Modular Arithmetic: A Simple Introduction

Óscar Pereira*

Abstract. When I first learned this topic, I remember thinking that the usual way it was presented in books and lecture notes, was needlessly complicated. Years later I had to *teach* it to undergrads—and began noticing that they were thinking the same way as my younger self had. Thus I wrote a small note explaining modular arithmetic the way I would have liked it had been explained to me—and what follows is the polished version of that text.

Keywords: integer division, divisibility, modular arithmetic, Euclidean algorithm.

1 Integer Division

We recall the notions of *divisibility* and *integer division*—which are the foundational concepts on top of which modular arithmetic is constructed.

Definition 1.1 (Divisibility). *Given two integers* a *and* b*, we say that* a **divides** b*, or equivalently, that* b **is a multiple of** a*, if there exists an integer* k *such that* $b = ka$*. In which case we write* $a \mid b$.

Theorem 1.2 (Integer division). *Given integers* a *and* $b > 0$*, there exist unique integers* q *and* r*, with* $0 \leq r < b$*, such that* $a = bq + r$.

The proof is not needed to understand the sequel, and can be omitted on a first reading.

*CONTACT: {https://, oscar@}gd7.eu. DATE: October 23, 2025.
Updated versions of this document and other related information can be found at https://gd7.eu/scholarship/intro-mod-arithmetic.

Proof. Consider the set $S = \{a - bt : t \in \mathbb{Z} \wedge a - bt \geq 0\}$. S is nonempty: if $a \geq 0$, set $t = 0$, otherwise set $t = a$, to obtain $a - ba = a(1 - b)$ which is non-negative, because the first multiplicative factor is negative, and the other is either zero, or negative. From the well-ordering principle,[1] it follows that S must have a smallest element; let that element be $r = a - bq$. We need to show that $r < b$ and that r and q are unique.

To show that $r < b$, suppose that that was *not* the case; i.e. suppose that $r \geq b$. Then $r - b \geq 0$, and as also $r - b = a - bq - b = a - b(q + 1)$, we conclude that $r - b \in S$. But $r - b < r$, and r was supposed to be S's smallest element—which shows our supposition that $r \geq b$ cannot be true. Hence, $r < b$.

To show the uniqueness of q and r, let q', r' be such that $a = bq + r = bq' + r'$, with $0 \leq r, r' < b$. Rearranging terms we obtain $r' - r = b(q - q')$, and thus $b \mid r' - r$. But $-(b - 1) \leq r' - r \leq b - 1$, and the only multiple of b in that range is 0—which immediately gives $r' - r = 0 \Leftrightarrow r' = r$ and thus $q = q'$. ■

Definition 1.3. *Given two integers* a *and* b*, not simultaneously zero, their* **greatest common divisor** *(gcd) is the greatest integer* d *such that it divides both* a *and* b.

The set of divisors of any integer other than zero is always *finite*: indeed, no divisor of any nonzero integer a can be either greater than $|a|$, or smaller than $-|a|$. Hence, when a and b are both nonzero, any common divisor of both cannot be either greater than $\min(|a|, |b|)$, or smaller than $\max(-|a|, -|b|)$. Or put differently, the set of common divisors of nonzero integers a and b, is also always finite. Thus, as any finite set of integers has a (unique) largest element, the gcd is well-defined for pairs of nonzero integers. When (exactly) one of a or b is zero, it is straightforward to see that $\gcd(a, 0) = \gcd(0, a) = |a|$ holds for any integer $a \neq 0$. The remaining case—$\gcd(0, 0)$—we leave undefined. The following remark is also immediate.

Remark 1.4. The greatest common divisor, when defined, is always positive. △

Proposition 1.5. *Let* a *and* b *be two integers (not both zero). Then* $\gcd(a, b)$ *is the smallest positive integer that can be expressed as a linear combination of* a *and* b.

Proof. Consider the set $S = \{r, s \in \mathbb{Z} | ar + bs \geq 1\}$. This set is not empty (e.g. make $r = a$ and $s = b$); thus by the well-ordering principle, it contains a smallest

element. Let $d = ax+by$ be that element. Dividing a by d, we get $a = dq+r \Leftrightarrow a = (ax + by)q + r \Leftrightarrow r = a(1 - xq) + b(-yq)$. Thus the remainder is also a linear combination of a and b—which means that if $r > 0$, then $r \in S$. But r must be smaller than d, and d is, by hypothesis, supposed to be the smallest element of S—so r cannot belong to S. Hence we conclude that $r = 0$ (i.e. $d \mid a$). With b a similar reasoning shows that $d \mid b$—and thus d is a common divisor of both a and b.

Now we must show that every common divisor of a and b that is distinct from d, is also smaller than d. Let $d' \neq d$ be one such common divisor. If it is negative, we are done—otherwise, as we cannot have $d' = 0$, it must be the case that d' is positive.[2] And as any number that divides a and b must divide any linear combination of theirs, so we must have $d' \mid d$. I.e., there exists $k \in \mathbb{Z}$ such that $d'k = d$—and as d' and d are both positive and distinct, so must k be not only positive, but also greater than 1, entailing that $d' < d$. ■

The next corollary follows immediately from the proof above (the next two corollaries both assume that $\gcd(a, b)$ is defined).

Corollary 1.6. *Every common divisor of both* a *and* b *divides* $\gcd(a, b)$.

Corollary 1.7. $\gcd(a, b) = ax + by$ *for some (non-unique)* $x, y \in \mathbb{Z}$.

Proof. We need only address the non-uniqueness of x and y; the rest follows from (the proof of) proposition 1.5. We have, for instance: $\gcd(a, b) = ax + by = ax + by + ab - ab = a(x + b) + b(y - a)$. ■

How does one find these x and y values is the topic of §4.

Proposition 1.8. $\gcd(a, n) = d$ *if and only if* $\gcd(a + kn, n) = d$, *for any* $k \in \mathbb{Z}$.

Proof. The proof follows from the fact that any common divisor of a and n, is also a common divisor of $a + kn$ and n (the verification is straightforward), and vice-versa (which we show explicitly). Let t be a common divisor of $a + kn$ and n, which means there exists integers v and r such that $a + kn = tv$ and $n = tr$. Plugging the latter into the former we have $a + k(tr) = tv \Leftrightarrow a = t(v - kr)$—meaning t is also a divisor of a, and thus a common divisor of a and n. ■

2 Equivalence Classes

Two integers that, divided by the same positive integer n, leave the same remainder can be considered "equivalent," in a sense—which is precisely specified by the following definition.

Definition 2.1 (Equivalence relation). *Given a non-empty set S, a binary relation on S—meaning a non-empty subset R of $S \times S$—is said to be an* **equivalence relation** *if it verifies the following properties:*

- ***Reflexivity:*** $(a, a) \in R$*, for all* $a \in S$.
- ***Symmetry:*** $(a, b) \in R \to (b, a) \in R$.
- ***Transitivity:*** $\big((a, b) \in R \wedge (b, c) \in R\big) \to (a, c) \in R$.

What one tries to capture in this definition, is a generalisation of the notion of equality, from elements (the integers) to sets of elements (the equivalence classes, see below). In the case of the integers, the binary relation we are thinking about is the so-called *congruence* relation.

Definition 2.2 (Integer congruence). *Given a positive integer n—hereinafter called the* **modulus***—and any two integers a, b, they are said to be* **congruent modulo** n *if and only if $n \mid b - a$. In which case we write $a \equiv b \pmod{n}$ or $a \equiv_n b$—or, if the modulus is clear from context, just $a \equiv b$.*

Proposition 2.3. *Integer congruence is an equivalence relation as per definition 2.1.*

Proof. Reflexivity and symmetry are obvious; as for transitivity, note that $a \equiv b \pmod{n}$ if and only if there exists an integer k such that $b = a + kn$. Hence from $a \equiv b$ and $b \equiv c$ we have $b = a + kn$ and $c = b + k'n$—and replacing b in the latter equation gives us $c = (a + kn) + k'n = a + (k + k')n$, which means $a \equiv c$, as was to be shown. ∎

Given an equivalence relation R over a set S, the **equivalence class** of an element a of S is the set $\{b \in S : (a, b) \in R\}$. In the case of integer congruence, the equivalence class (modulo n) of an integer a is the set of all integers b such that $a \equiv b \pmod{n}$—which we shall represent by $[a]_n$, again omitting the modulus information when redundant. The integer a is called the **representative** of the equivalence class (any other integer belonging to the same class can also be used as a representative of that class).

Remark 2.4. It is not particularly useful to have a modulus of 1 (even though it is allowed by definition 2.2): given that the difference between **any** two integers is always a multiple of 1, we just get one (very) big equivalence class, namely all of $\mathbb{Z}$. △

Proposition 2.5. *The equivalence classes induced by an equivalence relation are pairwise disjoint.*

Proof. I will prove this for the specific case of integer congruences, but the more general proof is virtually identical. Suppose $[a]_n$ and $[b]_n$ had a common element, let us say, the integer c. By definition this means that modulo n, both $a \equiv c$ and $b \equiv c$ hold—entailing, via transitivity, that also $a \equiv b$. Now let $a' \in [a]_n \Leftrightarrow a' \equiv a$. Again by transitivity comes $a' \equiv b \Leftrightarrow a' \in [b]_n$, showing that $[a]_n \subseteq [b]_n$. Via a similar reasoning one shows the reverse inclusion, thus establishing that if two equivalence classes are not disjoint, they are equal. ■

Thus each equivalence class of R is a subset of S, and because each element of S belongs to exactly one equivalence class, the set of all equivalence classes constitutes a *partition* of that set. I.e., the equivalence classes are pairwise disjoint, and their union yields the entirety of S. In particular, integer congruence modulo n partitions the integers into n distinct equivalence classes, which we can represent via the integers $0, \ldots, n-1$: $[0], \ldots, [n-1]$. To see this, note that the integers that are congruent to 0, are precisely the multiples of n; those that are congruent to 1 are the integers that can be written as $1 + kn$, for some integer k, and so on. The fact that *any* integer belongs to one of those classes follows from the fact that by theorem 1.2, the remainder of division by n is always one of the integers $0, \ldots, n-1$. These integers are called the **residues modulo** n, and the set of equivalence classes $[0], \ldots, [n-1]$ are the **residue classes modulo** n, which we will denote by $\mathbb{Z}_n$. This latter set can be endowed with both addition and multiplication:[3]

Definition 2.6. *Given a modulus n and integers a, b, we define addition in $\mathbb{Z}_n$ as $[a]_n + [b]_n = [a+b]_n$. Multiplication in $\mathbb{Z}_n$ is defined similarly: $[a]_n [b]_n = [ab]_n$.*

Prima facie, this seems a straightforward enough way of extending the usual arithmetic of $\mathbb{Z}$ to $\mathbb{Z}_n$—the only catch is that we have to show that the result does *not* depend on the particular class representatives (i.e., the particular integers) chosen. This is taken care of by the next result.

Proposition 2.7. *With modulus* n*, suppose that* $a \equiv a'$*, and* $b \equiv b'$*. Then* $[a]_n + [b]_n = [a']_n + [b']_n$ *and* $[a]_n [b]_n = [a']_n [b']_n$.

Proof. By hypothesis we have $a' = a + k_1 n$ and $b' = b + k_2 n$, for some integers k_1 and k_2.

- **Addition.** We need to show that $[a + b]_n$ and $[a' + b']_n$ are the same equivalence class, which is tantamount to showing that $a + b \equiv a' + b'$ (because this establishes that they are not disjoint, and thus are equal by proposition 2.5). We have: $a' + b' = (a + k_1 n) + (b + k_2 n) = (a + b) + n(k_1 + k_2)$, entailing $a + b \equiv a' + b'$.
- **Multiplication.** We need to show that $[ab]_n$ and $[a'b']_n$ are the same equivalence class, which is tantamount to showing that $ab \equiv a'b'$. We have: $a'b' = (a+k_1 n)(b+k_2 n) = ab+ak_2 n+k_1 nb+k_1 nk_2 n = ab+n(ak_2+k_1 b+k_1 nk_2)$, and thus $ab \equiv a'b'$.

And we are done. ■

Excursus. When I first learned this subject, I wondered if the arithmetic of congruence equivalence classes could serve as an indication of how to decompose integers. For example, if $r \in [a + b]_n$, then there always exist integers s, t such that $r = s+t$ and $s \in [a]_n$ and $t \in [b]_n$. For we have $r = a + b + kn$ for some integer k, which we can always rewrite as, for example, $r = (a + 3n) + (b + (k - 3)n)$. However, after some trying I discovered an example with modular multiplication where this does *not* happen: $[3]_5 \times [4]_5 = [12]_5 = [2]_5$—but $7 \in [2]_5$ is a prime, and so cannot be written as a product ab, with $a \in [3]_5$ and $b \in [4]_5$.

The explanation is that the property just described for modular addition… is an extra: something that is not required for said operation to be well-defined. For a binary operation is nothing more than a function (that, in the particular setting of both modular addition and multiplication, has the signature $\mathbb{Z}_n \times \mathbb{Z}_n \mapsto \mathbb{Z}_n$). And so, for it to be well-defined, all that is needed is for any given pair of equivalence classes to uniquely correspond ("map") to a well-defined target equivalence class—a result that follows from definition 2.6 together with proposition 2.7.

3 Linear Congruences

It should be clear to the reader that the "worlds" of $\mathbb{Z}$ and $\mathbb{Z}_n$ are "linked," if you will, by the following property: $a \equiv b \pmod{n}$ if and only if $[a]_n = [b]_n$—and arithmetic in $\mathbb{Z}_n$ can aid us in better understanding some properties of congruences (which "inhabit" the realm of $\mathbb{Z}$). Firstly, if $[a]_n = [b]_n$, then for any integer c, we also have $[a]_n + [c]_n = [b]_n + [c]_n$ and $[a]_n [c]_n = [b]_n [c]_n$—otherwise, the operations would not be well-defined. Or equivalently, we have $[a + c]_n = [b + c]_n$ and $[ac]_n = [bc]_n$. But given any c' such that $c' \equiv_n c$, we can add (or multiply) $[c]_n$ on one hand side, and $[c']_n$ on the other—because they are equal—and similarly conclude that $[a]_n = [b]_n$ also implies $[a + c]_n = [b + c']_n$ and $[ac]_n = [bc']_n$. What does this mean in terms of congruences? That $a \equiv_n b$ implies $a + c \equiv_n b + c'$ and $ac \equiv_n bc'$, where c and c' needn't be equal, but only modularly equivalent. For example, modulo 11 we have $3 \equiv 14$, and if we multiply the left hand side by 13, and the right hand side by 2, we obtain $39 \equiv 28$, which the reader will easily verify to be true. Or for addition, add 24 and 2 to the left and right sides respectively, obtaining $27 \equiv 16$.

Secondly, take any congruence—say, $ab + c \equiv d \pmod{n}$, with a, b, c, d integers. We can rewrite it in terms of equivalence classes: $[ab + c]_n = [d]_n \Leftrightarrow [a]_n [b]_n + [c]_n = [d]_n$. But given integers a', b', c', d' equivalent respectively, modulo n, to a, b, c, d, we can write the same thing as: $[a']_n [b']_n + [c']_n = [d']_n \Leftrightarrow [a'b' + c']_n = [d']_n$—which, translates back to a congruence as $a'b' + c' \equiv d'$. Meaning that given any congruence, we can replace any one of the integers in it with equivalent ones, and preserve the equivalence.[4]

In practice, we usually replace an integer a with its residue modulo n (which, recall, is the remainder of its division by n)—customarily denoted by $a \bmod n$. That both are equivalent is an immediate consequence of the fact that by integer division, there exists an integer q such that $a \bmod n = a - nq$. In fact, we have the following result.

Proposition 3.1. $a \bmod n = b \bmod n$ *if and only if* $a \equiv b \pmod{n}$.

Proof. ($\rightarrow$) $a \bmod n = b \bmod n$ implies $a \bmod n \equiv_n b \bmod n$, which in turn implies $a \equiv b \pmod{n}$, because we have both $a \equiv_n (a \mod n)$, and $b \equiv_n (b \mod n)$.

($\leftarrow$) Via the same reasoning as above, $a \equiv b \pmod{n}$ also implies $(a \bmod n) \equiv_n (b \bmod n)$. This means that $(a \bmod n) - (b \bmod n)$ has to be a multiple of

n—but as both residues belong to the set $\{0, 1, \ldots, n-1\}$, their difference belongs to the set $\{-(n-1), \ldots, -1, 0, 1, \ldots, n-1\}$. The only multiple of n in that set is 0—which means that the equivalence between $a \bmod n$ and $b \bmod n$, is actually equality. ■

Let us now consider a real linear congruence, say, $19x \equiv 9 \pmod 7$, x being an unknown. Observe that in line with out discussion above, x is an integer but the solution to this congruence is not a single integer, but an entire equivalent class: if a is a solution, then so is any integer in $[a]_7$. And as just explained, the first thing we can do is replace all numbers by their residues, obtaining $5x \equiv 2$. If this were an "ordinary equation," we would multiply both sides by the (multiplicative) inverse of 5 (more on this below). And here we will do the same, but with 5's *modular inverse*, i.e., a number such that, when multiplied by 5, yields a result congruent with 1, modulo 7. Said number is 3: indeed, $3 \times 5 = 15 \equiv 1 \pmod 7$. Multiplying both sides we obtain $15x \equiv 6 \Leftrightarrow x \equiv 6$, meaning the solution is any integer in $[6]_7$—as the reader can quickly verify.[5]

Of course, now new questions pop up: does such an inverse always exist? If not, in what circumstances does it exist? How do we find it? We have the following result.

Proposition 3.2. *For any integer a and modulus n, a is invertible modulo n if and only if $\gcd(a, n) = 1$—i.e., if they are relatively prime.*

Proof. We require the properties of the gcd proved in §1. For the forward direction, note that if b is a's modular inverse, this means we have $ab + nk = 1$, for some integer k. But as the gcd is the smallest positive integer that can be written as a linear combination of a and n, it must be the case that $\gcd(a, n) = 1$. The backward direction is easy: if $\gcd(a, n) = 1$, there exist integers x, y such that $ax + ny = 1$—and thus, x is a's modular inverse. ■

This takes care of the first two questions—and by now, we have already (partly) answered the last remaining question, viz. how to compute modular inverses: given co-prime integers a and (the modulus) n, to find the former's inverse we write $\gcd(a, n)$ as a linear combination a and n—and the modular inverse is the weight of a. One way of doing this is via the so-called *extended Euclidean algorithm*, which we describe in the next section.[6]

Before that, though, we can again lean on $\mathbb{Z}_n$ to better grasp the concept of modular inverse. If it were an equation in $\mathbb{Q}$ or $\mathbb{R}$, then as stated above, solving $5x = 2$ would be a simple matter of multiplying both sides by $1/5$, the multiplicative inverse of 5. What this means is that $1/5$ is the number that, when multiplied by 5, yields 1—which is the multiplicative identity of both $\mathbb{Q}$ and $\mathbb{R}$, meaning that the result of multiplying any number by 1, is that number. The same is true in $\mathbb{Z}_n$, and in particular in $\mathbb{Z}_7$: multiplying any element by $[1]_7$, yields that element. And the element by which we must multiply $[5]_7$, to obtain $[1]_7$, is precisely $[3]_7$.

Restating the congruence we wanted to solve in terms of residue classes makes clear that we did (essentially) the same thing one does to solve identical equations in $\mathbb{Q}$ or $\mathbb{R}$. We have $[5]_7\,[x]_7 = [2]_7$, and multiplying both sides by $[3]_7$ yields (we omit the module):[7]

$$[5]\,[x] = [2] \Leftrightarrow [3]\,[5]\,[x] = [3]\,[2] \Leftrightarrow [15]\,[x] = [6] \Leftrightarrow [1]\,[x] = [6]$$

which is simply $[x]_7 = [6]_7$, or in congruential terms, $x \equiv 6 \pmod 7$—precisely the result we got above.

Lastly, if 5 is invertible modulo 7, then any integer in $[5]_7$ is also invertible. Indeed, as $5 \times 3 \equiv_7 1$, then courtesy of the above discussion, we also have that $a \times 3 \equiv_7 1$ holds, for any integer in $a \in [5]_7$. By symmetry, it also follows that any integer in $[3]_7$ is invertible.[8]

The restriction on modular inversion imposed by proposition 3.2 has two consequences:

- First, not all congruences are solvable, because for a congruence $ax \equiv b \pmod n$ to be solvable (x being the unknown), means that b can be written as a linear combination of a and n, i.e., that we have integers x, y such that $b = ax + ny$—which is only possible if $\gcd(a, n) \mid b$. It follows that if $\gcd(a, n) = 1$, the congruence is solvable for any integers b and $a \neq 0$.
  For example, $2x \equiv 5 \pmod 6$, has no solution, for 5 is not a multiple of $2 = \gcd(2, 6)$—otherwise there would exist integers x and k such that $2x + 6k = 5$, entailing that $2 \mid 5$, which is false. However, $2x \equiv 4 \pmod 6$ does have a solution, because 4 is a multiple of $\gcd(2, 6)$. To find it, we do the same that we do to find the modular inverse: write $\gcd(2, 6)$ as a linear combination of both numbers, i.e., $\gcd(2, 6) = 2 = 2 \times (-2) + 6 \times 1$. Multiplying both sides by 2,

we obtain $4 = 2 \times (-4) + 6 \times 2$—meaning the solution is $x \equiv -4 \pmod 6$, or to use the corresponding residue, $x \equiv 2 \pmod 6$.

- Second, it explains why we cannot always "cancel" multiplicative[9] factors: if $[a]_n = [b]_n$, then for any integer c we always have $[c]_n [a]_n = [c]_n [b]_n$—*but the converse is false!* For example again using modulus 6, we have $[2][3] = [2][12]$ (because $6 \equiv 24$), but $[3] \neq [12]$ (as $3 \not\equiv 12$)! This is because 2 has no modular inverse modulo 6, as $\gcd(2, 6) = 2 \neq 1$. And so, there is no equivalence class in $\mathbb{Z}_6$ that when multiplied by $[2]_6$, yields $[1]_6$.[10]

Integers and congruences vs. $\mathbb{Z}_n$. Especially in more advanced algebra texts, it is common to see the set $\mathbb{Z}_n$ defined as $\{0, 1, \ldots, n-1\}$. This is done for convenience: the elements of $\mathbb{Z}_n$ are still the residue classes, but when there is no ambiguity, they are denoted by their canonical representatives. As we saw above, the congruence $5x \equiv 2 \pmod 7$, where $5, 2$ and x represent integers, can be written as $[5]_7 [x]_7 = [2]_7$, omitting the modulus if clear from context. In more advanced books however, this could be written simply as $5x = 2$, with $5, 2, x$ now being elements of $\mathbb{Z}_7$.[11]

4 Euclid's Algorithm(s)

Let a, b be two different positive integers, and assume that $a > b$. The key to understand how to compute the gcd, is the following: due to integer division, we can write $a = bq + r$, and thus conclude that any common divisor of b and r also divides a—but *conversely*, as we also have $r = a - bq$, any common divisor of a and b also divides r. And so, $\gcd(a, b) = \gcd(b, r)$. Now if $r = 0$ we are done—$\gcd(a, b) = b$—but otherwise, we can do integer division of b by r, to obtain $b = rq' + r'$—and observe similarly that $\gcd(b, r) = \gcd(r, r')$. We can continue this process until we obtain a remainder of 0, which must necessarily happen after a finite number of divisions, because the sequence of remainders is strictly decreasing, and lower-bounded by 0. Suppose $r'''' = 0$, meaning the last nonzero remainder was r'''. As we have $r'' = r'''q'''' + r''''$, this means $r''' \mid r''$, and so $\gcd(r'', r''') = r'''$ (note that being nonzero, r''' must be positive). Which means that we also have $\gcd(a, b) = r'''$. This, then, is the "regular" Euclidean algorithm: just compute successive divisions until you get to the last nonzero remainder, which will be the envisaged gcd.

To get from here to the so-called *extended* Euclidean algorithm, the goal of which is to write $\gcd(a, b)$ as a linear combination of a and b, we just work in reverse through the list of remainders, until we get to a and b. (Henceforth we will refer to the *first* and *second* parts of the extended Euclidean algorithm: the first part is the one that is equal to the regular Euclidean algorithm—successive divisions to ascertain the gcd—and the second part is the going back through the list of remainders to obtain the sought out linear combination.) To simplify the notation, let $r_0 := a$ and $r_1 := b$, and suppose we have:

- $r_0 = r_1 q_1 + r_2$
- $r_1 = r_2 q_2 + r_3$
- $r_2 = r_3 q_3 + r_4$

with r_4 being the last nonzero remainder. This means we have $\gcd(a, b) = r_4 = 1 \times r_2 + (-q_3) \times r_3)$. Because we can write r_3 as a function of r_2 and r_1, it should be clear to see that we can eliminate r_3 from that expression, leaving $\gcd(a, b)$ written as a linear combination of r_2 and r_1. And carrying out the same process with r_2 we can eliminate it, and finish with the gcd as a linear combination of r_1 and r_0—which is the desired result.

But to be able to code this algorithm—which is the end goal—we must be both more generic and more explicit. So suppose we are at a point where we have the gcd written as a linear combination of r_i and r_{i+1}, i.e. we have something like $\gcd(a, b) = c_i r_i + c_{i+1} r_{i+1}$. As exemplified in the previous paragraph, the next step is to leverage the fact that we can write r_{i+1} in terms of r_{i-1} and r_i, to get the gcd written as a linear combination of r_{i-1} and r_i. From $r_{i-1} = r_i q_i + r_{i+1}$ comes $r_{i+1} = 1 \times r_{i-1} + (-q_i) r_i$, and we now have:

$$\begin{aligned} \gcd(a, b) &= c_i r_i + c_{i+1} r_{i+1} \\ &= c_i r_i + c_{i+1} \Big(1 \times r_{i-1} + (-q_i) r_i \Big) \\ &= c_{i+1} r_{i-1} + \Big(c_i + q_i (-c_{i+1}) \Big) r_i \end{aligned} \tag{4.1}$$

Observe that in each iteration we always deal with two remainders, one with a lower index and another with a higher one: in (4.1) above, we start with r_i and r_{i+1}, and end with r_{i-1} and r_i. However, to implement Euclid's extended algorithm in code, we don't need the actual reminders—all we need are their respective weights: for as we know how these change from one iteration to the next—this is what (4.1)

describes—we can compute sequentially the weights all the way up to the weights of r_0 and r_1, which values we do know (they are a and b, respectively).

The only thing missing to understand the code below, is to illustrate the "kick-start" of the second phase of the algorithm (i.e., the one that starts after we have the value of the gcd—cf. line 45). So suppose that r_n is the last nonzero remainder (i.e., it is the gcd); we have: $r_{n-2} = r_{n-1}q_{n-1} + r_n \Leftrightarrow r_n = 1 \times r_{n-2} + (-q_{n-1})r_{n-1}$. In the code below, the coefficient for the lower index remainder is called `cl`, and the one for the higher index remaider is `ch`—and hence, it should now be clear that the first iteration yields `cl` $:= 1$ and `ch` $:= (-q_{n-1})$. This is exactly what is done in lines 47 and 48 below. The quotients are stored during the first phase of the algorithm, by appending them to a list (line 40).[12] And are removed, during the second phase, in reverse order of insertion (the so-called LIFO: "Last In, First Out"). To understand the reason for reverse order, note that the first quotient to be used in the second phase of the algorithm, is q_{n-1}—which is the *last* quotient produced during the first phase.[13] And what is the next quotient to be used? Well, q_{n-2}, as we now show. The next step is to write r_{n-1} in terms of r_{n-2} and r_{n-3}, so as to end up with the gcd written as a linear combination of these two latter values. We have $r_{n-3} = r_{n-2}q_{n-2} + r_{n-1} \Leftrightarrow r_{n-1} = 1 \times r_{n-3} + (-q_{n-2})r_{n-2}$. This originates the following transition:

- **Old:** $r_n = 1 \times r_{n-2} + (-q_{n-1})r_{n-1}$
- **New:** $r_n = (-q_{n-1})r_{n-3} + (1 + q_{n-2}q_{n-1})r_{n-2}$

The reader is welcomed to check the math, but we followed the rule outlined above at (4.1): the coefficient of the old higher index remainder, r_{n-1}, becomes the coefficient of the new lower index remainder, r_{n-3}. As for the coefficient of the new higher order remainder r_{n-2}, it is constructed by the addition of old lower order coefficient, 1, to the product of the symmetric of old higher order coefficient $-q_{n-1}$, and the quotient that corresponds to the index of the current remainder $(n-2)$. And it is only this latter quotient, q_{n-2}, that has to be retrieved from the LIFO—cf. line 53. And as it is also the second to last to be added to the LIFO during the first phase, therefrom comes the need for reverse order.

```
1  #! /usr/bin/python
2
3  import sys
4
```

```
5   """
6   Invoke from the command line, with two positive integers as arguments. Prints
7   the linear combination of said integers, that yields their gcd.
8   """
9
10  a = int(sys.argv[1])
11  b = int(sys.argv[2])
12
13  if a == b:
14    print("gcd(%d, %d) = %d = %d * %d + %d * %d" % (a, b, abs(a), a, 1, b, 0))
15    sys.exit(0)
16
17  r0 = None # r_i
18  r1 = None # r_{i + 1}
19  r2 = None # r_{i + 2}
20
21  q = [] # List of quotients.
22
23  if a > b:
24    r0 = a
25    r1 = b
26  else: # a < b
27    r0 = b
28    r1 = a
29
30  r0_orig = r0
31  r1_orig = r1
32
33  r2 = r0 % r1
34
35  if r2 == 0:
36    print("gcd(%d, %d) = %d = %d * %d + %d * %d" % (r0, r1, r1, r0, 0, r1, 1))
37    sys.exit(0)
38
39  while r2 != 0:
40    q.append(r0 // r1)
41    r0 = r1
42    r1 = r2
43    r2 = r0 % r1
44
45  # r1 now contains gcd(a, b).
46
47  cl = 1 # Coefficient for lower index (r0).
48  ch = - q.pop() # Coefficient for higher index (r1).
49
50  while q:
51    cl_orig = cl
52    cl = ch
53    ch = cl_orig + q.pop() * (- ch)
54
55  # Sanity check.
56  if not r1 == r0_orig*cl + r1_orig*ch:
57    print("Error: %d NOT EQUAL TO %d * %d + %d * %d!" % (r1, r0_orig, cl,
```

```
58                                                          r1_orig, ch))
59
60  print("gcd(%d, %d) = %d = %d * %d + %d * %d" % (r0_orig, r1_orig, r1, r0_orig,
61                                                  cl, r1_orig, ch))
```

One final note about the code above: as indicated by the comments of lines 6 and 7, this code only deals with the algorithm proper, and not with other issues one might want to take into account, e.g. input validation, dealing with negative numbers, etc. A fuller version of the same code, that *does* deal with such things (in addition to a testing routine), is available at https://gd7.eu/scholarship/intro-mod-arithmetic.

Notes

1. The well-ordering principle states that any non-empty set of non-negative integers has a smallest element. It is an equivalent formulation to the induction principle, which is part of the axiomatic characterisation of natural numbers (and which naturally also applies to non-negative integers).

2. Zero can only be a common divisor of integers a and b if both of them are zero—a possibility that is disallowed by the proposition's hypothesis.

3. For the algebra-savvy reader, $\mathbb{Z}_n$ together with modular addition and multiplication, forms an algebraic structure known as an *abelian ring* ("abelian" means the operations are commutative). It is usually denoted $(\mathbb{Z}_n, +, \cdot)$.

4. Note, however, that because equality implies equivalence (informally, "$= \rightarrow \equiv$"), we don't have to substitute every integer for a *different* one. For example, $ab' + c' \equiv d$ is also an equivalent statement.

5. Note that $[6]_7$ also contains negative numbers: in particular, it contains -1. And sure enough, $19(-1) - 9 = -28$, which is a multiple of 7.

6. The "regular" Euclidean algorithm just computes $\gcd(a, n)$, but not their weights in the linear combination.

7. We rely here—just as we have relied implicitly above—on the fact that modular multiplication is associative. In fact, both modular addition and multiplication are not only associative, but also commutative. And moreover, multiplication distributes over addition. The simplest way to show all this, is computation in $\mathbb{Z}_n$. We give the proof for distributivity; the other ones are similar. We have: $[a]\big([b] + [c]\big) = [a]\,[b + c] = [a(b + c)] = [ab + ac] = [ab] + [ac] = [a]\,[b] + [a]\,[c]$.

8. Observe that both results also follow directly from proposition 3.2 and proposition 1.8, setting $d = 1$ (and $n = 7$) in the latter.

9. However, we can always cancel *additive* factors, because for any $[a]_n$, we always have $[a]_n + [-a]_n = [0]_n$.

10. In line with what was said above, we could have written $[c]_n\,[a]_n = [c']_n\,[b]_n$, with $c' \equiv_n c$—and the exact same reasoning would apply—for $[c]_n$ and $[c']_n$ are equal.

11. For further discussion of this topic, the reader is referred to Shoup [2008, §2.5], especially p. 29f.

12. q_1 gets appended first, then q_2, and so on.

13. To see this is so, consider the particular iteration of the `while` loop in lines 39–43, that starts with the variables `r0`, `r1` and `r2` already having the following values: `r0 :=` r_{n-2}, `r1 :=` r_{n-1}, `r2 :=` r_n. r_n is nonzero, so the body of the loop gets executed—and the very first task done (line 40), is to append to the end of the quotient list the quotient of the integer division of `r0` by `r1`, i.e. of r_{n-2} by r_{n-1}. Which is precisely q_{n-1}.
Now consider what the rest of the body does: it assigns r_{n-1} to `r0`, and r_n to `r1`, and then proceeds to compute r_{n+1}, which it assigns to `r2`—but which has the value 0. Hence the iteration stops, and in particular, no more quotients get added to quotient list `q`.

References

Shoup, Victor, 2008. *A Computational Introduction to Number Theory and Algebra*. New York: Cambridge University Press, 2nd edition. Electronic version available at https://shoup.net/ntb/.