A Hopefully Intuitive Explanation of the Euclidean Algorithm

Alysa Meng

1. Motivation

I assume reader knows what prime numbers and factors are, as well as basic programming control structures. Section 2-3 consider mathematical proofs, section 4-5 considers practical application with computational examples, and section 6 considers arguing correctness by intuition in a ”proof-by-picture“ approach.

A greatest common divisor (GCD) between two integers a, b is the largest integer that is a factor of both a, b.

This can be computed by matching the common factors in a prime factorization of a and b.

1.1 Popular approach

Recall this common GCD calculation method: Given $a = a_1 a_2 \cdot a_\ell$ and $b = b_1 b_2 \cdot b_m$ where $a_1, \ldots, a_m, b_1, \ldots b_n$ are prime numbers, we can match factors $p_k = a_i = b_j$ to get some integer $d = p_1 p_2 \cdots p_n$ that we call the greatest common divisor. The uniqueness (up to reordering) of an integer's prime factorization ensures us that this process gives the greatest common divisor between two integers.

1.2 Problems with the popular approach

This factor matching process described above is dependent on having a prime factorization of each integer to begin with. In its general case, prime factorization is a hard problem to solve for people and computers alike. The problem of finding the prime factorization of an integer is not known to have a solution in polynomial time (i.e., it's computationally slow).

1.3 An alternative approach

However, it turns out that prime factorizations are not necessary for determining GCDs. Finding the GCD between two integers can be done in polynomial time (i.e., computers can do this relatively quickly, even for very large integers). One efficient method of finding the greatest common divisor is by running the Euclidean Algorithm, also known as Euclid's Algorithm. It remains uncertain if the algorithm was discovered by Euclid, but it was documented in *Elements*.

The Euclidean Algorithm has served as a basis for other faster GCD-calculating algorithms, but its simplicity and elegance makes the algorithm worthwhile to study on its own.

2. Terminology

We will need some number theory definitions to understand how the Euclidean Algorithm works.

Definition 2.1 *[Divides] Let $a, b \in \mathbb{Z}$. We write $b \mid a$ ("b divides a") if and only if there exists some $k \in \mathbb{Z}$ such that $a = bk$.*

This means that b is a divisor of a.

Definition 2.2 *[Greatest Common Divisor] Let $a, b \in \mathbb{Z}$. We say the integer $d = \gcd(a, b)$ ("d is the greatest common divisor") if and only $d \mid a$ and $d \mid b$ and there is no such integer $c > d$ such that $c \mid a$ and $c \mid b$.*

We will also need a theorem.

Theorem 2.1 *Let $a, b \in \mathbb{Z}$. It follows that there are some $q, r \in \mathbb{Z}$ such that $a = qb + r$, where $0 \leq r < a$.*

e.g., for $a = 9, b = 6$, we can write $9 = 1 \cdot 6 + 3$, where $q = 1, r = 3$.

Definition 2.3 *[Modulo] Let $a, n \in \mathbb{Z}$. We define $a \pmod{n}$ ("a modulo n") to be the remainder r given by the Division Theorem results $a = bn + r$ from above.*

This is more commonly known as the mathematical remainder operation.

3. GCD and Modulo

The Euclidean Algorithm relies on 2 main results:

1. Division Theorem, mentioned above
2. For $a, b \in \mathbb{N}$, $\gcd(a, b) = \gcd(b, a \pmod{b})$ and $\gcd(a, 0) = a$

We will now explore this second fact.

First, consider why $\gcd(a, 0) = a$ must be true. The GCD of a and 0 is the integer d such that $d \mid a$ and $d \mid 0$. Notice that $d = a$ satisfies this condition. Why do we know a is the greatest choice for d? Any integer greater than a does not divide a.

Next, consider why $\gcd(a, b) = \gcd(b, a \pmod{b})$ must be true.

When $b > a$, $a \pmod{b} = a$ so the fact trivially holds since $\gcd(a, b) = \gcd(b, a) = \gcd(b, a \pmod{b})$.

When $b = a$, the GCD is $\gcd(a, b) = a = b$ since the GCD is at most $a = b$ and clearly $a \mid a$ and $b \mid b$. So the formula works since $\gcd(a, b) = \gcd(b, 0) = b$

When $b < a$, the GCD is at most b so we can let b be the new upper bound. However, we still need information about a for the GCD. Let's consider $a - b$, since that will make a smaller and we can easily get a back by adding b, so some notion of a is preserved. If two integers have a shared integer factor, their sum/difference will also have that factor. This gives us the nice result that $\gcd(a, b) \mid a$ and $\gcd(a, b) \mid b$, so we have $\gcd(a, b) \mid a - b$. This last fact is significant because we've found that $\gcd(a, b) = \gcd(b, a - b)$. In fact, it doesn't matter how many multiples

$q \in \mathbb{N}$ of b we subtract from a. We'll still have $\gcd(a, b) \mid a - qb$, so we'll pick q to be the largest integer possible to minimize $a - qb$ while maintaining a positive value. It is convenient that such q is given by the Division Theorem with $0 \leq r = a - qb < b$. This can be written as the desired result that $\gcd(a, b) = \gcd(b, a \pmod b)$.

In the end, the takeaway of this argument is that $a \pmod b$ is sufficient information about a to compute $\gcd(a, b)$.

4. Euclidean Algorithm

The Euclidean Algorithm is a procedure with logarithmic time complexity, meaning the number of recursive calls roughly grows with respect to $\log(\min(a, b))$. It is outlined in the following pseudocode, given positive integers a, b:

```
function FindGCD(a, b)
    if b = 0
        return a
    else
        return FindGCD(b, a (mod b))
```

Why does this work mathematically? The Division Theorem ensures that the remainders in the calculation strictly decrease, i.e., $a \pmod b < b$. Therefore, the algorithm terminates in a finite amount of steps. The if-else control structure aligns with the cases of $\gcd(a, 0) = a$ and $\gcd(a, b) = \gcd(b, a \pmod b)$ that terminate the algorithm and recurse into a subcalculation, respectively. The correctness of the algorithm can be more formally proven using mathematical induction on the number of steps needed to find the GCD.

For instance, if we wanted to compute $\gcd(50, 30)$ according the the algorithm, we'd do:

```
gcd(50, 30)
-> gcd(30, 50 (mod 30)) = gcd(30, 20)
    -> gcd(20, 30 (mod 20)) = gcd(20, 10)
        -> gcd(10, 20 (mod 10)) = gcd(10, 0)
            -> 10
```

More succinctly, $\gcd(50, 30) = \gcd(30, 20) = \gcd(20, 10) = \gcd(10, 0) = 10$.

Notice that the structure of sequence of GCD calls encodes a minimal amount of information (a dividend and a remainder), but we can preserve more information (e.g., the quotient) in each step by writing each step in the Division Theorem equation form.

$$\begin{aligned} 50 &= 1 \cdot 30 + 20 \\ 30 &= 1 \cdot 20 + 10 \\ 20 &= 2 \cdot 10 + 0 \end{aligned}$$

This format for showing GCD calculations is also known as a Tableau Form.

5. Application: Extended Euclidean Algorithm

The Euclidean Algorithm opens the gates to solving other problems in number theory.

Sometimes we care about not only finding the GCD of two integers a, b but also finding a linear combination of a, b that yields the GCD. That is, finding the solutions $s, t \in \mathbb{Z}$ to $\gcd(a, b) = sa + tb$.

It turns out that the Euclidean Algorithm can be extended to both show existence of and find such linear combination.

Consider the Tableau form generated from computing the GCD of 50 and 30 above.

$$\begin{aligned} 50 &= 1 \cdot 30 + 20 \\ 30 &= 1 \cdot 20 + 10 \\ 20 &= 2 \cdot 10 + 0 \end{aligned}$$

Notice that $a = 50, b = 30$ appear in the first equation, and $\gcd(a, b) = \gcd(50, 30) = 10$ appears in the second-to-last and last equations.

If we want to find $s, t \in \mathbb{Z}$ such that $10 = s50 + t30$, the equations in the Tableau form above are sufficient since the first equation and second-to-last equations are related by 20. While the last equation does illustrate that 10 is the GCD because the remainder is 0, it doesn't add extra information about the relationship between 10, 50, 30 that we need, so we can ignore it.

To do this, isolate 10 and 20:

$$\begin{aligned} 50 &= 1 \cdot 30 + 20 & &\implies 50 - 1 \cdot 30 = 20 \\ 30 &= 1 \cdot 20 + 10 & &\implies 30 - 1 \cdot 20 = 10 \end{aligned}$$

Now see that starting with the second equation, substituting the first, and then simplifying yields:

$$\begin{aligned} 10 &= 30 - 1 \cdot 20 & &\text{Second equation} \\ &= 30 - 1 \cdot (50 - 1 \cdot 30) & &\text{Substitute 20 with the first equation} \\ &= 30 + (-1) \cdot 50 + 1 \cdot 30 & &\text{Distributing the } -1 \\ &= -1 \cdot 50 + 2 \cdot 30 & &\text{Combining colored terms} \end{aligned}$$

So we've found $s = -1, t = 2$ that are integer solutions for $10 = s50 + t30$.

This process generalizes for any $a, b \in \mathbb{N}$ as all we've used is the fact that the first equation has a, b and the second-to-last equation has $\gcd(a, b)$, with the intermediate equations always having two "colored" terms in common (allowing this substitution to work).

The correctness of the back-substitution procedure, often called the Extended Euclidean Algorithm, can be proved using induction similar to how it can be done for the Euclidean Algorithm.

6. Not yet convinced that the Extended Euclidean Algorithm does what we want?

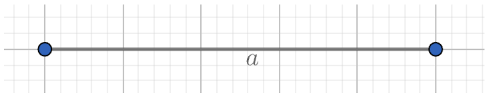
The two promises about the Extended Euclidean Algorithm made in earlier sections are the following:

1. It finds $\gcd(a, b)$ for $a, b \in \mathbb{N}$.
2. And it finds $s, t \in \mathbb{Z}$ such that $sa + tb = \gcd(a, b)$

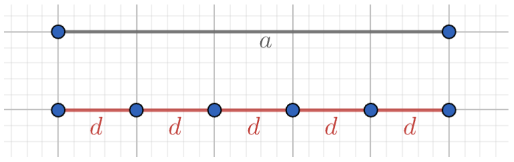
6.1 Background for visualization

Let's step through the mechanics visually.

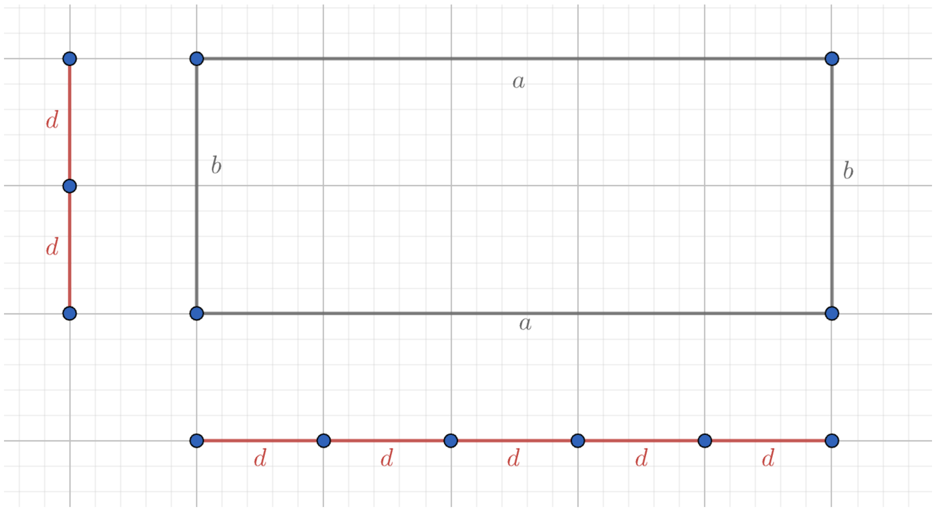
We can represent a positive integer a by a segment of length a.

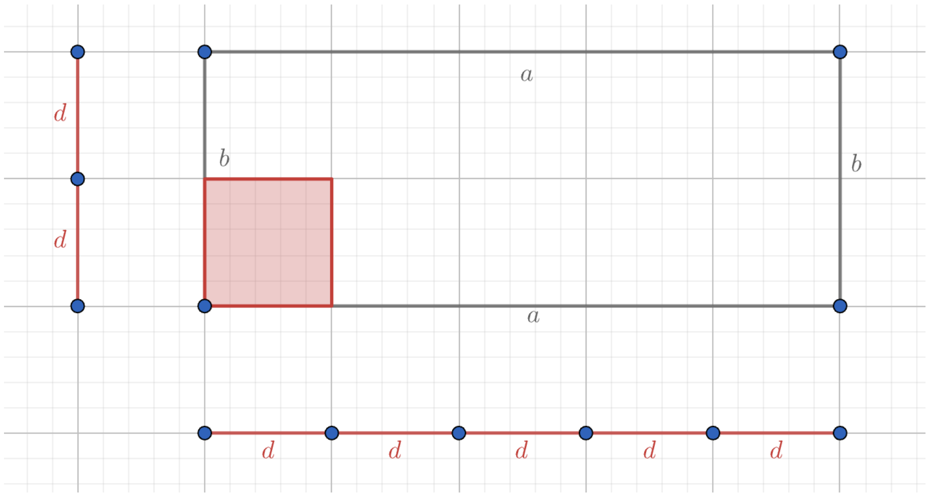


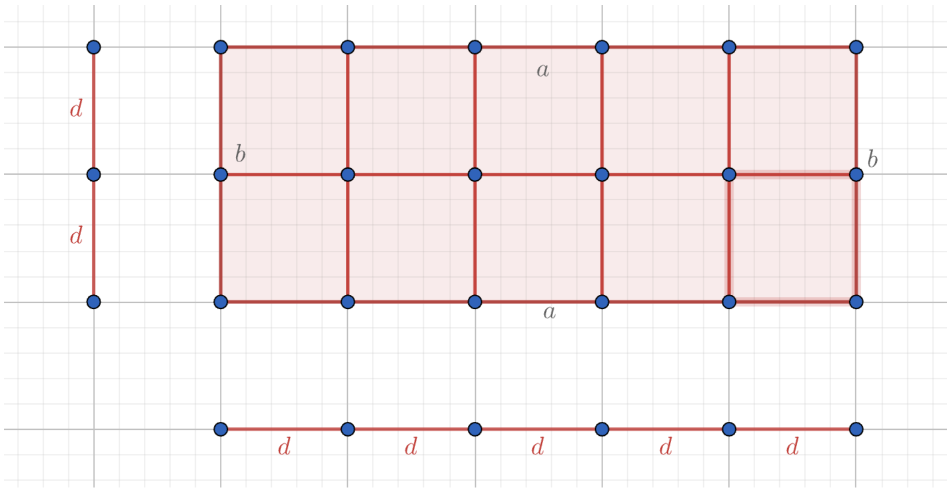
We can represent a divisor d that divides a by another line segment of length d such that when duplicating it some (integer) number of times, it perfectly matches the length of a.



This lends us to a visualization of a common divisor of a, b: a square with side length d such that when duplicating it some (integer) number of times, it perfectly fills the rectangle of side lengths a, b.



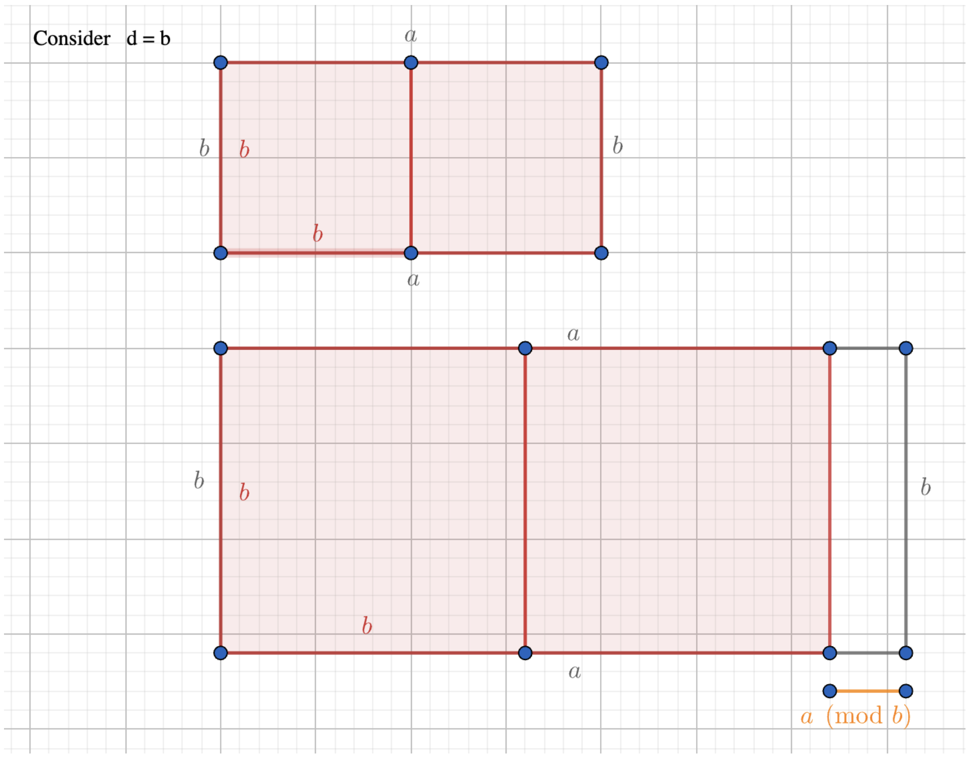




gcd(a, b) then corresponds to the largest such side length for this square.

6.2 Finding the GCD

Assuming that $a \geq b$, let's try to fit as many squares of side length b as we can, since a square with side length b is the largest square possible that fits!

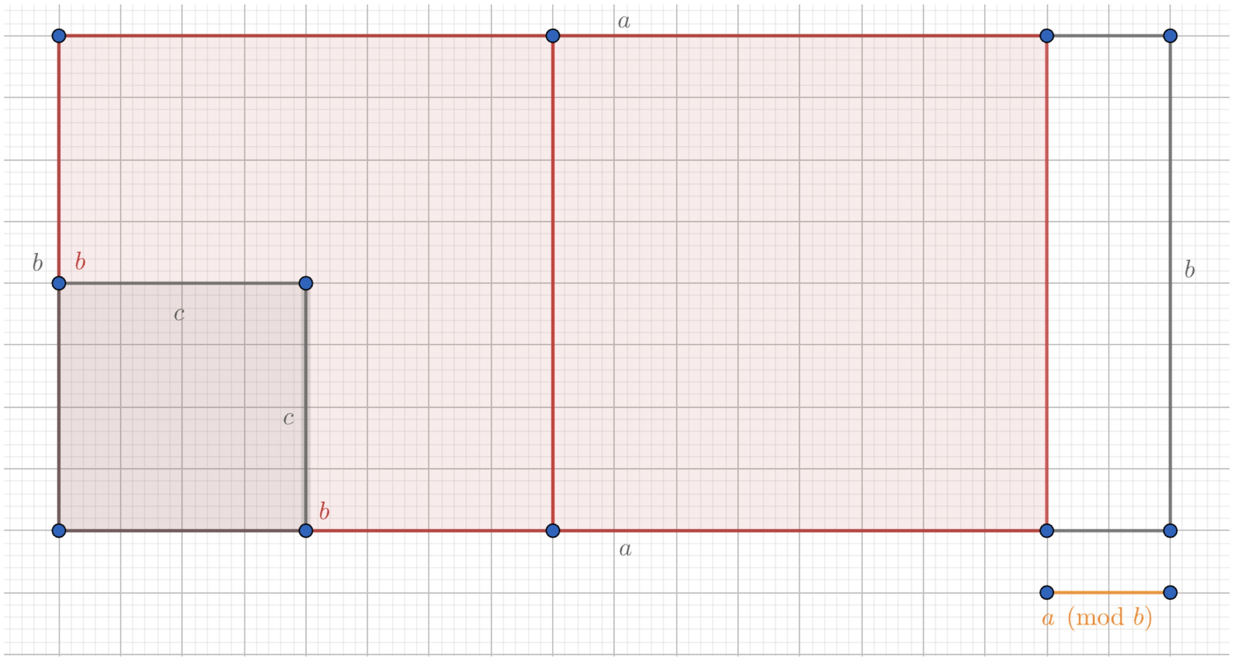


If an integer number of squares with side length b fits as shown in the top case, that's the GCD.

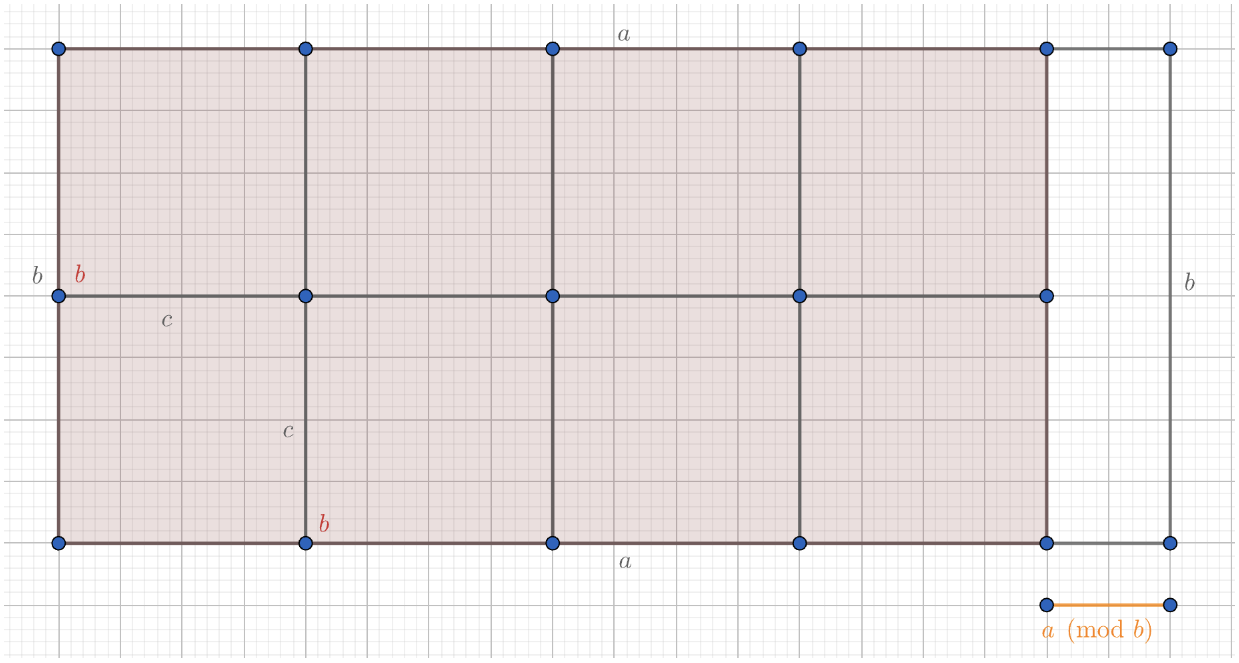
But if it doesn't as shown in the bottom case, we've zoomed in on a smaller area that we can consider and $\gcd(a, b) < b$.

In turns out that the extra part preventing a perfect fit is the division theorem remainder $a \pmod b$, and $\gcd(a, b) \leq a \pmod b < b$.

How do we know that $a \pmod b$ is indeed the new upper bound and by extension the next largest candidate that we should consider? Consider if this were not the case. That is, we have some larger divisor of b, denoted by c that divides a.

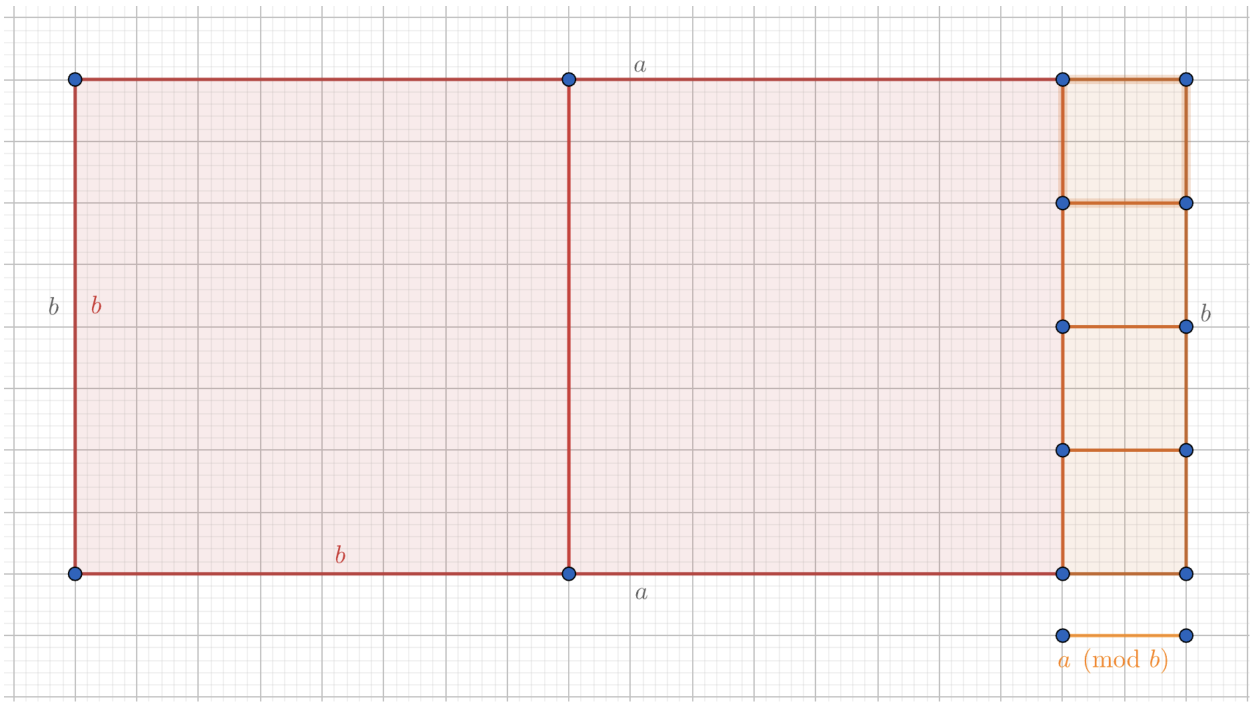


Let's try to fit an integer number of squares of length c into the rectangle.



As c divides b, squares of side length c will fill the squares of b perfectly, but since $c > a \pmod b$, it won't fill the extra remainder portion that squares of length b failed to fill either. Therefore, $a \pmod b$ is indeed the upper bound for $\gcd(a, b)$.

So repeating this process not only yields a common divisor but also the greatest common divisor — in this case $\gcd(a, b) = a \pmod b$.

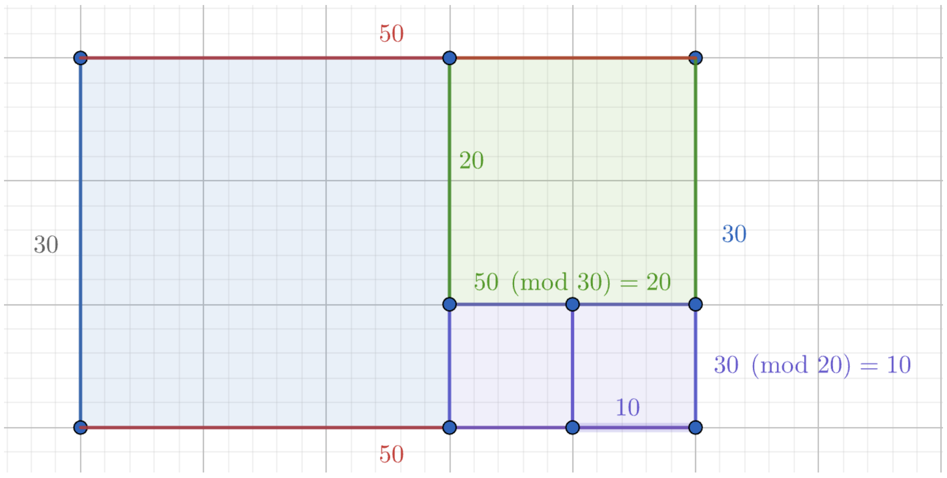


We see $a \pmod{b} \mid b$ because an integer number of segments of length $a \pmod{b}$ perfectly align with the segment of length of b. It follows that an integer k number of line segments of length $a \pmod{b}$ perfectly fits into qb where $q \in \mathbb{N}$, and we broke the line segment of length $a = bq + a \pmod{b}$ in the image (namely, $q = 2$). So $k + 1$ segments of length $a \pmod{b}$ perfectly aligns with the length a.

6.3 Finding $s, t \in \mathbb{Z}$ such that $sa + tb = \gcd(a, b)$

The second promise was that the Extended Euclidean Algorithm finds $s, t \in \mathbb{Z}$ such that $sa + tb = \gcd(a, b)$.

The intuition for this general result is best illustrated by fixing an example. Let's revisit the previous $\gcd(50, 30)$ example in visual form to understand how $s, t \in \mathbb{Z}$ are found. We can see that the rectangle visualization nicely gives the Tableau form equations by reversing the logic used to generate the visualization.



First, focus on the right vertical side composed of green and purple segments to span the length of a blue segment. Starting with $\gcd(50, 30) = 10$, we see remainder 10 as a result of fitting 1 20 into 30. This means that 10 can be expressed as $30 - 1 \cdot 20$.

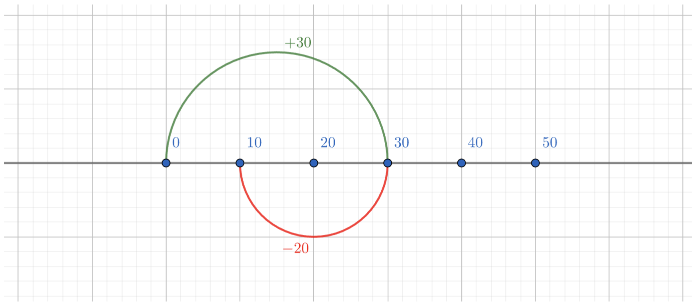
We have the original vertical rectangle side length, 30, in the expression $10 = 30 - 1 \cdot 20$; we still need the original horizontal length, 50. Focus on the top vertical side composed of blue and green segments to span the length of a red segment. We see remainder 20 as as result of fitting 1 30 into 50. This means that 20 can be expressed as $50 - 1 \cdot 30$.

We have the same two relationships that we had before: $10 = 30 - 1 \cdot 20$ and $20 = 50 - 1 \cdot 30$.

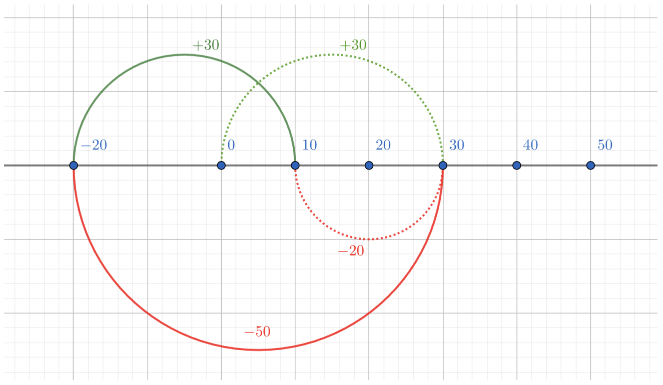
But what's going on when we do substitution and simplification to get $10 = -1 \cdot 50 + 2 \cdot 30$?

While $s = -1, t = 2$ checks out algebraically, one may wonder what the back-substitution looks like geometrically.

The first equation we consider is $10 = 30 - 1 \cdot 20$. We want to get to 10, and we do this by jumping forward 30 and back 20.



We don't want a hop of 20 and we know from the second equation, $20 = 50 - 1 \cdot 30$, that hopping forward 20 is exactly a hop forward 50 and a hop back 30. Since we want to hop back 20, we instead do the reverse: hop back 50 with hop forward 30.



We can see that $s = -1$ is the one hop back of 50, and $t = 2$ is the total two hops forward of 30, landing us at $\gcd(50, 30) = 10$.