Your quirky boss collects rare, old coins...

They found out you're a programmer and asked you to solve something they've been wondering for a long time.

Write a function that, given:

1. an `amount` of money
2. an array of coin `denominations`

computes the number of ways to make the `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↴

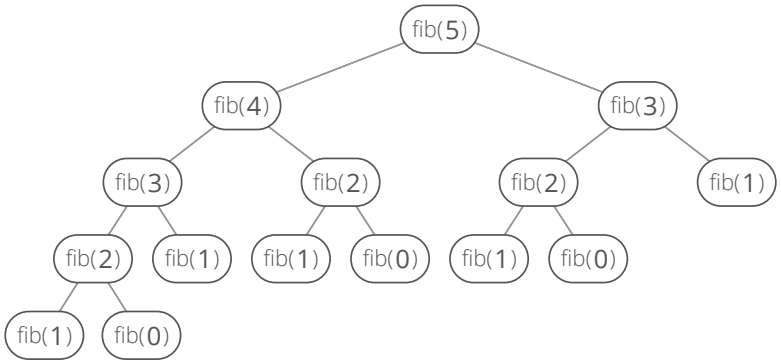
> **Memoization** ensures that a function doesn't run for the same inputs more than once by keeping a record of the results for given inputs (usually in a hash).
>
> For example, a simple recursive function for computing the nth fibonacci number:

```ruby
def fib_recursive(n)

    if n < 0
        raise IndexError, 'Index was negative. No such thing as a negative index in
    end

    # base cases
    if n == 0 || n == 1
        return n
    end

    puts "computing fib_recursive(#{n})"
    return fib_recursive(n - 1) + fib_recursive(n - 2)
end
```

Will run on the same inputs multiple times:

```
001:0> fib_recursive(8)
computing fib_recursive(8)
computing fib_recursive(7)
computing fib_recursive(6)
computing fib_recursive(5)
computing fib_recursive(4)
computing fib_recursive(3)
computing fib_recursive(2)
computing fib_recursive(2)
computing fib_recursive(3)
computing fib_recursive(2)
computing fib_recursive(4)
computing fib_recursive(3)
computing fib_recursive(2)
computing fib_recursive(2)
computing fib_recursive(5)
computing fib_recursive(4)
computing fib_recursive(3)
computing fib_recursive(2)
computing fib_recursive(2)
computing fib_recursive(3)
computing fib_recursive(2)
computing fib_recursive(6)
computing fib_recursive(5)
computing fib_recursive(4)
computing fib_recursive(3)
computing fib_recursive(2)
computing fib_recursive(2)
computing fib_recursive(3)
computing fib_recursive(2)
computing fib_recursive(4)
computing fib_recursive(3)
computing fib_recursive(2)
computing fib_recursive(2)
=> 21
```

We can imagine the recursive calls of this function as a tree, where the two children of a node are the two recursive calls it makes. We can see that the tree quickly branches out of control:



To avoid the duplicate work caused by the branching, we can wrap the function in a class that stores an instance variable, `@memo`, that maps inputs to outputs. Then we simply:

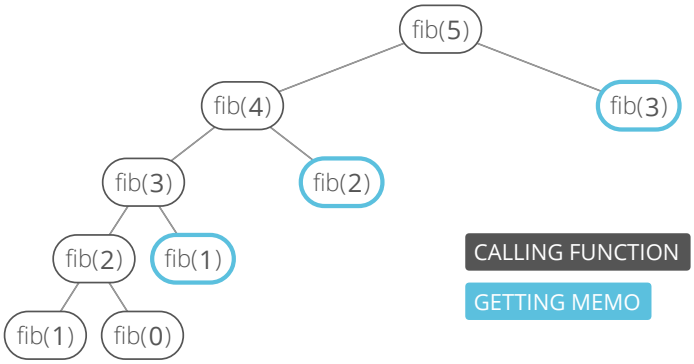
1. Check `@memo` to see if we can avoid computing the answer for any given input, and
2. Save the results of any calculations to `@memo`.

```ruby
class Fibber

    def initialize
        @memo = {}
    end

    def fib(n)

        if n < 0
            raise Exception, "Index was negative. No such thing as a negative index

        # base cases
        elsif n == 0 || n == 1
            return n
        end

        # see if we've already calculated this
        if @memo.include? n
            puts "grabbing memo[#{n}]"
            return @memo[n]
        end

        print "computing fib(#{n})"
        result = self.fib(n - 1) + self.fib(n - 2)

        # memoize
        @memo[n] = result

        return result
    end
end
```

We save a bunch of calls by checking the memo:

```
001:0> Fibber().fib(8)
computing fib(8)
computing fib(7)
computing fib(6)
computing fib(5)
computing fib(4)
computing fib(3)
computing fib(2)
grabbing memo[2]
grabbing memo[3]
grabbing memo[4]
grabbing memo[5]
grabbing memo[6]
=> 21
```

Now in our recurrence tree, no node appears more than twice:



Memoization is a common strategy for **dynamic programming** problems, which are problems where the solution is composed of solutions to the same problem with smaller inputs (as with the fibonacci problem, above). The other common strategy for dynamic programming problems is **going bottom-up (/concept/bottom-up)**, which is usually cleaner and often more efficient.

, but there's a cleaner bottom-up⤵

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 often "starts from the end and works backwards."

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

```ruby
def product_1_to_n(n)
    # we assume n >= 1
    return n > 1 ? n * product_1_to_n(n-1) : 1
end
```

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**:

```ruby
def product_1_to_n(n)
    # we assume n >= 1

    result = 1
    (1..n).each do |num|
        result *= num
    end

    return result
end
```

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

> *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.

Going bottom-up is a common strategy for **dynamic programming** problems, which are problems where the solution is composed of solutions to the same problem with smaller inputs (as with the fibonacci problem, above). The other common strategy for dynamic programming problems is **memoization (/concept/memoization)**.

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)
        end
    end
    return answer
end
```

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:

```ruby
def change_possibilities_top_down(amount_left, denominations, current_index=0)

    # base cases:
    # we hit the amount spot on. yes!
    return 1 if amount_left == 0

    # we overshot the amount left (used too many coins)
    return 0 if amount_left < 0

    # we're out of denominations
    return 0 if current_index == denominations.length

    puts "checking ways to make #{amount_left} with #{denominations[current_index..-1]}"

    # 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
    end

    return num_possibilities
end
```

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]:

```
001:0> 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:

```ruby
class Change
    def initialize
        @memo = {}
    end

    def change_possibilities_top_down(amount_left, denominations, current_index=0)

        # check our memo and short-circuit if we've already solved this one
        memo_key = [amount_left, current_index].to_s
        if @memo.include? memo_key
            puts "grabbing memo[#{memo_key}]"
            return @memo[memo_key]
        end

        # base cases:
        # we hit the amount spot on. yes!
        return 1 if amount_left == 0

        # we overshot the amount left (used too many coins)
        return 0 if amount_left < 0

        # we're out of denominations
        return 0 if current_index == denominations.length

        puts "checking ways to make #{amount_left} with #{denominations[current_index..-1]}

        # 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,
            amount_left -= current_coin
        end

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

And now our checking has no duplication:

```
001:0> Change.new().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 subarrays will take $O(m^2)$ space!)

However, we can do better. Because our function is recursive it will build up a **large call stack**

> The **call stack** is what a program uses to keep track of what function it's currently running and what to do with that function's return value.
>
> Whenever you call a function, a new **frame** gets pushed onto the call stack, which is popped off when the function returns. As functions call other functions, the stack gets taller. In recursive functions, the stack can get as tall as the number of times the function calls itself. This can cause a problem: the stack has a limited amount of space, and if it gets too big you can get a **stack overflow** error.

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 an array `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 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`.

```ruby
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?

```ruby
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:

```ruby
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).

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

    denominations.each do |coin|
        (coin..amount).each do |higher_amount|
            higher_amount_remainder = higher_amount - coin
            ways_of_doing_n_cents[higher_amount] += ways_of_doing_n_cents[higher_amount_rem
        end
    end

    return ways_of_doing_n_cents[amount]
end
```

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 array 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 an array.

We built that array 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.
