

3 more free questions left!

Upgrade Now ➡

## Imagine you landed a new job as a cashier...

Your quirky boss found out that you're a programmer and has a weird request about something they've been wondering for a long time.

Write a function that, given:

- 1. an amount of money
- 2. a list of coin denominations

computes the number of ways to make amount of money with coins of the available denominations.

**Example:** for amount=4 (4¢) and denominations=[1, 2, 3] (1¢, 2¢ and 3¢), your program would output 4—the number of ways to make 4¢ with those denominations:

- 1. 1¢, 1¢, 1¢, 1¢
- 2. 1¢, 1¢, 2¢
- 3. 1¢, 3¢
- 4. 2¢, 2¢

Gotchas

Type code!

We can do this in  $O(n * m)$  time and  $O(n)$  space, where  $n$  is the amount of money and  $m$  is the number of denominations.

A simple recursive approach works, but you'll find that your function gets called more than once with the same inputs. We can do better.

We could avoid the duplicate function calls by memoizing, but there's a cleaner bottom-up approach.

## Breakdown

We need to find some way to break this problem down into subproblems.

Here's one way: for **each denomination**, we can use it once, or twice, or...as many times as it takes to reach or overshoot the amount with coins of that denomination alone.

For each of those choices of how many times to include coins of each denomination, we're left with the subproblem of seeing how many ways we can get the remaining amount from the remaining denominations.

Here's that approach in pseudocode:

```
def num_ways(amount, denominations):  
    for each denomination in denominations:  
        for each num_times_to_use_denomination in possible_num_times_to_use_denomination_wi  
            answer += num_ways(amount_remaining, other_denominations)
```

The answer for some of those subproblems will of course be 0. For example, there's no way to get 1¢ with only 2¢ coins.

As a recursive function, we could formalize this as:



Type code!

```
def change_possibilities_top_down(amount_left, denominations_left):  
    # base cases:  
    # we hit the amount spot on. yes!  
    if amount_left == 0: return 1  
    # we overshoot the amount left (used too many coins)  
    if amount_left < 0: return 0  
    # we're out of coins  
    if len(denominations_left) == 0: return 0  
  
    print "checking ways to make %i with %s" % (amount_left, denominations_left)  
  
    # choose a current_coin  
    current_coin, rest_of_coins = denominations_left[0], denominations_left[1:]  
  
    # see how many possibilities we can get  
    # for each number of times to use current_coin  
    num_possibilities = 0  
    while (amount_left >= 0):  
        num_possibilities += change_possibilities_top_down(amount_left, rest_of_coins)  
        amount_left -= current_coin  
  
    return num_possibilities
```

But there's a problem—we'll often duplicate the work of checking remaining change possibilities.

Note the duplicate calls with the input 4, [1,2,3]:

Type code!



```
>>> change_possibilities_top_down(4,[1,2,3])
checking ways to make 4 with [1, 2, 3]
checking ways to make 4 with [2, 3]
checking ways to make 4 with [3]
checking ways to make 2 with [3]
checking ways to make 3 with [2, 3]
checking ways to make 3 with [3]
checking ways to make 1 with [3]
checking ways to make 2 with [2, 3]
checking ways to make 2 with [3]
checking ways to make 1 with [2, 3]
checking ways to make 1 with [3]
4
```

For example, we check ways to make 2 with [3] *twice*.

We can do better. How do we avoid this duplicate work and bring down the time cost?

One way is to memoize .

Here's what the memoization might look like:

Type code!



```
class Change:
    def __init__(self):
        self.memo = {}

    def change_possibilities_top_down(self, amount_left, denominations_left):
        # check our memo and short-circuit if we've already solved this one
        memo_key = str((amount_left, denominations_left))
        if self.memo.has_key(memo_key):
            print "grabbing memo[%s]" % memo_key
            return self.memo[memo_key]

        # base cases:
        # we hit the amount spot on. yes!
        if amount_left == 0: return 1
        # we overshot the amount left (used too many coins)
        if amount_left < 0: return 0
        # we're out of coins
        if len(denominations_left) == 0: return 0

        print "checking ways to make %i with %s" % (amount_left, denominations_left)

        # choose a current_coin
        current_coin, rest_of_coins = denominations_left[0], denominations_left[1:]

        # see how many possibilities we can get
        # for each number of times to use current_coin
        num_possibilities = 0
        while (amount_left >= 0):
            num_possibilities += self.change_possibilities_top_down(amount_left, rest_of_coins)
            amount_left -= current_coin

        # save the answer in our memo so we don't compute it again
        self.memo[memo_key] = num_possibilities
        return num_possibilities
```

Type code!

And now our checking has no duplication:

```
>>> Change().change_possibilities_top_down(4, [1, 2, 3])
checking ways to make 4 with [1, 2, 3]
checking ways to make 4 with [2, 3]
checking ways to make 4 with [3]
checking ways to make 2 with [3]
checking ways to make 3 with [2, 3]
checking ways to make 3 with [3]
checking ways to make 1 with [3]
checking ways to make 2 with [2, 3]
grabbing memo[(2, [3])]
checking ways to make 1 with [2, 3]
grabbing memo[(1, [3])]
4
```

This answer is quite good. It certainly solves our duplicate work problem. It takes  $O(n * m)$  time and  $O(n * m)$  space, where  $n$  is the size of amount and  $m$  is the number of items in denominations.

However, we can do better. Because our function is recursive it will build up a **large call stack**<sup>1</sup> of size  $O(m)$ . Of course, this cost is eclipsed by the memory cost of `self.memo`, which is  $O(n * m)$ . But it's still best to avoid building up a large stack like this, because it can cause a **stack overflow** (yes, that means recursion is *usually* better to avoid for functions that might have arbitrarily large inputs).

It turns out we can get  $O(n)$  additional space.

A great way to avoid recursion is to go **bottom-up**<sup>1</sup>

Going **bottom-up** is a way to avoid recursion, saving the **memory cost** that recursion incurs when it builds up the **call stack**.

Put simply, a bottom-up algorithm "starts from the beginning," while a recursive algorithm "starts from the end and works backwards."

Type code!

For example, if we wanted to multiply all the numbers in the range  $1 \dots n$ , we could use this cute, **top-down**, recursive one-liner:

```
def product_1_to_n(n):  
    # we assume n >= 1  
    return n * product_1_to_n(n-1) if n > 1 else 1
```

This approach has a problem: it builds up a **call stack** of size  $O(n)$ , which makes our total memory cost  $O(n)$ . This makes it vulnerable to a **stack overflow error**, where the call stack gets too big and runs out of space.

To avoid this, we can instead go **bottom-up**:

```
def product_1_to_n(n):  
    # we assume n >= 1  
    result = 1  
    for num in range(1, n+1):  
        result *= num  
    return result
```

This approach uses  $O(1)$  space (  $O(n)$  time).

Note that *some* compilers and interpreters will do what's called **tail call optimization** (TCO), where it can optimize *some* recursive functions to avoid building up a tall call stack. Python and Java decidedly do not use TCO. Some Ruby implementations do, but most don't. Some C implementations do, and the JavaScript spec recently *allowed* TCO. Scheme is one of the few languages that *guarantee* TCO in all implementations. In general, best not to assume your compiler/interpreter will do this work for you.

Our recursive approach was top-down because it started with the final value for amount and recursively broke the problem down into subproblems with smaller values for amount. What if instead we tried to **compute the answer for small values of amount first**, and use those answers

Type code!

to iteratively compute the answer for higher values until arriving at the final amount?

We can **start by making an array** `ways_of_doing_n_cents`, where the key is the amount and the value is the number of ways of getting that amount.

This array will take  $O(n)$  space, where  $n$  is the size of amount.

To further simplify the problem, we can work with only the first coin in denominations, then add in the second coin, then the third, etc.

What would `ways_of_doing_n_cents` look like for just our first coin: 1¢? Let's call this `ways_of_doing_n_cents_1`.

```
ways_of_doing_n_cents_1 = [
    0: 1,    # (no coins)
    1: 1,    # (one 1c coin)
    2: 1,    # (2 1c coins)
    3: 1,    # etc...
    4: 1,
    5: 1,
]
```

Now what if we add a 2¢ coin?

```
ways_of_doing_n_cents_1_2 = [
    0: 1,    # no change
    1: 1,    # no change
    2: 1+1,  # new: [(2)]
    3: 1+1,  # new: [(2,1)]
    4: 1+2,  # new: [(2,1,1), (2,2)]
    5: 1+2,  # new: [(2,1,1,1), (2,2,1)]
]
```

Type code!

How do we formalize this process of going from `ways_of_doing_n_cents_1` to



ways\_of\_doing\_n\_cents\_with\_1\_2?

Let's **suppose we're partway through already** (this is a classic dynamic programming approach). Say we're trying to calculate `ways_of_doing_n_cents_1_2[5]`. Because we're going bottom-up, we know we already have:

1. `ways_of_doing_n_cents_1_2` for amounts less than 5
2. a fully-populated `ways_of_doing_n_cents_1`

So how many *new* ways should we add to `ways_of_doing_n_cents_1[5]` to get `ways_of_doing_n_cents_1_2[5]`?

Well, if there are *any* new ways to get 5¢ now that we have 2¢ coins, those new ways must involve at least one 2¢ coin. So if we presuppose that we'll use one 2¢ coin, that leaves us with  $5 - 2 = 3$  left to come up with. We already know how many ways we can get 3¢ with 1¢ and 2¢ coins: `ways_of_doing_n_cents_1_2[3]`, which is 2.

So we can see that:

```
ways_of_doing_n_cents_with_1_2[5] = ways_of_doing_n_cents_1[5] + ways_of_doing_n_cents_1_2[3]
```

**Why don't we also need to check `ways_of_doing_n_cents_1_2[5-2-2]` (two 2¢ coins)?** Because we already checked `ways_of_doing_n_cents_1_2[1]` when calculating `ways_of_doing_n_cents_1_2[3]`. We'd be counting some arrangements multiple times. In other words, `ways_of_doing_n_cents_1_2[k]` already includes the full count of possibilities for getting  $k$ , including possibilities that use 2¢ any number of times. We're only interested in how many *more* possibilities we might get when we go from  $k$  to  $k + 2$  and thus have the ability to add one *more* 2¢ coin to each of the possibilities we have for  $k$ .

## Solution

Type code! We use a bottom-up algorithm to build up a table `ways_of_doing_n_cents` such that

`ways_of_doing_n_cents[k]` is how many ways we can get to `k` cents using our denominations. We start with the base case that there's one way to create the amount zero, and progressively add each of our denominations.

The number of new ways we can make a `higher_amount` when we account for a new coin is simply `ways_of_doing_n_cents[higher_amount-coin]`, where we know that value already includes combinations involving `coin` (because we went bottom-up, we know smaller values have already been calculated).

```
def change_possibilities_bottom_up(amount, denominations):
    ways_of_doing_n_cents = [0] * (amount + 1)
    ways_of_doing_n_cents[0] = 1

    for coin in denominations:
        for higher_amount in xrange(coin, amount + 1):
            higher_amount_remainder = higher_amount - coin
            ways_of_doing_n_cents[higher_amount] += ways_of_doing_n_cents[higher_amount_remainder]

    return ways_of_doing_n_cents[amount]
```

Here's how `ways_of_doing_n_cents` would look in successive iterations of our function for `amount=5` and `denominations=[1, 3, 5]`.

Type code!



```

=====
key:
a = higher_amount
r = higher_amount_remainder
=====

```

```

=====
for coin = 1:
=====
[1, 1, 0, 0, 0, 0]
r a

```

```

[1, 1, 1, 0, 0, 0]
r a

```

```

[1, 1, 1, 1, 0, 0]
r a

```

```

[1, 1, 1, 1, 1, 0]
r a

```

```

[1, 1, 1, 1, 1, 1]
r a

```

```

=====
for coin = 3:
=====
[1, 1, 1, 2, 1, 1]
r a

```

```

[1, 1, 1, 2, 2, 1]
r a

```

Type code!

```

[1, 1, 1, 2, 2, 2]
r a

```

Complexity

$O(n * m)$  time and  $O(n)$  additional space, where  $n$  is the amount of money and  $m$  is the number of potential denominations.



We'll review this one again later



➤ Next question

Like this problem? Pass it on!

 Share  
(https://www.facebook.com/sharer.php?u=https://www.interviewcake.com/question/coin)

 Tweet  
(http://twitter.com/intent/tweet?text=This%20coding%20interview%20question%20on%20@interviewcake%20is%20pretty%20easy%20to%20solve%20on%20interviewcake.com)

   
(https://www.facebook.com/interviewcake)

   
(https://twitter.com/interviewcake)

Copyright © 2015 Cake Labs, Inc. All rights reserved.  
110 Capp St., Suite 200, San Francisco, CA US 94110 (804) 876-2253  
[Privacy \(/privacy-policy\)](#) | [Terms \(/terms-and-conditions\)](#)

Type code!

