	5	1	3	

Either by including the bill i.e. amount = 0

Bills

	1	2	5
Amount	0	1	2
0	1	1	1
1	1	1	1
2	1	2	2
3	1	2	2
4	1	3	3
5	1	3	

or by excluding it

Bills

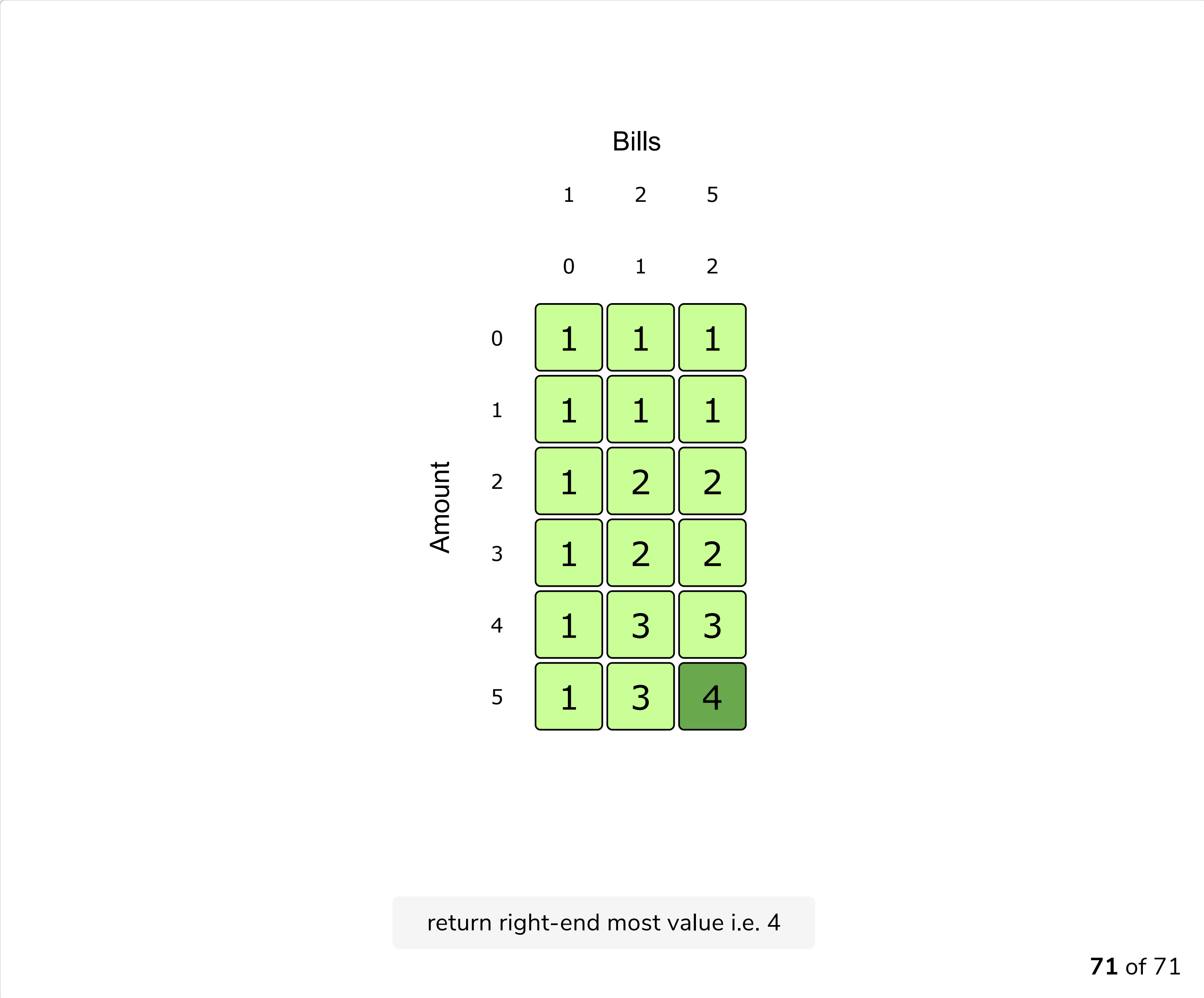
	1	2	5
Amount	0	1	2
0	1	1	1
1	1	1	1
2	1	2	2
3	1	2	2
4	1	3	3
5	1	3	4

thus we get 4

Bills

Amount	1 (0)	2 (1)	5 (2)
0	1	1	1
1	1	1	1
2	1	2	2
3	1	2	2
4	1	3	3
5	1	3	4

table filled



Time and space complexity

As apparent from the visualization as well, we are only iterating over a 2-d array of size `Cxn`, where `C` is the `amount` provided to the algorithm and `n` is the number of bills. Thus, both the time and space complexities of this algorithm are *O(Cn)**.

Solution 4: Space optimized bottom-up dynamic programming

Again, if you notice in the visualization above for filling up a column, we always require the previous column's values, i.e., for filling column against `n` th bill, we require column for `(n-1)` th bill and its column. Thus, there is no point in storing all the previous `(n-2)` columns. The following is the space-optimized algorithm based on this explanation.

```
def countways(bills, amount):
  if amount <= 0:
    return 0
  dp = [1 for _ in range(amount + 1)]
  for j in range(len(bills)):
    thiscol = [1 for _ in range(amount + 1)]
    for amt in range(1,amount + 1):
      bill = bills[j]
      if amt - bill >= 0:
        x = thiscol[amt - bill]
      else:
        x = 0
      if j >= 1:
        y = dp[amt]
      else:
        y = 0
      thiscol[amt] = x + y
    dp = thiscol
  return dp[amount]

print(countways([1,2,5], 5))
```

Explanation

Instead of creating a 2-d array, we only create a 1-d array of size `amount + 1`. Next, we update this 2-d array for each `bill` with the calculation the same as before.

Time and space complexity

The time complexity would remain the same, **O(Cn)**, because we still have to do the calculation for each value amount and each bill. The space complexity, however, reduces to **O(C)** since we are only maintaining an array the size of `amount + 1`.

In the next lesson, we cover another famous dynamic programming problem called the rod cutting problem.