


# CSC110 Tutorial 6: Number Theory

 Print this handout

In this tutorial, you'll get some more practice with the concepts in number theory we covered in this week's lectures. You'll get more practice writing proofs involving the definitions and statements/theorems in number theory, and then translating proofs and concepts into Python programs. (We know you've been doing proofs in other courses as well, but often our students want more practice with proofs, so here we are!)

In Part 1 of this tutorial you'll complete a few different number theory proofs of increasing complexity. Then in Part 2, you'll implement one Python function based on one of the statements from Part 1. And finally in Part 3, you'll explore different implementations of a common operation in number theory: computing the *prime factorization* of a number.

## Part 1: Proofs with number theory, greatest common divisor, and modular arithmetic

In this part, you are given several statements about number theory to prove. For each statement, first read it to make sure that you understand what it's saying (you can try plugging in some concrete numbers for the variables to help), and then start your proof by writing an appropriate *proof header* introducing all variables and assumptions in your proof.

You can (and should) expand *definitions* in your proofs (e.g., of divisibility). Also, for each proof you can use the statements in previous parts as “external facts” in your proof, as long as you are explicit about where you use them. This is good practice in “statement reuse” in proofs, where you look at previous statements you've learned and find ways to use them in your current proof.

You are also given the following theorem from lecture, which you can use in your proofs in this section (though not every proof will require it).

**Theorem (GCD Characterization Theorem).** For all  $m, n \in \mathbb{Z}$  where at least one is non-zero,  $\gcd(m, n)$  is the *smallest positive integer* that can be written as a linear combination of  $m$  and  $n$ .

1. Prove that  $\forall a, b, c \in \mathbb{Z}, a \mid b \wedge b \mid c \Rightarrow a \mid c$ .
2. Prove that  $\forall a, b \in \mathbb{Z}^+, \forall x \in \mathbb{N}, \gcd(x, ab) = 1 \Rightarrow \gcd(x, a) = 1 \wedge \gcd(x, b) = 1$ .  
  
*Hint:* When trying to prove  $\gcd(x, a) = 1$ , you can start by defining an arbitrary common divisor  $d$  of  $x$  and  $a$  (“let  $d \in \mathbb{Z}$  and assume  $d \mid x$  and  $d \mid a$ ”), and then proving that  $d = 1$ .
3. Prove that  $\forall a, b \in \mathbb{Z}^+, \gcd(a, b) = 1 \Rightarrow (\forall n \in \mathbb{Z}, n \text{ is a linear combination of } a \text{ and } b)$ .  
  
*Hint:* use the GCD Characterization Theorem.
4. Prove that  $\forall a, b, x \in \mathbb{Z}, (\gcd(a, b) = 1 \wedge a \mid x \wedge b \mid x) \Rightarrow ab \mid x$ .  
  
*Hint:* This is the hardest proof of this exercise. Introduce all variables carefully, including the ones from expanding the definition of divisibility. Then use the *GCD Characterization Theorem* to write  $1$  as a linear combination of  $a$  and  $b$ , and multiply both sides by  $x$ . Remember that your goal is to find a  $k$  such that  $x = k \cdot ab$ .

## Part 2: From proof to program

Download the starter file [tutorial6.py](#). In the top section of this file, we've provided the *Extended Euclidean Algorithm* as well as a new function `get_linear_combination` for you to implement.

Your task is to implement this function *following the same logic as the proof you wrote for Question 3 in Part 1*, and using the Extended Euclidean Algorithm as a helper function. If you do so, your implementation should be quite short—again showcasing the power of mathematical reasoning to help us develop our algorithms!

## Part 3: Computing prime factorizations using while loops

One of the most important theorems in number theory is the **Fundamental Theorem of Arithmetic**, which says that any integer greater than 1 can be written as a unique product of primes. For example:

$$\begin{aligned} 2 &= 2 \\ 4 &= 2^2 \\ 100 &= 2^2 \cdot 5^2 \\ 1450 &= 2 \cdot 5^2 \cdot 19 \\ 8840 &= 2^3 \cdot 5 \cdot 13 \cdot 17 \end{aligned}$$

In this part, your task is to apply what you've learned about while loops this week to write a function that takes an integer and returns its prime factorization.

```
def prime_factorization(n: int) -> ...:
    """Return the prime factorization of n.

    Preconditions:
        - n > 1
    """
```

Here's an idea for how to develop this algorithm. Suppose we wanted to find the prime factorization for the number 1450.

- First, we find the smallest integer  $> 1$  that divides 1450. That's **2**, since  $1450 = 2 \cdot 725$ .
- Now we repeat with the remaining number 725: find the smallest integer  $> 1$  that divides 725, which is **5** ( $725 = 5 \cdot 145$ ).
- Repeat again, with 145: the smallest integer  $> 1$  that divides 145 is again **5** ( $145 = 5 \cdot 29$ ).
- Repeat again, with 29: the smallest integer  $> 1$  that divides 29 is **29** itself.

The bolded numbers form the prime factorization of our original number:  $1450 = 2 \cdot 5 \cdot 5 \cdot 29 = 2 \cdot 5^2 \cdot 29$ .

Here is how we could represent this repeated calculation using a table:

Iteration	Current number	Prime factorization so far
0	1450	
1	725	2
2	145	$2 \cdot 5$
3	29	$2 \cdot 5^2$
4	1	$2 \cdot 5^2 \cdot 29$

Your task is now to take this idea and turn it into real Python code! We've provided a place for you to complete your work in the second section of [tutorial6.py](#).

Notes:

- We'll leave it to you when implementing this algorithm to decide how to actually represent the prime factorization in Python. There is more than one approach you could take here.
- You should use a while loop to repeat the step of finding the smallest integers that divides the “current” number—and that “current” number should be the loop variable.
- We recommend defining a helper function which takes a positive integer  $n$  and returns the smallest integer greater than 1 that divides  $n$ . You can then call this function inside your while loop.
- Another approach is to use a *nested loop*; try out both approaches, and see which one you like more!
- You might notice that we said “find the smallest integer  $> 1$ ” and not “find the smallest prime number” in our description of the algorithm above. See the additional exercise!

## Additional exercises

1. Prove that for all  $n, d \in \mathbb{Z}$  if  $d$  is the smallest integer  $> 1$  that divides  $n$ , then  $d$  is prime.

You can use a *proof by contradiction* here: start by assuming that there exist  $n, d \in \mathbb{Z}$  such that  $d$  is the smallest integer  $> 1$  that divides  $n$ , and that  $d$  is *not* prime.

2. Prove that for all  $a, b, x_1, x_2 \in \mathbb{Z}^+$ , if all of the following are True:
  - $\gcd(a, b) = 1$
  - $a \mid x_1 \wedge b \mid x_1$
  - $a \mid x_2 \wedge b \mid x_2$
  - $x_1 < x_2$then  $x_2 - x_1 \geq ab$ .

(You may use statements proved in Part 1 of this tutorial.)

3. First, let's review a definition about functions.

Let  $f : A \rightarrow B$ . We say that  $f$  is **one-to-one** when for all  $x_1, x_2 \in A$ , if  $f(x_1) = f(x_2)$  then  $x_1 = x_2$ .

For all  $a, b \in \mathbb{Z}^+$ , we define the function  $f_{a,b} : \{0, 1, \dots, ab - 1\} \rightarrow (\{0, 1, \dots, a - 1\} \times \{0, 1, \dots, b - 1\})$  as follows:

$$f_{a,b}(n) = (n \% a, n \% b)$$

Prove that for all  $a, b \in \mathbb{Z}^+$ , if  $\gcd(a, b) = 1$  then this function  $f_{a,b}$  is one-to-one. Here is the start of a proof header:

Let  $a, b \in \mathbb{Z}^+$ , and define the function  $f_{a,b}$  as above. Let  $n_1, n_2 \in \{0, 1, \dots, ab - 1\}$  and assume  $f_{a,b}(n_1) = f_{a,b}(n_2)$ . We want to show that  $n_1 = n_2$ .

*Hint:* consider  $n_1 - n_2$  and use the statement from the Additional Exercises, Question 2. You want to try to prove that  $n_1 - n_2 = 0$ .