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

What if there's *no way* to make the `amount` with the `denominations`? Does your function have reasonable behavior?

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 number_of_ways(amount, denominations):
    answer = 0
    for each denomination in denominations:
        for each num_times_to_use_denomination in possible_num_times_to_use_denomination_wi
            answer += number_of_ways(amount_remaining, other_denominations)
    return answer
```

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:

```python
def change_possibilities_top_down(amount_left, denominations, current_index=0):

    # 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 denominations
    if current_index == len(denominations): return 0

    print "checking ways to make %i with %s" % (amount_left, denominations[current_index:])

    # choose a current coin
    current_coin = denominations[current_index]

    # 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, denominations, curr
        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]:

```
>>> 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:

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

    def change_possibilities_top_down(self, amount_left, denominations, current_index=0):

        # check our memo and short-circuit if we've already solved this one
        memo_key = str((amount_left, current_index))
        if memo_key in self.memo:
            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 denominations
        if current_index == len(denominations): return 0

        print "checking ways to make %i with %s" % (amount_left, denominations[current_inde

        # choose a current coin
        current_coin = denominations[current_index]

        # 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, denominati
            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
```

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, 2)]
checking ways to make 1 with [2, 3]
grabbing memo[(1, 2)]
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`. (Except we'd need to remove the line where we print "checking ways to make..." because making all those sublists will take $O(m^2)$ space!)

However, we can do better. Because our function is recursive it will build up a **large call stack** of size $O(m)$. Of course, this cost is eclipsed by the memory cost of `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**.

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

We can **start by making a list `ways_of_doing_n_cents`**, where the index is the `amount` and the value at each index is the number of ways of getting that amount.

This list 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`.

```python
ways_of_doing_n_cents_1 = [
    1,  # 0c:  no coins
    1,  # 1c:  1 1c coin
    1,  # 2c:  2 1c coins
    1,  # 3c:  3 1c coins
    1,  # 4c:  4 1c coins
    1,  # 5c:  5 1c coins
]
```

Now what if we add a 2¢ coin?

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

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

```python
ways_of_doing_n_cents_1_2[5] = ways_of_doing_n_cents_1[5] + ways_of_doing_n_cents_1_2[5-2]
```

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 calulating `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

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).

Python

```
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_rem

    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=$[1, 3, 5]$.

```
===========
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

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

============
for coin = 5:
=============
[1, 1, 1, 2, 2, 3]
```

```
 r              a


final answer: 3
```

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`.

What We Learned

This question is in a broad class called "dynamic programming." We have a bunch more dynamic programming questions (/concept/bottom-up#related_questions) we'll go over later.

Dynamic programming is *kind of* like the next step up from greedy↴ . You're taking that idea of "keeping track of what we need in order to update the best answer so far," and applying it to situations where the new best answer so far might not *just* have to do with the previous answer, but some *other* earlier answer as well.

So as you can see in this problem, we kept track of *all* of our previous answers to smaller versions of the problem (called "subproblems") in a big list called `ways_of_doing_n_cents`.

Again, same *idea* of keeping track of what we need in order to update the answer as we go, like we did when storing the max product of 2, min product of 2, etc in the highest product of 3 (/question/highest-product-of-3) question. Except now the thing we need to keep track of is *all* our previous answers, which we're keeping in a list.

We built that list bottom-up, but we also talked about how we could do it top-down and memoize. Going bottom-up is cleaner and usually more efficient, but often it's easier to think of the top-down version first and try to adapt from there.

Dynamic programming is a weak point for lots of candidates. If this one was tricky for you, don't fret. We have more coming later.