

CSC110 Lecture 16: Greatest Common Divisor, Revisited

David Liu, Department of Computer Science

Navigation tip for web slides: press ? to see keyboard navigation controls.

Announcements & Today's plan

- Assignment 3 has been [posted](#)—please start early!
 - Check out the [A3 FAQ \(+ corrections\)](#)
 - [Additional TA office hours](#)
 - Review [advice on academic integrity](#)
- [PythonTA survey 1](#)

From yesterday: functions, variable reassignment, and object mutation

Story so far

At this point, we've covered the fundamental building blocks of programming in Python.

Data: data types, literals, basic operators, comprehensions, tabular data, data classes

Functions: built-in functions, methods; defining functions, function correctness (pre-/postconditions), testing

Control flow statements: if statements, for loops

Story so far

We've seen how **mathematical properties**

$$\forall p \in \mathbb{Z}, \text{Prime}(p) \Leftrightarrow (p > 1 \wedge (\forall d \in \mathbb{N}, 2 \leq d \leq \sqrt{p} \Rightarrow d \nmid p))$$

can be turned into **algorithms**

```
def is_prime(p: int) -> bool:
    """Return whether p is prime."""
    possible_divisors = range(2, floor(sqrt(p)) + 1)
    return (
        p > 1 and
        all({not divides(d, p) for d in possible_divisors})
    )
```

Story so far

Now, we're ready to begin our study of **more complex algorithms**, that combine data, functions, and control flow statements in non-obvious ways.

Over the next two weeks, we're going to dive into **number theory**, learning about some new **algorithms** that can be used to perform computations with numbers, gcd, and modular arithmetic.

Eventually, we'll learn about some **cryptographic algorithms** that we use to **encrypt** and **decrypt** data to communicate securely, without others being able to eavesdrop.

Today you'll learn to...

1. Define the term **greatest common divisor**.
2. State key properties of the greatest common divisor.
3. Apply these properties to develop the **Euclidean Algorithm** and **Extended Euclidean Algorithm** for computing gcds.
4. Use **while loops** in Python, and differentiate them from for loops.
5. Reason about and document loop behaviour using **loop invariants**.

Computing the Greatest Common Divisor

Definition recap

Let $x, y, d \in \mathbb{Z}$. We say that d is the **greatest common divisor** of x and y when it is the largest number that is a common divisor of x and y , or 0 when x and y are both 0.

We can define the function $\gcd : \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{N}$ as the function which takes numbers x and y , and returns their greatest common divisor.

E.g., $\gcd(100, 72) = 4$ and $\gcd(0, 0) = 0$.

Can we do better?

The Euclidean Algorithm

GCD and remainders

Theorem. For all $a, b \in \mathbb{Z}$, if $b \neq 0$ then $\gcd(a, b) = \gcd(b, a \% b)$.

Key idea: Even if a is very large, $a \% b$ is $< |b|$.

Example: Compute $\gcd(124124124, 110)$.

$$\gcd(124124124, 110) = \gcd(110, 124124124 \% 110) = \gcd(110, 14)$$

$$\gcd(110, 14) = \gcd(14, 110 \% 14) = \gcd(14, 12)$$

$$\gcd(14, 12) = \gcd(12, 14 \% 12) = \gcd(12, 2)$$

$$\gcd(12, 2) = \gcd(2, 12 \% 2) = \gcd(2, 0)$$

$$\gcd(2, 0) = 2 \text{ **(Done!)**}$$

As a “loop table”

Iteration	First number	Second number
0	124124124	110
1	110	14
2	14	12
3	12	2
4	2	0

The Euclidean Algorithm

Given: non-negative integers a and b . **Returns:** $\text{gcd}(a, b)$.

1. Initialize two variables x , y to the given numbers a and b .
2. Let r be the remainder when x is divided by y .
 - i.e., $r = x \% y$
3. Reassign x and y to y and r , respectively.
4. Repeat steps 2 and 3 until y is 0.
5. At this point, x refers to the gcd of a and b .

How do we “repeat until” a condition is met?

The while loop

A **while loop** is a compound statement that repeats its body as long as its `<condition>` is **True**.

```
while <condition>:  
    <statement>  
    ...
```

Implementing the Euclidean Algorithm

Given: non-negative integers a and b . **Returns:** $\text{gcd}(a, b)$.

1. Initialize two variables x , y to the given numbers a and b .
2. Let r be the remainder when x is divided by y .
3. Reassign x and y to y and r , respectively.
4. Repeat steps 2 and 3 until y is 0.
 - Or, repeat steps 2 and 3 **while** y is **not** 0.
5. At this point, x refers to the gcd of a and b .

To PyCharm!

Documenting loops (when there's no accumulator)

The Euclidean Algorithm does not have a traditional “accumulator”: it uses variable reassignment. How can we “understand” it?

A **loop invariant** is a property about loop variables that must be true at the start and end of each loop iteration.

Loop invariants act as documentation and “mini-tests” in loop bodies.

Loop invariant for the Euclidean Algorithm

`gcd(124124124, 110)`

Iteration	x	y
0	124124124	110
1	110	14
2	14	12
3	12	2
4	2	0

At each iteration,

```
gcd(124124124, 110) ==  
gcd(x, y)
```

```
while y != 0:  
    # Loop invariant  
    # gcd(a, b) == gcd(x, y)  
  
    ...
```

One Python challenge: order of reassignments

```
while y != 0
    r = x % y

    x = y
    y = r
```

```
while y != 0
    r = x % y

    y = r
    x = y
```

When reassigning multiple variables, the order of reassignment can make a big difference!

Python improvement: parallel assignment

In Python, we can assign multiple variables using a **parallel assignment statement**.

```
x, y = y, x
```

```
# Or,
```

```
y, x = x, y
```

When the Python interpreter executes a parallel assignment statement, it:

1. Evaluates **every** expression on the right-hand side.
2. Then, it assigns each of the resulting values to the corresponding variable on the left-hand side.

Linear Combinations and the Extended Euclidean Algorithm

Linear combinations

Let $m, n, a \in \mathbb{Z}$. We say that a is a **linear combination of m and n** when there exist $p, q \in \mathbb{Z}$ such that $a = pm + qn$.

For example, 1 is a linear combination of 10 and 7, since

$$1 = (-2) \cdot 10 + 3 \cdot 7$$

A surprising property of gcd

Theorem (GCD Characterization Theorem). Let $m, n \in \mathbb{Z}$, and assume at least one of them is non-zero. Then $\gcd(m, n)$ is the smallest positive integer that is a linear combination of m and n .

Example: $\gcd(10, 7) = 1$, and


$$1 = (-2) \cdot 10 + 3 \cdot 7$$

But how do we find this linear combination?

$$\gcd(124124124, 110) = 2$$

$$2 = 8 \cdot 124124124 + (-9027209) \cdot 110$$

It turns out, somewhat amazingly, that we can modify the Euclidean Algorithm so that it computes both $\gcd(a, b)$ and the linear combination!

This will be the most complex algorithm we've studied to date, so let's get started. 

The Extended Euclidean Algorithm

Given: non-negative integers a and b .

Returns: $\text{gcd}(a, b)$, p , q such that $\text{gcd}(a, b) == p * a + q * b$.

```
def extended_euclidean_gcd(a: int, b: int) -> tuple[int, int, int]:
    """Return the gcd of a and b, and integers p and q such that

    gcd(a, b) == p * a + b * q.

    Preconditions:
    - a >= 0
    - b >= 0

    >>> extended_euclidean_gcd(13, 10)
    (1, 7, -9)
    """
```

```

def extended_euclidean_gcd(a: int, b: int) -> tuple[int, int, int]:
    """Return the gcd of a and b, and integers p and q such that
    gcd(a, b) == p * a + b * q.
    """
    x, y = a, b

    while y != 0:
        assert math.gcd(x, y) == math.gcd(a, b)  # Loop invariant

        r = x % y
        x, y = y, r

    return x, ..., ...  # Need to return "p" and "q" here!

```

Key idea: at each loop iteration, **x** and **y** are linear combinations of **a** and **b**.

```

def extended_euclidean_gcd(a: int, b: int) -> tuple[int, int, int]:
    """Return the gcd of a and b, and integers p and q such that
    gcd(a, b) == p * a + b * q.
    """
    x, y = a, b

    while y != 0:
        assert math.gcd(x, y) == math.gcd(a, b)      # L.I. 1
        # x is a linear combination of a and b      L.I. 2 (NEW)
        # y is a linear combination of a and b      L.I. 3 (NEW)

        r = x % y
        x, y = y, r

    return x, ..., ... # Need to return "p" and "q" here!

```

Okay, but... how do we know what those linear combinations are?

Idea: add new loop variables px, qx, py, qy such that

- $x == px * a + qx * b$ and $y == py * a + qy * b$

```
def extended_euclidean_gcd(a: int, b: int) -> tuple[int, int, int, int]:
    x, y = a, b

    px, qx, py, qy = ..., ..., ..., ... # NEW

    while y != 0:
        assert math.gcd(x, y) == math.gcd(a, b) # L.I. 1
        assert x == px * a + qx * b             # L.I. 2 (NEW)
        assert y == py * a + qy * b             # L.I. 3 (NEW)

        r = x % y
        x, y = y, r

        px, qx, py, qy = ..., ..., ..., ... # NEW

    return x, ..., ...
```


Setting initial values for px, qx, py, qy

```
def extended_euclidean_gcd(a: int, b: int) -> tuple[int, int, int]:  
    x, y = a, b  
  
    px, qx = ..., ... # NEW  
    py, qy = ..., ... # NEW
```

We want:

$$x == \underline{\hspace{1cm}} * a + \underline{\hspace{1cm}} * b$$

$$y == \underline{\hspace{1cm}} * a + \underline{\hspace{1cm}} * b$$

Exercise 2: Completing the Extended Euclidean Algorithm

Summary

Today you'll learned to...

1. Define the term **greatest common divisor**.
2. State key properties of the greatest common divisor.
3. Apply these properties to develop the **Euclidean Algorithm** and **Extended Euclidean Algorithm** for computing gcds.
4. Use **while loops** in Python, and differentiate them from for loops.
5. Reason about and document loop behaviour using **loop invariants**.

Homework

- Readings:
 - From today: 7.1 (prep), 7.2, 7.3
 - For tomorrow: 7.4, 7.5
- Please start [Assignment 3](#)!

```
def find_lecture_meme():  
    while meme is low-quality:  
        look_at_another_meme()
```

**DAVID
LAST NIGHT**

find_lecture_meme()

