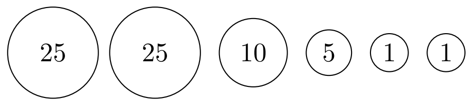
CS 115 Lab 5: Making Change

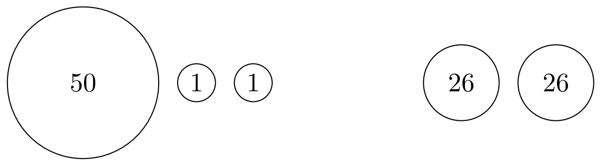
If you've ever worked as a cashier, you've probably heard of the following nighmare scenario: A man walks in, buys $25 worth of goods, and insists on paying in pennies. That's 2500 pennies that you now have to manually count up.

In general, when dealing with coins, most people try to use as few coins as possible. So you use the one-dollar coins until you only have cents left, then you use quarters, then dimes, and so on. For example, the easiest way to make 67 coins is to take two 25-cent quarters, one 10-cent dime, one 5-cent nickel, and two 1-cent pennies.



However, the reason starting from the most expensive works is because US coinage decays sufficiently quickly. In general, taking the largest coins might not always be the best.

To demonstrate, let's suppose that some nation has three types of coins in circulation: Coins worth 1 cent, coins worth 50, and coins worth 26. If you want to count out 52 cents, using the 50 and two 1 cents coins is less efficient than just using two 26 cent coins. *The way to count up a total using the minimal amount of coins does not always include the biggest coin you can use.*



So how do we decide what the best option is? Well, while analysing this by hand might be difficult, we thankfully have computers, which can solve this problem for us!

In this lab, you will use branching recursion to solve this exact task, developing your skills at using it along the way.

Task

Your challenge is to implement the `coin_change` function. This function takes in two arguments.

The first argument is `coins`, which will be a tuple of valid types of coins. For example, the prior example would be represented by the tuple `(50, 26, 1)`, while the standard US coins would be represented by `(100, 5, 10, 25, 1)`. *Note that the coins may not be in sorted order, but the coins will* ***always*** *contain 1!* They are also guaranteed to all be positive integers.

The second argument is `total`, which specifies the total number of cents you need to put together using your coins. In particular, this will always be a non-negative integer, and because there is always a 1-cent coin, you can always put together this total.

Your goal is to output the **number** of coins needed to put together that total. You don't need to say what coins are used to obtain this total, just how many coins you need.

A few demo inputs are listed below, along with an explanation for each.

We suggest doing the CA Check-In early during this lab, as it may help you solve the problem.

```
>>> coin_change((50, 26, 1), 52)
2
# Explanation: 52 = 26 + 26
>>> coin_change((100, 5, 10, 25, 1), 67)
6
# Explanation: 67 = 25 + 25 + 10 + 5 + 1 + 1
>>> coin_change((1, 10, 12, 15), 52)
4
# Explanation: 52 = 10 + 12 + 15 + 15
```

CA Check-In

Pick one of the other branching recursion problems we discussed in class, such as `subset_sum` or the knapsack problem. Explain to the CA what you can use from those problems to help you with this problem.

Tracing analysis

Go to www.pythontutor.com and copy your code for `coin_change` and run it with input `coin_change((5,10,25),25)`. Trace the code until you reach a return value. Use the visualizer to answer the following questions.

1. How does the `coins` value change for frames deeper (further down) in the stack?
2. How does the `total` value change for frames deeper (further down) in the stack?
3. What was the first return value you observed?
4. Continue stepping through the code until you observe *another return value.*
5. Write the code that *called the function* which returned that returned value.

Rubric

Category	**Description**	**Points**
Name	For stating your name on every file for submission.	1
Pledge	For writing the full Stevens Honor Pledge on every file for submission.	3
Collaborators	For filling out the Collaboration Statement.	1
Docstrings	For having detailed docstrings on your code.	10
Check-In	For performing the CA Check-In during the lab.	10
Correctness	For passing our comprehensive test cases.	60
Tracing Analysis	For complete and correct responses to the tracing analysis.	15
Total		100

Submission Instructions

For this lab, you must submit the following files, with exactly the following names:

- `lab5.py`
- `lab5_report.txt`

Remember to put your name and the Stevens Honor Pledge on every file you submit! Failure to put the pledge can result in marks being deducted, with repeat penalties potentially getting stricter between assignments.

Submissions must be handled through Gradescope (accessible through the homework page on Canvas). Grades will be released through Gradescope once the homework is graded.

We expect students to submit their labs to Gradescope and verify that their code passes the public tests before they leave the lab so CAs can help with any submission issues.

Tips, Suggestions, and Reminders

- **Remember to include docstrings!** If your submission does not have docstrings, or has docstrings that are not related to your code, you will not get credit for them.
- **Until we tell you otherwise, you are not allowed to use loops!** We want you to learn to use recursion

- We have provided a test file for you to test your solutions. By running this test file, you should be able to verify that your files are named properly, and that your code works correctly.
- Code that doesn't compile (throws a `SyntaxError` on being run) will receive a 0 for the assignment! Make sure you submit code that can be run, even if it does not pass all the test cases.
- You will want to use branching recursion, but how exactly it should work is left up to you.
- While we promise that every input can be used to form the total, your recursive calls might end in a situation where you cannot possibly form the total. In that case, you should return a special value, indicating that this total is not possible. Given that this outcome is worse than any possible finite number of coins (I'd rather need 1000 coins to make the total than not be able to make it at all), it helps to represent this outcome using `float('inf')`, which gives you a value that is larger than all possible integers.
- As with many branching recursion problems, you will likely run into very large runtimes. In particular, testing totals above 100 is likely to take too much time, so we will not be testing your program on any totals over 100.
- A very tempting way to speed up the algorithm is to use the so-called "greedy" strategy: To take the maximum number of whatever coin you are working on. *This doesn't work!* You can't just assume that a coin is either used as much as possible or not used at all. In some cases, all coins are used in a way where you could use more of them, so you should be really careful with how you handle your cases.
- When making your recursive calls, think carefully about your cases. If you decide to use a specific coin, what should the recursive call look like? And if you decide not to use the coin, what does that mean? In particular, should you choose not to use a coin, I suggest treating it as never using it again, and updating your recursive inputs appropriately.
- You might need more than one base case. In particular, take care to ensure that you never get a negative total. How this can occur, and how to avoid it, is a question we leave up to you.
- The built-in functions `min` and `max` return the smallest and largest of their inputs, respectively. This might be quite handy.
- **Don't forget to add your name and pledge!**