Interview Cake

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 method 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 method 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 method gets called more than once with the same inputs. We can do better.

We could avoid the duplicate method 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:

```
public static int numberOfWays(amount, denominations) {
    answer = 0;
    for (denomination : denominations) {
        for (numTimesToUseDenomination : possibleNumTimesToUseDenominationWithoutOvershootingAmount)
            answer += numberOfWays(amountRemaining, otherDenominations);
        }
    }
    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 method, we could formalize this as:

Java

```java
public static int changePossibilitiesTopDown(int amount, int[] denominations) {
    return changePossibilitiesTopDown(amount, denominations, 0);
}

private static int changePossibilitiesTopDown(int amountLeft, int[] denominations, int currentIndex)

    // base cases:
    // we hit the amount spot on. yes!
    if (amountLeft == 0) {
        return 1;
    }

    // we overshot the amount left (used too many coins)
    if (amountLeft < 0) {
        return 0;
    }

    // we're out of denominations
    if (currentIndex == denominations.length) {
        return 0;
    }

    System.out.printf("checking ways to make %d with %s\n",
        amountLeft, Arrays.toString(Arrays.copyOfRange(denominations,
        currentIndex, denominations.length)));

    // choose a current coin
    int currentCoin = denominations[currentIndex];

    // see how many possibilities we can get
    // for each number of times to use currentCoin
    int numPossibilities = 0;
    while (amountLeft >= 0) {
        numPossibilities += changePossibilitiesTopDown(amountLeft, denominations,
            currentIndex + 1);
        amountLeft -= currentCoin;
    }

    return numPossibilities;
}
```

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]:

```
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:

Java

```
class Change {

    private Map<String, Integer> memo = new HashMap<>();

    public int changePossibilitiesTopDown(int amount, int[] denominations) {
        return changePossibilitiesTopDown(amount, denominations, 0);
    }

    private int changePossibilitiesTopDown(int amountLeft, int[] denominations, int currentIndex) {

        // check our memo and short-circuit if we've already solved this one
        String memoKey = amountLeft + ", " + currentIndex;
        if (memo.containsKey(memoKey)) {
            System.out.println("grabbing memo [" + memoKey + "]");
            return memo.get(memoKey);
        }

        // base cases:
        // we hit the amount spot on. yes!
        if (amountLeft == 0) return 1;

        // we overshot the amount left (used too many coins)
        if (amountLeft < 0) return 0;

        // we're out of denominations
        if (currentIndex == denominations.length) return 0;

        System.out.printf("checking ways to make %d with %s\n",
            amountLeft, Arrays.toString(Arrays.copyOfRange(denominations,
            currentIndex, denominations.length)));

        // choose a current coin
        int currentCoin = denominations[currentIndex];

        // see how many possibilities we can get
        // for each number of times to use currentCoin
        int numPossibilities = 0;
        while (amountLeft >= 0) {
            numPossibilities += changePossibilitiesTopDown(amountLeft, denominations,
                currentIndex + 1);
```

```
            amountLeft -= currentCoin;
        }

        // save the answer in our memo so we don't compute it again
        memo.put(memoKey, numPossibilities);
        return numPossibilities;
    }
}
```

And now our checking has no duplication:

```
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 method 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 methods 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 `waysOfDoingNCents`**, 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 `waysOfDoingNCents` look like for just our first `coin`: 1¢? Let's call this `waysOfDoingNCents1`.

Java ▾

```java
int[] waysOfDoingNCents1 = new int[] {
    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?

Java ▾

```java
int[] waysOfDoingNCents1And2 = new int[] {
    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 `waysOfDoingNCents1` to `waysOfDoingNCents1And2`?

Let's **suppose we're partway through already** (this is a classic dynamic programming approach). Say we're trying to calculate `waysOfDoingNCents1And2[5]`. Because we're going bottom-up, we know we already have:

1. `waysOfDoingNCents1And2` for amounts less than 5
2. a fully-populated `waysOfDoingNCents1`

So how many *new* ways should we add to `waysOfDoingNCents1[5]` to get `waysOfDoingNCents1And2[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: `waysOfDoingNCents1And2[3]`, which is 2.

So we can see that:

Java

```
waysOfDoingNCents1And2[5] = waysOfDoingNCents1[5] + waysOfDoingNCents1And2[5 - 2]
```

Why don't we also need to check `waysOfDoingNCents1And2[5 - 2 - 2]` (two 2¢ coins)? Because we already checked `waysOfDoingNCents1And2[1]` when calulating `waysOfDoingNCents1And2[3]`. We'd be counting some arrangements multiple times. In other words, `waysOfDoingNCents1And2[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 `waysOfDoingNCents` such that `waysOfDoingNCents[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 `higherAmount` when we account for a new `coin` is simply `waysOfDoingNCents[higherAmount - coin]`, where we know that value already includes combinations involving `coin` (because we went bottom-up, we know smaller values have already been calculated).

Java

```java
public static int changePossibilitiesBottomUp(int amount, int[] denominations) {
    int[] waysOfDoingNCents = new int[amount + 1];  // array of zeros from 0..amount
    waysOfDoingNCents[0] = 1;

    for (int coin : denominations) {
        for (int higherAmount = coin; higherAmount <= amount; higherAmount++) {
            int higherAmountRemainder = higherAmount - coin;
            waysOfDoingNCents[higherAmount] += waysOfDoingNCents[higherAmountRemainder];
        }
    }

    return waysOfDoingNCents[amount];
}
```

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

```
===========
key:
a = higherAmount
r = higherAmountRemainder
===========

============
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 `waysOfDoingNCents`.

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.