My cryptography book

John Doe (April 19, 2023)

Contents

Chapter 1

Basic number theory

SUGGESTIONS. For this chapter, state the basic axioms and properties/theorems of $\mathbb{Z}$. Provide proofs. But remember that most of the properties/theorems can be generlized to properties/theorems for rings. It's still a good idea to prove the facts for $\mathbb{Z}$ since $\mathbb{Z}$ is not as abstract as general rings and will prepare you for the general results.

1.1 Axioms of $\mathbb{Z}$

We will assume that $(\mathbb{Z}, +, \cdot, 0, 1)$ satisfies the following axioms.

- PROPERTIES OF $+$:
  - Closure: If $x, y \in \mathbb{Z}$, then $x + y \in \mathbb{Z}$.
  - Associativity: If $x, y, z \in \mathbb{Z}$, then $(x + y) + z = x + (y + z)$.
  - Inverse: If $x \in \mathbb{Z}$, then there is some y such that $x + y = 0 = y + x$. The y in the above is an additive inverse of x.
  - Neutrality: If $x \in \mathbb{Z}$, then $0 + x = x = x + 0$.
  - Commutativity: If $x, y \in \mathbb{Z}$, then $x + y = y + x$.

  (Memory aid for the first four: CAIN.)
- PROPERTIES OF $\cdot$:
  - Closure: If $x, y \in \mathbb{Z}$, then $x \cdot y \in \mathbb{Z}$.
  - Associativity: If $x, y, z \in \mathbb{Z}$, then $(x \cdot y) \cdot z = x \cdot (y \cdot z)$.
  - Neutrality: If $x \in \mathbb{Z}$, then $1 \cdot x = x = x + 1$.
  - Commutativity: If $x, y \in \mathbb{Z}$, then $x \cdot y = y \cdot x$.

  It is common to write xy instead of $x \cdot y$.
- DISTRIBUTIVITY: If $x, y, z \in \mathbb{Z}$, then $x \cdot (y + z) = x \cdot y + x \cdot z$ and $(y + z) \cdot x = y \cdot x + z \cdot x$

A set R with operations $+, \cdot$ and elements $0_R, 1_R$ satisfying the above properties

the above is called a commutative ring. This is an important generalization because there are many very useful commutative rings and we want to prove results about commutative rings so that these results can be applied to all commutative rings, including but not restricted to $\mathbb{Z}$. And if the commutativity of multiplication is left out, then we have the concept of a ring; for emphasize these are called non-commutative rings. This is also an important concept since $n \times n$ matrices with $\mathbb{R}$ entries, $\mathrm{M}_{n\times n}(\mathbb{R})$, form a non-commutative ring. In fact, more generally, the set of $n \times n$ matrices with entries in a commutative ring R, $\mathrm{M}_{n\times n}(R)$, is itself a non-commutative ring. We will return to the concept of commutative and non-commutative rings later.

The next property of $\mathbb{Z}$, integrality, is very special and does not apply to many communtative rings and is therefore left out of the definition of commutative ring:

- Integrality: If $x, y \in \mathbb{Z}$, then $xy = 0 \implies x = 0$ or $y = 0$.

Another property of $\mathbb{Z}$ that we will assume is

- Nontriviality: $0 \neq 1$

This axiom of $\mathbb{Z}$ is extremely simple, but cannot be deduced from the previous axioms.

The above forms the algebraic properties of $\mathbb{Z}$, i.e., properties involving addition and multiplication.

It is actually possible to first define axioms for $\mathbb{N} = \{0, 1, 2, ...\}$ and then define $\mathbb{Z}$ in terms of $\mathbb{N}$. We will not do that except to mention that the axioms for $\mathbb{N}$ are called the Peano-Dedekind axioms and that one very important Peano-Dedekind axiom of $\mathbb{N}$ is the

- Well-ordering principle (WOP) for $\mathbb{N}$: If X is a nonempty subset of $\mathbb{N}$, then X contains a minimum element, i.e., there is some $m \in X$ such that
$$m \leq x$$
for all $x \in X$.

Without going into details, it can be shown that for $\mathbb{N}$, the WOP is equivalent to each of the following axioms:

- Weak Mathematical Induction for $\mathbb{N}$: Let X be a subset of $\mathbb{N}$ satisyfing the following two conditions:

- $0 \in X$ and
- Let $n \in \mathbb{N}$. If $n \in X$, then $n+1 \in X$.

Then $X = \mathbb{N}$.

and

- Strong Mathematical Induction for $\mathbb{N}$: Let X be a subset of $\mathbb{N}$ satisyfing the following two conditions:
  - $0 \in X$ and
  - Let $n \in \mathbb{N}$. If $k \in X$ for all $0 \leq k \leq n$, then $n+1 \in X$.

  Then $X = \mathbb{N}$.

In the above two induction axioms, if we write $X = \{n \mid P(n)\}$ where $P(n)$ is a propositional formula, then the induction axioms can be rewritten in the following way:

- Weak Mathematical Induction: Let $P(n)$ be a proposition for $n \in \mathbb{N}$ satisyfing the following two conditions:
  - $P(0)$ is true and
  - Let $n \in \mathbb{N}$. If $P(n)$ is true, then $P(n+1)$ is true.

  Then $P(n)$ is true for all $n \in \mathbb{N}$.

and

- Strong Mathematical Induction: Let $P(n)$ be a proposition for $n \in \mathbb{N}$ satisyfing the following two conditions:
  - $P(0)$ is true and
  - Let $n \in \mathbb{N}$. If $k \in X$ for all $0 \leq k \leq n$, then $n+1 \in X$.
  - Let $n \in \mathbb{N}$. If $P(k)$ is true for $0 \leq k \leq n$, then $P(n+1)$ is true.

  Then $P(n)$ is true for all $n \in \mathbb{N}$.

The above are the algebraic axioms of $\mathbb{Z}$. There's also the order relation of $\mathbb{Z}$ which is used in WOP and the two induction principles. I will formalize the axioms of the order relation later. For now one can assume that the order relation is defined as follows: If $x \in \mathbb{Z}$, then

$$x < y$$

if there is some $z \in \mathbb{N}$ such that

$$x + z = y$$

There is one more axiom of $\mathbb{Z}$ that is related to the "topology" of $\mathbb{Z}$ and uses the order relation:

- Topology: Given any $x \in \mathbb{Z}$, there is no $y \in \mathbb{Z}$ such that

$$x < y < x + 1$$

You can think of topology of a set as study of "closeness" of values in that set. For $\mathbb{Q}$, given any two distinct rational values $x < y$, there is also some $z \in \mathbb{Q}$ such that $x < z < y$. This is the same for $\mathbb{R}$. Therefore the topology of $\mathbb{Z}$ is very different from the topology of $\mathbb{Q}$ and $\mathbb{R}$ because there are "holes" in $\mathbb{Z}$ where there are no $\mathbb{Z}$ values. $\mathbb{Z}$ has what is called a discrete topology.

The above assume the existence of an order relation on $\mathbb{Z}$, i.e., $<$. We have to include the following axioms of $<$ on $\mathbb{Z}$. There is a set $\mathbb{Z}^+$ such that the following holds:

- Trichotomy: If $x \in \mathbb{Z}$, then exactly one of the following holds: $-x \in \mathbb{Z}^+$, $x = 0$, $x \in \mathbb{Z}^+$.
- Closure of $+$: If $x, y \in \mathbb{Z}^+$, then $x + y \in \mathbb{Z}^+$.
- Closure of $\cdot$: If $x, y \in \mathbb{Z}^+$, then $x \cdot y \in \mathbb{Z}^+$.

We then define $<$ as follows: If $x, y \in \mathbb{Z}$, then we write $x < y$ if

$$y - x \in \mathbb{Z}^+$$

Since $<$ is defined, we can define $x \leq y$ to mean "either $x < y$ or $x = y$". The above order relation is expressed abstractly without refering to the fact that $\mathbb{Z}^+$ is $\{1, 2, 3, ...\}$, i.e., the set of positive integers. In fact, you can prove $\mathbb{Z}^+ = \{1, 2, 3, ...\}$ from the above axioms – see exercises below.

Proposition 1.1.1. *The additive inverse for x is unique. In other words, if y, y' satisfies*

$$x + y = 0 = y + x$$
$$x + y' = 0 = y' + x$$

Then $y = y'$.

Proof. TODO

Since the additive inverse of x is unique, we can choose to write the additive inverse of x in terms of x. This is usually written $-x$. We define the operator $-$ in terms of the additive inverse:

Definition 1.1.1. Let $x, y \in \mathbb{Z}$. We define the subtraction operator as

$$x - y = x + (-y)$$

Note that every x in $\mathbb{Z}$ as an additive inverse, but we did not require value of $\mathbb{Z}$ to have multiplicative inverse:

Definition 1.1.2. Let $x \in \mathbb{Z}$. Then y is a multiplicative inverse of x if

$$x \cdot y = 1 = y \cdot x$$

We say that x is a **unit** if x has a multiplicative inverse. We can also say that x is **invertible**.

unit

invertible

Intuitively, you know that the only values of $\mathbb{Z}$ with multiplicative inverses are 1 and -1.

Proposition 1.1.2. *Let $x \in \mathbb{Z}$. If x is a unit, then the multiplicative inverse of x is unique. In other words if y, y' satisfies*

$$xy = 1 = yx$$
$$xy' = 1 = y'x$$

then $y = y'$.

Definition 1.1.3. The multiplicative inverse of x, it is exists, is is denoted by x^{-1}.

Proposition 1.1.3. *Cancellation law for addition. Let $x, y, z \in \mathbb{Z}$.*

(a) *If $x + z = y + z$, then $x = y$.*
(b) *If $z + x = z + y$, then $x = y$.*

Proposition 1.1.4. *Let $x \in \mathbb{Z}$.*

(a) $0x = 0 = x0$

(b) $-0 = 0$
(c) $x - 0 = x$. *(NOTE CORRECTION)*

Proof. (a) We will first prove $0x = 0$:

$$\begin{aligned} 0x &= (0+0)x && \text{by Neutrality of } + \\ &= 0x + 0x && \text{by Distributivity} \end{aligned} \tag{1}$$

Since $0x \in \mathbb{Z}$ by Closure of $\cdot$, there is exists some $y \in \mathbb{Z}$ which is an additive inverse of $0x$, i.e.,

$$0x + y = 0 = y + 0x \tag{2}$$

From (1),

$$\begin{aligned} y + 0x &= y + (0x + 0x) \\ 0 &= y + (0x + 0x) && \text{by (2)} \\ 0 &= (y + 0x) + 0x && \text{by Associativity of } + \\ 0 &= 0 + 0x && \text{by (2)} \\ 0 &= 0x && \text{by Neutrality of } + \end{aligned}$$

To prove $0 = x0$, from above

$$\begin{aligned} 0 &= 0x \\ &= x0 && \text{by Commutativity of } \cdot \end{aligned}$$

(b) TODO

(c) TODO □

Proposition 1.1.5. *Let $x, y \in \mathbb{Z}$.*

(a) $-(-1) = 1$
(b) $-(-x) = x$
(c) $x(-1) = -x = (-1)x$
(d) $(-1)(-1) = 1$
(e) $(-x)(-y) = xy$

Proposition 1.1.6. *Cancellation law for multiplication. Let $x, y, z \in \mathbb{Z}$.*

(a) *If $xz = yz$ and $z \neq 0$, then $x = y$.*

(b) *If* $zx = zy$ *and* $z \neq 0$*, then* $x = y$*.*

For convenience, I will write $x^2 = xx$ and in general

$$x^n = \begin{cases} 1 & \text{if } n = 0 \\ x^{n-1}x & \text{if } n > 0 \end{cases}$$

If x has a multiplicative inverse, i.e., if x^{-1} exists, then, for $n \geq 0$, I will define

$$x^{-n} = \left(x^{-1}\right)^n$$

Proposition 1.1.7. *Let* $x \in \mathbb{Z}$*. Then* $[n]x = n \cdot x$*.*

Note that nx has two meanings: nx can be the multiplication of n and x and it can also be $x + \cdots + x$ with n number of x. Of course you would expect them to be the same. For now define

$$[n]x = \begin{cases} 0 & \text{if } n = 0 \\ [n-1]x + x & \text{if } n > 0 \end{cases}$$

and if n is negative, we define

$$[n]x = -([-n]x)$$

1.2 Divisibility

Definition 1.2.1. Let $m, n \in \mathbb{Z}$. Then we say that m **divides** n, and we write $m \mid n$, if there is some $x \in \mathbb{Z}$ such that $mx = n$, i.e.,

$$\exists x \in \mathbb{Z} \cdot [mx = n]$$

Proposition 1.2.1. *Let $a \in \mathbb{Z}$.*

(a) *Then $1 \mid a$.*
(b) *Then $a \mid 0$.*
(c) *(Reflexive) $a \mid a$.*
(d) *If $a \mid b$ and $b \mid a$, then $a = \pm b$.*
(e) *(Transitive) If $a \mid b$ and $b \mid c$, then $a \mid c$.*
(f) *If $a \mid b$, then $a \mid bc$.*
(g) *If $a \mid b$, $a \mid c$, then $a \mid b + c$.*
(h) *(Linearity) If $a \mid b$, $a \mid c$, then $a \mid bx + cy$ for $x, y \in \mathbb{Z}$.*

Proof. TODO

1.3 Congruences

Definition 1.3.1. Let $a, b \in \mathbb{Z}$ and $N \in \mathbb{Z}$ with $N > 0$. Then a is congruent to b mod N and we write

$$a \equiv b \pmod{N}$$

if $N \mid a - b$.

Proposition 1.3.1. *Let $a, b, c, a', b' \in \mathbb{Z}$.*

(a) *(Reflexivity)* $a \equiv a \pmod{N}$
(b) *(Symmetry) If* $a \equiv b \pmod{N}$, *then* $b \equiv a \pmod{N}$
(c) *(Transitivity) If* $a \equiv b, b \equiv c \pmod{N}$, *then* $a \equiv c \pmod{N}$
(d) *If* $a \equiv b$, $a' \equiv b' \pmod{N}$, *then* $a + a' \equiv b + b' \pmod{N}$.
(e) *If* $a \equiv b$, $a' \equiv b' \pmod{N}$, *then* $aa' \equiv bb' \pmod{N}$.

Proof. TODO

Proposition 1.3.2. *Let $a, N \in \mathbb{Z}$ with $N > 0$. Let $q, r \in \mathbb{Z}$ such that*

$$a = Nq + r, \quad 0 \leq r < N$$

Then $a \equiv r \pmod{N}$.

Proof. TODO

Exercise 1.3.1. Show that the cancellation law for $\mathbb{Z}$ does not translate to $\mathbb{Z}$ mod N. In other words, find N, a, b, c such that $c \not\equiv 0 \pmod{N}$ and

$$ac \equiv bc \pmod{N}, \quad a \not\equiv b \pmod{N}$$

1.4 Euclidean property

$\mathbb{Z}$ satisfies this very important property:

Euclidean property

Theorem 1.4.1. (**Euclidean property**) *If $a, b \in \mathbb{Z}$ with $b \neq 0$, then there are integers q and r satisfying*

$$a = bq + r, \quad 0 \leq |r| < |b|$$

The above theorem is the version that can be generalized to general rings. Below is the version for $\mathbb{Z}$. The only difference is the $|r|$ is replaced by r:

Euclidean property 2

Theorem 1.4.2. (**Euclidean property 2**) *If $a, b \in \mathbb{Z}$ with $b \neq 0$, then there are integers q and r satisfying*

$$a = bq + r, \quad 0 \leq r < |b|$$

In many cases, one is interested in the case when $a \geq 0$. So this version is the one found in most textbooks:

Euclidean property 3

Theorem 1.4.3. (**Euclidean property 3**) *If $a, b \in \mathbb{Z}$ with $a \geq 0, b > 0$, then there are integers $q \geq 0$ and $r \geq 0$ satisfying*

$$a = bq + r, \quad 0 \leq r < b$$

quotient
remainder

q is called the **quotient** when a is divided by b; r is the **remainder**. q and r are unique (see proposition below). For instance if $a = 25$ and $b = 3$, then

$$25 = 3 \cdot 8 + 1, \quad 0 \leq 1 < 3$$

The computation

$$a, b \to q, r$$

division algorithm

is called a **division algorithm**.

In Python, you can do this:

```
a = 25
b = 8
q, r = divmod(25, 8)
print("%s = %s * %s + %s" % (a, b, q, r))
```

```
[student@localhost ciss451-book-project] python divmod.py
25 = 8 * 3 + 1
```

Algorithmically, when a and b have a huge number of digits and they are represented using arrays of digits, the division algorithm to compute q, r is basically long division you learnt in middle school. At the hardware level, the same division algorithm occurs but the computation is in terms of bits and not digits.

If we peek ahead and pretend for the time being that fractions such as $\frac{a}{b}$ exists, then for $a > 0$ and $b > 0$, we have

$$q = \left\lfloor \frac{a}{b} \right\rfloor, \quad r = a - bq$$

where $\lfloor x \rfloor$ means the floor of x. If we write `(a/b)` for the *integer* quotient of a by b (i.e. this is the / in C++ for integers) and `(a%b)` for the corresponding remainder, then of course we have

```
a = b * (a/b) + (a%b)
```

Although the above Euclidean property is for $\mathbb{Z}$, We will first prove it for $a \geq 0$ and $b > 0$. The q, r will satisfy $q \geq 0$, $r \geq 0$. (Furthermore in this setup q, r are unique.) Once we have proven the Euclidean property for integer $a \geq 0$, it will not be difficult to extend the result to the whole of $\mathbb{Z}$.

To prove the Euclidean property of $\mathbb{Z}$, we will use WOP. (One can also prove the Euclidean property of $\mathbb{Z}$ using induction.)

Well-ordering principle for $\mathbb{N}$

Well-ordering principle for $\mathbb{N}$: Let X be a nonempty subset of $\mathbb{N}$. Then X has a minimal element. In other words there is some $m \in X$ such that $m \leq x$ for all $x \in X$.

You can prove the following version of well-ordering principle on $\mathbb{Z}$:

Well-ordering principle for $\mathbb{Z}$

Well-ordering principle for $\mathbb{Z}$: Let X be a nonempty subset of $\mathbb{Z}$ that is *bounded below*. Then X has a minimal element. In other words there is

some $m \in X$ such that $m \leq x$ for all $x \in X$.

$\mathbb{R}$ does not satisfy the second version well-ordering principle with $\mathbb{Z}$ replaced by $\mathbb{R}$. For instance the open interval $X = (0, 1)$ is bounded below (for instance by -42). However there is no m in X such that $m \leq x$ for all x in X. For instance $m = 0.01 \in X$ is not a minimum element of X since $0.0001 \in X$ is smaller than m. Also, $m = 0.0000001 \in X$ is also not a minimum of X since $0.0000000001 \in X$ is less than m. In fact for any $m \in X$, $(1/2)m$ is in X and is less than m. In other words no value in X can be a minimum value of X.

Now we will prove Theorem 1.4.3.

Proof. TODO

Proposition 1.4.1. *Given a, b, the q, r in Theorem 1.4.3 are unique. In other words, if*

$$a = bq + r, \;\; 0 \leq r < |b|$$
$$a = bq' + r', \;\; 0 \leq r' < |b|$$

then

$$q = q', \quad r = r'$$

Proof. From $bq + r = a = bq' + r'$, we have

$$bq + r = bq' + r'$$

If $q = q'$, then $r = r'$. We now assume $q \neq q'$. Without loss of generality, we'll assume that $q > q'$. We have

$$r' = b(q - q') + r > b + r \geq b$$

which contradicts $r' < b$. □

Now I'm going to prove Theorem 1.4.1 which allows a to be any integer.

Proof of Theorem 1.4.1. Now I'll use Euclidean Property 3 to prove Euclidean Property 1. We just need to handle the case when $a < 0$. Let u be ± 1 so that $ua \geq 0$. Let v be ± 1 so that $vb > 0$. Note that $(\pm 1)^2 = 1$, i.e., $u^{-1} = u, v^{-1} = v$. Using Euclidean Property 3, there exist $q' \geq 0, r'$ such that

$$a' = b'q' + r', \;\; 0 \leq r' < b'$$

Then

$$ua = vbq' + r', \;\; 0 \leq r' < vb = |b|$$

Multiplying by u^{-1}

$$a = uvbq' + ur', \;\; 0 \leq r' < vb = |b|$$

and hence

$$a = b(uvq') + ur', \;\; 0 \leq |ur'| < vb = |b|$$

(Note that $r' \geq 0$ and hence $|ur'| = |u||r'| = r'$.) Hence if $q = uvq'$ and $r = ur'$, then

$$a = bq + r, \;\; 0 \leq |r| < |b|$$

and we are done. □

Exercise 1.4.1. Using the Euclidean property, prove that every integer is congruence to $0, 1, 2,$ or $3 \mod 4$.

Exercise 1.4.2. Prove that squares are 0 or 1 mod 4. In other words if $a \in \mathbb{Z}$, then $a^2 \equiv 0$ or $1 \pmod 4$.

Exercise 1.4.3. Solve $4x^3 + y^2 = 5z^2 + 6$ (in $\mathbb{Z}$).

Exercise 1.4.4. Prove that $11, 111, 1111, 11111, 111111, ...$ are all not perfect squares. (An integer is a perfect square is it's of the form a^2 where a is an integer.)

Exercise 1.4.5. How many of $3, 23, 123, 1123, 11123, 111123, 1111123, ...$ are perfect squares?

1.5 Euclidean algorithm – GCD

Now let me use the Euclidean property to compute the gcd of two integers.

Let's use the division algorithm on 20 and 6.

$$20 = 6 \cdot 3 + 2, \quad 0 \leq 2 < 6$$

Suppose I want to compute $\gcd(20, 6)$. Of course the example is small enough that we know that it is 2. But notice something about this:

$$20 = 6 \cdot 3 + 2, \quad 0 \leq 2 < 6$$

If d is a divisor of 20 and 6, then it must also divide 2. Therefore $\gcd(20, 6)$ must divide 2. The converse might not be true. In general, we have this crucial bridge between Euclidean property and common divisors:

GCD Lemma

Lemma 1.5.1. (**GCD Lemma**) *If $a, b, q, r \in \mathbb{Z}$ such that*

$$a = bq + r$$

then

$$\{d \mid d \text{ is a common divisor of } a, b\} = \{d \mid d \text{ is a common divisor of } b, r\}$$

Hence

$$\gcd(a, b) = \gcd(b, r)$$

Proof. TODO

In particular, given $a, b \in \mathbb{Z}$ where $a > b > 0$. By the Euclidean property of $\mathbb{Z}$, there exist $q, r \in \mathbb{Z}$ such that

$$a = bq + r, \;\; 0 \leq r < b$$

Hence

$$\gcd(a, b) = \gcd(b, r)$$

Note that in the above, I only require $a = bq + r$. For instance for to $\gcd(120, 15)$, I can use $120 = 1 \cdot 15 + (120 - 15)$, i.e., $a = 120, b = 15, q = 1, r = 120 - 15$. Then $\gcd(120, 15) = \gcd(15, 120 - 15) = \gcd(15, 105)$.

However if I use the division algorithm, then r is "small":

$$0 \leq r < b$$

So if you want to compute $\gcd(a, b)$, make sure $a \geq b$ (otherwise swap them). Then $\gcd(a, b) = \gcd(b, r)$ and you would have $a \geq b > r$. So instead of computing $\gcd(a, b)$, you are better off computing $\gcd(b, r)$.

But like I said, we do not need the q and r to be the quotient and remainder. For instance suppose I want to compute the GCD of 514 and 24.

$$514 = 24 \cdot 1 + (514 - 24)$$

Then

$$\gcd(514, 24) = \gcd(24, 514 - 24)$$

which gives us

$$\gcd(514, 24) = \gcd(24, 490)$$

Note that $\gcd(0, n) = n$ for any positive integer n. I'll let you think about that one. (Remember what I said before: 0 is in some sense a big number, like a black hole. Because every positive number divides 0.)

Of course this gives rise to the following algorithm

```
ALGORITHM: GCD
Inputs: a, b
Output: gcd(a, b)

if b > a:
    swap a, b

if b == 0:
    return a
else:
    return GCD(a - b, b)
```

This only subtracts one copy of b from a. Suppose we can compute

$$a = bq + r, \quad 0 \leq r < b$$

Then

$$\gcd(a, b) = \gcd(b, r)$$

Of course r is the remainder when a is divided by b. Using this we rewrite the above code to get the **Euclidean Algorithm**:

Euclidean Algorithm

```
ALGORITHM: GCD (Euclidean algorithm)
Inputs: a, b
Output: gcd(a, b)

if b > a:
    # To make sure that for gcd(a,b), a >= b
    swap a, b

if b == 0:
    return a
else:
    return GCD(b, a % b)
```

Note that if `a` $<$ `b`, then

$$\texttt{GCD(a, b) = GCD(b, a \% b) = GCD(b, a)}$$

Therefore the swap is not necessary:

```
ALGORITHM: GCD (Euclidean algorithm)
Inputs: a, b
Output: gcd(a, b)

if b == 0:
    return a
else:
    return GCD(b, a % b)
```

In this case, I'm assuming that `a % b` is available. As an example:

$$\begin{aligned}\gcd(514, 24) &= \gcd(24, 514\%24) = \gcd(24, 10)\\ &= \gcd(10, 24\%10) = \gcd(10, 4)\\ &= \gcd(10, 10\%4) = \gcd(10, 2)\\ &= \gcd(2, 10\%2) = \gcd(2, 0)\\ &= 2\end{aligned}$$

The above can also be done in a loop:

```
ALGORITHM: GCD (Euclidean algorithm)
Inputs: a, b
Output: gcd(a, b)

while 1:
    if b == 0:
```

```
        return a
    else:
        a, b = b, a % b
```

Exercise 1.5.1. Compute the following using the Euclidean Algorithm explicitly.

(a) $\gcd(10, 1)$
(b) $\gcd(10, 10)$
(c) $\gcd(107, 5)$
(d) $\gcd(107, 26)$
(e) $\gcd(84, 333)$

Exercise 1.5.2. Compute the following. You should go as far as you can. In other words, either you can a fixed integer (such as 1) or derive the $\gcd(\alpha, \beta)$ where α, β are as simple as possible. For instance, to simplify $\gcd(3 + 2a, a)$, since $3 + 2a = 2 \cdot a + 3$, we have

$$\gcd(3 + 2a, a) = \gcd(a, 3)$$

In the following $a, b, x, n \in \mathbb{Z}$ are positive integers.

(a) $\gcd(ab, b)$
(b) $\gcd(a, a + 1)$
(c) $\gcd(ab + a, b)$ where $0 < a < b$
(d) $\gcd(a(a + 1) + a, (a + 1))$ where $0 < a < b$
(e) $\gcd(1 + x + \cdots + x^n, x)$
(f) $\gcd(F_{10}, F_{11})$ where F_n is the n–th Fibonacci number. (Recall: $F_0 = 1, F_1 = 1, F_{n+2} = F_{n+1} + F_n$ for $n \geq 0$.)

Despite the fact that the Euclidean algorithm is one of the fastest algorithm to compute the GCD of two numbers and has been discovered by Euclid a long time ago (BC 300), the actual runtime was not known until Lamé proved in 1844 that the number of steps to compute $\gcd(a, b)$ using the Euclidean algorithm is ≤ 5 times the number of digits (in base 10 notation) of $\min(a, b)$. For instance for the example above of $\gcd(514, 24)$, the number of digits of $\min(514, 24)$ is 2. Lamé theorem says that the number of steps made by the Euclidean algorithm in the computation of $\gcd(514, 24)$ is at most $5 \times 2 = 10$.

The actual number of steps in the earlier computation

$$\begin{aligned}\gcd(514, 24) &= \gcd(24, 514\%24) = \gcd(24, 10)\\ &= \gcd(10, 24\%10) = \gcd(10, 4)\\ &= \gcd(10, 10\%4) = \gcd(10, 2)\\ &= \gcd(2, 10\%2) = \gcd(2, 0)\\ &= 2\end{aligned}$$

is 4 (not counting the base case step), i.e.,

$$\gcd(514, 24) = \gcd(24, 10) = \gcd(10, 4) = \gcd(10, 2) = \gcd(2, 0)$$

Lamé's work is generally considered the beginning of computational complexity theory, which is the study of resources needed (time or space) to execute an algorithm. Another fascinating fact about Lamé's theorem is that historically the above proof is the first ever "use" of the Fibonacci sequence.

Theorem 1.5.1. (Lamé 1844) *Let $a > b > 0$ be integers. If the GCD computation of a, b using Euclidean algorithm results in n steps:*

$$\gcd(a_{n+1}, b_{n+1}) = \gcd(a_n, b_n) = \cdots = \gcd(a_1, b_1), \quad b_1 = 0$$

where $(a_{n+1}, b_{n+1}) = (a, b)$, and $a_i > b_i$, then

(a) *$a \geq F_{n+2}$ and $b \geq F_{n+1}$, where F_n are the Fibonacci numbers ($F_1 = 1, F_2 = 1, F_3 = 2, F_4 = 3, F_5 = 5$, etc. Note that the index starts with 1.)*
(b) *n is at most 5 times the number of digits in b.*

Proof. TODO

Proposition 1.5.1. *The number of digits of a positive integer b is*

$$\lfloor \log_{10} b + 1 \rfloor$$

Proof. TODO

On analyzing the proof, the above is in fact true for any base > 1. In other words the number of base–B symbols to represent b is

$$\lfloor \log_B b + 1 \rfloor$$

where $B > 1$. For instance the number of bits needed to represent b is

$$\lfloor \log_2 b + 1 \rfloor$$

For instance, $b = 9_{10} = 1001_2$ which has 4 bits and

$$\lfloor \log_2 9 + 1 \rfloor = \lfloor 3.1699... + 1 \rfloor = \lfloor 4.1699... \rfloor = 4$$

Proposition 1.5.2. *Let positive integer b be written in base B where $B > 1$ is an integer. Then the number of base-B symbols used to represent b is*

$$\lfloor \log_B b + 1 \rfloor$$

Exercise 1.5.3. Leetcode 650.
https://leetcode.com/problems/2-keys-keyboard/
There is only one character `'A'` on the screen of a notepad. You can perform one of two operations on this notepad for each step:

- Copy All: You can copy all the characters present on the screen (a partial copy is not allowed).
- Paste: You can paste the characters which are copied last time.

Given an integer `n`, return the minimum number of operations to get the character `'A'` exactly `n` times on the screen.

Exercise 1.5.4. Leetcode 2447.
https://leetcode.com/problems/number-of-subarrays-with-gcd-equal-to-k
Given an integer array `nums` and an integer `k`, return the number of subarrays of `nums` where the greatest common divisor of the subarray's elements is `k`. A subarray is a contiguous non-empty sequence of elements within an array. The greatest common divisor of an array is the largest integer that evenly divides all the array elements.

Exercise 1.5.5. Leetcode 1998
https://leetcode.com/problems/gcd-sort-of-an-array/
You are given an integer array `nums`, and you can perform the following operation any number of times on `nums`:

- Swap the positions of two elements `nums[i]` and `nums[j]` if `gcd(nums[i], nums[j]) > 1`

where `gcd(nums[i], nums[j])` is the greatest common divisor of `nums[i]` and `nums[j]`.

Return true if it is possible to sort `nums` in non-decreasing order using the above swap method, or false otherwise.

1.6 Extended Euclidean algorithm: GCD as linear combination

Here's another super important fact:

Extended Euclidean Algorithm

Theorem 1.6.1. (**Extended Euclidean Algorithm**) *If a and b be integers which are not both zero, then there are integers x, y such that*

$$\gcd(a, b) = ax + by$$

Bézout's coefficients

The x, y in the above theorem are called **Bézout's coefficients** of a, b. They are not unique.

Exercise 1.6.1. Prove that $a \neq 0$, then there are many possible x, y such that $ax + by = \gcd(a, b)$. □

First let me prove that there are x, y such that

$$ax + by = \gcd(a, b)$$

The theorem does not give you the algorithm. Then I'll do a computational example that compute the gcd of a, b as a linear combination of a and b. The example actually contains the idea behind the algorithm to compute the Bézout's coefficients. The algorithm is called the Extended Euclidean Algorithm.

Proof. For convenience, let me write (a, b) as the set of linear combinations of a and b, i.e.,

$$(a, b) = \{ax + by \mid x, y \in \mathbb{Z}\}$$

We will also write (g) for the linear combination of g, i.e.,

$$(g) = \{gx \mid x \in \mathbb{Z}\}$$

(Such linear combinations are called ideals. They are extremely important in of themselves. Historically, they were created to study Fermat's last theorem. Since then they are crucial in the the study of ring theory.)

The proof proceeds in two steps:

1. Given a, b not both zero, there is some g such that $(a, b) = (g)$. If a, b are not both zero, g can be chosen to be > 0.
2. The g in the above is in fact $\gcd(a, b)$.

Now let's prove step 1, i.e., given a, b, there is some g such that

$$(a, b) = (g)$$

First of all if $b = 0$, then by definition

$$(a, 0) = (a)$$

and we're done. Next, we now assume $b \neq 0$. Then $|b| > 0$. The set

$$X = \{ax + by \mid x, y \in \mathbb{Z} \text{ and } ax + by > 0\} \subseteq \mathbb{N}$$

is nonempty since it contains $0 \cdot a + 1 \cdot |b|$. By the well-ordering principle of $\mathbb{N}$, X has a minimum element, say g. We will now show that $(a, b) = (g)$.

Since g is a minimum element of X, g is in X. Therefore $g = ax + by$. Hence $gz = a(xz) + b(yz) \in (a, b)$ for all $z \in \mathbb{Z}$. This implies that $(g) \subseteq (a, b)$.

Now we will prove that $(a, b) \subseteq (g)$. Let $c \in (a, b)$, i.e., $c = ax + by$ for some $x, y \in \mathbb{Z}$. Therefore by the Euclidean property of $\mathbb{Z}$, there exists $q, r \in \mathbb{Z}$ such that

$$c = gq + r, \;\; 0 \leq r < g$$

(Look at the definition of X again. X is a subset of $\mathbb{N}$ so that $g \geq 1$). If $r \neq 0$, then

$$r = c - gq$$

Note that $c = ax + by$ by our assumption. We have already shown that $(g) \subseteq (a, b)$, i.e., $g = ax' + by'$. Therefore, altogether we have

$$r = c - gq = ax + by - (ax' + b'y)q = a(x - x'q) + b(y - y'q)$$

Hence $r \in X$. But $0 \leq r < d$ implies that

$$r = a(x - x'q) + b(y - b'q)$$

is an element of X which is less than g which contradicts the minimality of g. Hence $r = 0$ and we have

$$c = gq + r = gq \in (g)$$

We have shown that $(a, b) \subseteq (g)$.

Altogether, we have shown $(a, b) = (g)$. Step 1 is now completed.

For step 2, we will show that g is the gcd of a and b. Since $(a, b) = (g)$, we have

$$a \in (a, b) = (g)$$

i.e., $a = xg$ which means g divides a. Likewise g divides b. Hence g is a common divisor of a and b. Since $(g) \subseteq (a, b)$, $g = ax_0 + by_0$. Suppose d is any divisor of a and b. Then $d \mid ax_0 + by_0$ by the linearity of divisibility. Hence $d \mid g$. Therefore g is the largest common divisor of a and b, i.e., $g = \gcd(a, b)$. □

The above does not give you an algorithm to compute the x and y. First let me do an example to show you that it's possible to compute $\gcd(a, b)$ as a linear combination of a and b. Then I'll give you the algorithm.

Recall that we have computed $\gcd(514, 24) = 2$. Extended Euclidean Algorithm says that it's possible to find x and y such that

$$2 = \gcd(514, 24) = 514x + 24y$$

How do we compute the x and y? Just like the gcd computation (the Euclidean Algorithm), the x, y are computed using the Euclidean property. First we have

$$514 = 21 \cdot 24 + 10$$

This implies that

$$514 \cdot 1 + 24 \cdot (-21) = 10$$

Now it would be nice if the pesky 10 goes away and is replaced by 2. How would we do that? Well look at 24 and 10 now. We have

$$24 = 2 \cdot 10 + 4$$

again by Euclidean algorithm. Multiplying the equation

$$514 \cdot 1 + 24 \cdot (-21) = 10$$

throughout by 2 gives us

$$514 \cdot 2 + 24 \cdot (-42) = 2 \cdot 10$$

The previous equation

$$24 = 2 \cdot 10 + 4$$

say that $2 \cdot 10$ can be replaced by $24 - 4$. This means that

$$514 \cdot 2 + 24 \cdot (-42) = 24 - 4$$

Hmmm ... this says that we have now

$$514 \cdot 2 + 24 \cdot (-43) = -4$$

or

$$514 \cdot (-2) + 24 \cdot 43 = 4$$

What about 4? Well, if we look at 10 and 4 just like what we did with 24 and 10 we would get

$$10 = 2 \cdot 4 + 2$$

and the remainder 2 gives us the GCD!!! Rearranging it a bit we have

$$1 \cdot 10 + (-2) \cdot 4 = 2$$

i.e. 2 is a linear combination of 10 and 4. But earlier we say that 4 is a linear combination of 514 and 24 ...

$$514 \cdot (-2) + 24 \cdot 43 = 4$$

and even earlier we saw that 10 is also a linear combination of 514 and 24 ...

$$514 \cdot 1 + 24 \cdot (-21) = 10$$

Surely if we substitute all these values into the equation

$$1 \cdot 10 + (-2) \cdot 4 = 2$$

we would get 2 as a linear combination of 514 and 24. Let's do it ...

$$\begin{aligned} 2 &= 1 \cdot 10 + (-2) \cdot 4 \\ &= 1 \cdot (514 \cdot 1 + 24 \cdot (-21)) + (-2)(514 \cdot (-2) + 24 \cdot 43) \\ &= 514 \cdot 1 + 24 \cdot (-21) + 514 \cdot 4 + 24 \cdot (-86) \\ &= 514 \cdot 5 + 24 \cdot (-107) \end{aligned}$$

Vóila!

Exercise 1.6.2. Using the above idea, compute the gcd and Bézout's coefficients of 210 and 78, i.e., compute x and y such that $210x+78y = \gcd(210, 78)$.

Exercise 1.6.3. Analyze the above and design an algorithm so that when given a and b, the algorithm computes x and y such that $ax + by = \gcd(a, b)$.

To help you analyze the above computation, let me organize our computations a little. If we can make the process systematic, then there is hope that we can make the idea work for all a and b, i.e., then we would have an algorithm and hence can program it and compute it's runtime performance.

We know for sure that we have to continually use Euclidean property on pairs of numbers. So here we go:

$$\begin{aligned} 514 &= 21 \cdot 24 + 10 \\ 24 &= 2 \cdot 10 + 4 \\ 10 &= 2 \cdot 4 + 2 \\ 4 &= 2 \cdot 2 + 0 \end{aligned}$$

Note that this corresponds to the gcd computation

$$\begin{aligned} \gcd(514, 24) &= \gcd(24, 514 - 21 \cdot 24) = \gcd(24, 10) \\ &= \gcd(10, 24 - 2 \cdot 10) = \gcd(10, 4) \\ &= \gcd(4, 10 - 2 \cdot 4) = \gcd(4, 2) \\ &= \gcd(2, 4 - 2 \cdot 2) = \gcd(2, 0) \\ &= 2 \end{aligned}$$

So in the computation

$$\begin{aligned} 514 &= 21 \cdot 24 + 10 \\ 24 &= 2 \cdot 10 + 4 \\ 10 &= 2 \cdot 4 + 2 \\ 4 &= 2 \cdot 2 + 0 \end{aligned}$$

if the remainder is 0 (see the last line), then the previous line's remainder must be the gcd.

Let's look at our computation of the gcd of 514 and 24:

$$\begin{aligned} 514 &= 21 \cdot 24 + 10 \\ 24 &= 2 \cdot 10 + 4 \\ 10 &= 2 \cdot 4 + 2 \\ 4 &= 2 \cdot 2 + 0 \end{aligned}$$

Recall that the above computation means that the gcd is 2. Note only that through backward substitution, we can rewrite 2 as a linear combination of 514 and 24.

Let's try to do this in a more organized way. So here's our facts again:

$$\begin{aligned} 514 &= 21 \cdot 24 + 10 \\ 24 &= 2 \cdot 10 + 4 \\ 10 &= 2 \cdot 4 + 2 \end{aligned}$$

Let me put the remainders on one side:

$$\begin{aligned} 10 &= 514 - 21 \cdot 24 && (1) \\ 4 &= 24 - 2 \cdot 10 && (2) \\ 2 &= 10 - 2 \cdot 4 && (3) \end{aligned}$$

Note that (1) tells you that 10 is a linear combination of $514, 24$. (2) tells you that 4 is a linear combination of $24, 10$. If we substitute (1) into (2), 4 will become a linear combination of $514, 24$. (3) says that 2 is a linear combination of $10, 4$. But 10 is a linear combination of $514, 24$ and 4 is a linear combination of $514, 24$. Hence 2 is also a linear combination of $514, 24$. See it?

OK. Let's do it. From

$$\begin{aligned} 10 &= 514 - 21 \cdot 24 && (1) \\ 4 &= 24 - 2 \cdot 10 && (2) \\ 2 &= 10 - 2 \cdot 4 && (3) \end{aligned}$$

if we substitute (1) into (2) and (3) (i.e., rewrite 10 as a linear combination of $514, 24$):

$$\begin{aligned} 10 &= 514 - 21 \cdot 24 && (1) \\ 4 &= 24 - 2 \cdot (514 - 21 \cdot 24) && (2) \\ 2 &= (514 - 21 \cdot 24) - 2 \cdot 4 && (3) \end{aligned}$$

Collecting the multiples of 514 and 24:

$$\begin{aligned} 10 &= 514 - 21 \cdot 24 && (1) \\ 4 &= (-2) \cdot 514 + (1 + (-2)(-21)) \cdot 24 && (2') \\ 2 &= (1) \cdot 514 + (-21) \cdot 24 - 2 \cdot 4 && (3') \end{aligned}$$

and simplifying:

$$\begin{aligned} 10 &= 514 - 21 \cdot 24 && (1)\\ 4 &= (-2) \cdot 514 + (43) \cdot 24 && (2')\\ 2 &= (1) \cdot 514 + (-21) \cdot 24 - 2 \cdot 4 && (3') \end{aligned}$$

Substituting $(2')$ into $(3')$:

$$\begin{aligned} 10 &= 514 - 21 \cdot 24 && (1)\\ 4 &= (-2) \cdot 514 + (43) \cdot 24 && (2')\\ 2 &= (1) \cdot 514 + (-21) \cdot 24 - 2 \cdot ((-2) \cdot 514 + (43) \cdot 24) && (3') \end{aligned}$$

Tidying up:

$$\begin{aligned} 10 &= 514 - 21 \cdot 24 && (1)\\ 4 &= (-2) \cdot 514 + (43) \cdot 24 && (2')\\ 2 &= (1 - 2(-2)) \cdot 514 + (-21 - 2(43)) \cdot 24 && (3'') \end{aligned}$$

Simplifying:

$$\begin{aligned} 10 &= 514 - 21 \cdot 24 && (1)\\ 4 &= (-2) \cdot 514 + (43) \cdot 24 && (2')\\ 2 &= (5) \cdot 514 + (-107) \cdot 24 && (3'') \end{aligned}$$

(It's a good idea to check after each substitution that the equalities still hold. We all make mistakes, right?)

OK. That's great. It looks more organized now. So much so that you can now easily write a program to compute the above.

Now let's look at the general case. Suppose instead of 514 and 24, we write a and b. The computation will look like this:

$$\begin{aligned} a &= q_1 \cdot b + r_1\\ b &= q_2 \cdot r_1 + r_2\\ r_1 &= q_3 \cdot r_2 + r_3\\ r_2 &= q_4 \cdot r_3 + 0 \end{aligned}$$

To make things even more regular and uniform, let me rewrite it this way:

$$\begin{aligned} r_0 &= q_1 \cdot r_1 + r_2 \\ r_1 &= q_2 \cdot r_2 + r_3 \\ r_2 &= q_3 \cdot r_3 + r_4 \\ r_3 &= q_4 \cdot r_4 + 0 \end{aligned}$$

A lot nicer, right? Let me write it this way with the remainder term on the lefts:

$$\begin{aligned} r_2 &= (1) \cdot r_0 + (-q_1) \cdot r_1 \\ r_3 &= (1) \cdot r_1 + (-q_2) \cdot r_2 \\ r_4 &= (1) \cdot r_2 + (-q_3) \cdot r_3 \end{aligned}$$

(Remember that r_4 is the gcd ... $r_0 = 514, r_1 = 24$ ... right?) Organized this way, we have the gcd on one side of the equation. Now if we substitute the first equation into the second we get

$$\begin{aligned} r_2 &= (1) \cdot r_0 + (-q_1) \cdot r_1 \text{ ... USED} \\ r_3 &= (1) \cdot r_1 + (-q_2) \cdot ((1) \cdot r_0 + (-q_1) \cdot r_1) \\ r_4 &= (1) \cdot r_2 + (-q_3) \cdot r_3 \end{aligned}$$

i.e.,

$$\begin{aligned} r_2 &= (1) \cdot r_0 + (-q_1) \cdot r_1 \text{ ... USED} \\ r_3 &= (-q_2) \cdot r_0 + (1 + q_1 q_2) \cdot r_1 \\ r_4 &= (1) \cdot r_2 + (-q_3) \cdot r_3 \end{aligned}$$

Note that we cannot throw away the first equation yet. We need to keep r_2 around since it appears in the third equation! So when can we throw the first equation away? Look at the general case. Suppose we have

$$\begin{aligned} r_2 &= (1) \cdot r_0 + (-q_1) \cdot r_1 \\ r_3 &= (1) \cdot r_1 + (-q_2) \cdot r_2 \\ r_4 &= (1) \cdot r_2 + (-q_3) \cdot r_3 \\ r_5 &= (1) \cdot r_3 + (-q_4) \cdot r_4 \\ r_6 &= (1) \cdot r_4 + (-q_5) \cdot r_5 \\ &\ldots \end{aligned}$$

Aha! r_2 is used only in the next *two* equations.

Suppose we are at equation 3:

$$r_3 = c_1 \cdot r_0 + d_1 \cdot r_1$$
$$r_4 = c_2 \cdot r_0 + d_2 \cdot r_1$$

We have to compute the next equation: This requires r_3, r_4. Then we have

$$r_5 = (1) \cdot r_3 + (-q_4) \cdot r_4$$

where

$$q_4 = \lfloor r_3/r_4 \rfloor, \quad r_5 = r_3 - q_4 r_4$$

Altogether we have

$$r_3 = c_1 \cdot r_0 + d_1 \cdot r_1$$
$$r_4 = c_2 \cdot r_0 + d_2 \cdot r_1$$
$$r_5 = (1) \cdot r_3 + (-q_4) \cdot r_4$$

The last equation becomes

$$r_5 = c_1 \cdot r_0 + d_1 \cdot r_1 + (-q_4) \cdot (c_2 \cdot r_0 + d_2 \cdot r_1)$$

i.e.

$$r_5 = (c_1 - q_4 c_2) \cdot r_0 + (d_1 - q_4 d_2) \cdot r_1$$

Let me repeat that in a slightly more general context. If we have

$$r_3 = c_1 \cdot r_0 + d_1 \cdot r_1$$
$$r_4 = c_2 \cdot r_0 + d_2 \cdot r_1$$

then we get (throwing away the first equation):

$$r_4 = c_2 \cdot r_0 + d_2 \cdot r_1$$
$$r_5 = (c_1 - q_4 c_2) \cdot r_0 + (d_1 - q_4 d_2) \cdot r_1$$

To put it in terms of numbers instead of equations this is what we get: If we have

$$c_1, d_1, c_2, d_2, r_3, r_4$$

then we get

$$c_2, d_2, c_1 - \lfloor r_3/r_4 \rfloor c_2, d_1 - \lfloor r_3/r_4 \rfloor d_2, r_4, r_3 - \lfloor r_3/r_4 \rfloor r_4$$

In general, if we have

$$c, d, c', d', r, r'$$

then we get

$$c', d', c - \lfloor r/r' \rfloor\, c', d - \lfloor r/r' \rfloor\, d', r', r - \lfloor r/r' \rfloor\, r'$$

Of course since we start off with r_0, r_1 (i.e. what we call a and b above), we have

$$\begin{aligned} r_0 &= 1 \cdot r_0 + 0 \cdot r_1 \\ r_1 &= 0 \cdot r_0 + 1 \cdot r_1 \end{aligned}$$

i.e., you would start off with

$$c = 1, d = 0, c' = 0, d' = 1, r = r_0, r' = r_1$$

Let's check this algorithm with our $r_0 = 514, r_1 = 24$.

STEP 1: The initial numbers are

$$c = 1, d = 0, c' = 0, d' = 1, r = 514, r' = 24$$

Again this corresponds to

$$\begin{aligned} r_3 &= 1 \cdot 514 + 0 \cdot 24 \\ r_4 &= 0 \cdot 514 + 1 \cdot 24 \end{aligned}$$

STEP 2: The new numbers (6 of them) are

$$\begin{aligned} c' &= 0 \\ d' &= 1 \\ c - \lfloor r/r' \rfloor\, c' &= 1 - \lfloor 514/24 \rfloor\, 0 = 1 \\ d - \lfloor r/r' \rfloor\, d' &= 0 - \lfloor 514/24 \rfloor\, 1 = 0 - 21 = -21 \\ r' &= 24 \\ r - \lfloor r/r' \rfloor\, r' &= 514 - \lfloor 514/24 \rfloor\, 24 = 514 - 504 = 10 \end{aligned}$$

So the new numbers (we reset the variables in the algorithm):

$$c = 0, d = 1, c' = 1, d' = -21, r = 24, r' = 10$$

These corresponds to the data on the second and third line of the following:

$$\begin{aligned} 514 &= 1 \cdot 514 + 0 \cdot 24 \\ 24 &= 0 \cdot 514 + 1 \cdot 24 \\ 10 &= 1 \cdot 514 + (-21) \cdot 24 \end{aligned}$$

STEP 3: From the 6 numbers from STEP 2 we get

$$\begin{aligned} c' &= 1 \\ d' &= -21 \\ c - \lfloor r/r' \rfloor\, c' &= 0 - \lfloor 24/10 \rfloor\, 1 = -2 \\ d - \lfloor r/r' \rfloor\, d' &= 1 - \lfloor 24/10 \rfloor\, (-21) = 1 + 42 = 43 \\ r' &= 10 \\ r - \lfloor r/r' \rfloor\, r' &= 24 - \lfloor 24/10 \rfloor\, 10 = 24 - 20 = 4 \end{aligned}$$

So the new numbers (we reset the variables in the algorithm):

$$c = 1, d = -21, c' = -2, d' = 43, r = 10, r' = 4$$

These corresponds to the data on the third and fourth line of the following:

$$\begin{aligned} 514 &= 1 \cdot 514 + 0 \cdot 24 \\ 24 &= 0 \cdot 514 + 1 \cdot 24 \\ 10 &= 1 \cdot 514 + (-21) \cdot 24 \\ 4 &= (-2) \cdot 514 + 43 \cdot 24 \end{aligned}$$

STEP 4: From the 6 numbers from STEP 3 we get

$$\begin{aligned} c' &= -2 \\ d' &= 43 \\ c - \lfloor r/r' \rfloor\, c' &= 1 - \lfloor 10/4 \rfloor\, (-2) = 1 + 4 = 5 \\ d - \lfloor r/r' \rfloor\, d' &= -21 - \lfloor 10/4 \rfloor\, (43) = -21 - 86 = -107 \\ r' &= 4 \\ r - \lfloor r/r' \rfloor\, r' &= 10 - \lfloor 10/4 \rfloor\, 4 = 10 - 8 = 2 \end{aligned}$$

So the new numbers (we reset the variables in the algorithm):

$$c = -2, d = 43, c' = 5, d' = -107, r = 4, r' = 2$$

These corresponds to the data on the fourth and fifth line of the following:

$$\begin{aligned} 514 &= 1 \cdot 514 + 0 \cdot 24 \\ 24 &= 0 \cdot 514 + 1 \cdot 24 \\ 10 &= 1 \cdot 514 + (-21) \cdot 24 \\ 4 &= (-2) \cdot 514 + 43 \cdot 24 \\ 2 &= 5 \cdot 514 + (-107) \cdot 24 \end{aligned}$$

STEP 5: From the 6 numbers from STEP 4 we get

$$\begin{aligned} c' &= 5 \\ d' &= -107 \\ c - \lfloor r/r' \rfloor\, c' &= -2 - \lfloor 4/2 \rfloor\, 5 = -2 - 10 = -12 \\ d - \lfloor r/r' \rfloor\, d' &= 43 - \lfloor 4/2 \rfloor\, (-107) = 43 + 214 = 257 \\ r' &= 2 \\ r - \lfloor r/r' \rfloor\, r' &= 4 - \lfloor 4/2 \rfloor\, 2 = 4 - 4 = 0 \end{aligned}$$

So the new numbers (we reset the variables in the algorithm):

$$c = 5, d = -107, c' = -12, d' = 257, r = 2, r' = 0$$

These corresponds to the data on the fifth and sixth line of the following:

$$\begin{aligned} 514 &= 1 \cdot 514 + 0 \cdot 24 \\ 24 &= 0 \cdot 514 + 1 \cdot 24 \\ 10 &= 1 \cdot 514 + (-21) \cdot 24 \\ 4 &= (-2) \cdot 514 + 43 \cdot 24 \\ 2 &= 5 \cdot 514 + (-107) \cdot 24 \\ 0 &= (-12) \cdot 514 + 257 \cdot 24 \end{aligned}$$

Of course (as before) at this point, you see that the $r' = 0$. Therefore

$$\gcd(514, 24) = 2$$

and furthermore from $c = 5, d = -107$, we get

$$5 \cdot 514 + (-107) \cdot 24 = \gcd(514, 24)$$

Here's a Python implementation with some test code:

```
ALGORITHM: EEA
INPUTS: a, b
OUTPUTS: r, c, d where r = gcd(a, b) = c*a + d*b

a0, b0 = a, b
d0, d = 0, 1
c0, c = 1, 0
q = a0 // b0
r = a0 - q * b0

while r > 0:
    d, d0 = d0 - q * d, d
    c, c0 = c0 - q * c, c

    a0, b0 = b0, r
    q = a0 // b0
    r = a0 - q * b0

r = b0
return r, c, d
```

By the way, this is somewhat similar to what we call *tail recursion* (CISS445) an extremely important technique in functional programming. All LISP hackers and people working in high performance computing and compilers swear by it. You don't see recursion in the above code, but you can replace the while-loop with recursion and if you have a compiler/interpreter that can perform true tail recursion, then it would run exactly like the above algorithm.

Exercise 1.6.4. Leetcode 365
https://leetcode.com/problems/water-and-jug-problem/description/
(Also, Die Hard 3 problem https://www.math.tamu.edu/~dallen/hollywood/diehard/diehard.htm.) You are given two jugs with capacities `jug1Capacity` and `jug2Capacity` liters. There is an infinite amount of water supply available. Determine whether it is possible to measure exactly `targetCapacity` liter using these two jugs.

If `targetCapacity` liters of water are measurable, you must have `targetCapacity` liters of water contained within one or both buckets by the end.

Operations allowed:

1. Fill any of the jugs with water.
2. Empty any of the jugs.
3. Pour water from one jug into another till the other jug is completely full, or the first jug itself is empty.

You'll see that there are times when you're only interested in the value of x and not y (or y and not x – the above is symmetric about x and y). Do you notice x comes from c? If you analyze the above algorithm, you see immediately that c is computed from c' and c' is computed from c, c', q, q is computed from r, r', r is computed from r', and finally (phew!) r' is computed from r, q, r'. Therefore if you're interested in c, you don't need to compute d or d'. So you can change the EEA to this:

```
ALGORTHM: EEA2 (sort of EEA ... without the d, d0)
INPUTS: a, b
OUTPUTS: r, c where r = gcd(a, b) = c*a + d*b for some d

a0, b0 = a, b
c0, c = 1, 0
q = a0 // b0
r = a0 - q * b0

while r > 0:
    c, c0 = c0 - q * c, c

    a0, b0 = b0, r
    q = a0 // b0
    r = a0 - q * b0

r = b0
return r, c
```

Later you'll see why we compute only c. It's not that we have something against d.

Exercise 1.6.5. Compute the following gcd and the Bézout's coefficients of the given numbers by following the Extended Euclidean Algorithm.

1. $\gcd(0, 10)$
2. $\gcd(10, 0)$
3. $\gcd(10, 1)$
4. $\gcd(10, 10)$
5. $\gcd(107, 5)$
6. $\gcd(107, 26)$
7. $\gcd(84, 333)$
8. $\gcd(F_{10}, F_{11})$ where F_n is the n–th Fibonacci number. (Recall: $F_0 = 0, F_1 = 1, F_{n+2} = F_{n+1} + F_n$.)
9. $\gcd(ab, b)$
10. $\gcd(a, a + 1)$
11. $\gcd(ab + a, b)$ where $0 < a < b$. Go as far as you can.
12. $\gcd(a(a + 1) + a, (a + 1))$ where $0 < a < b$. Go as far as you can.

Exercise 1.6.6. Prove that if $a \mid c$, $b \mid c$, and $\gcd(a, b) = 1$, then $ab \mid c$.

Exercise 1.6.7. Prove that if $a \mid c$, $b \mid c$, then

$$\frac{ab}{\gcd(a, b)} \mid c$$

Exercise 1.6.8. Leetcode 920.
https://leetcode.com/problems/number-of-music-playlists
Your music player contains n different songs. You want to listen to goal songs (not necessarily different) during your trip. To avoid boredom, you will create a playlist so that:

- Every song is played at least once.
- A song can only be played again only if k other songs have been played.

Given `n`, `goal`, and `k`, return the number of possible playlists that you can create. Since the answer can be very large, return it modulo $109 + 7$.

1.7 Primes

Definition 1.7.1. A prime p is a positive integer greater than 1 that is divisible by only 1 and itself. In other words $p \in \mathbb{N}$ is a **prime** if $p > 1$ and if $d \mid p$, then $d = 1$ or $d = p$. prime

Examples of primes are $2, 3, 5, 7, 11, 13, 17, 19, ...$

Integers least zero can be divided into the following types:

- 0 – the zero element
- 1 – the unit element (i.e. the only invertible element ≥ 0)
- primes – $2, 3, 5, 7, 11, ...$
- **composites** – integers > 1 which are not primes composites

(Instead of primes of $\mathbb{N} \cup \{0\}$, it's also possible to define primes of $\mathbb{Z}$. A prime of $\mathbb{Z}$ is an integer not $-1, 0, 1$ such that if $d \mid p$, then $d = \pm 1$ or $d = \pm p$. In that case primes of $\mathbb{Z}$ are $\pm 2, \pm 3, \pm 5, \pm 7, \pm 11, ...$)

Proposition 1.7.1. (Euclid) *There are infinitely many primes.*

Proof. TODO

Proposition 1.7.2. (Euclid) *There are arbitrarily long consecutives integers of composites.*

Proof. $n! + 2$, $n! + 3$, $n! + 4$, ..., $n! + n$ are all composities for $n \geq 2$. This list is therefore a list of consecutive composites of length $n - 1$. □

The follow lemma is extremely important and is used for instance in the fundamental theorem of arithmetic to prove the uniqueness of prime factorization in $\mathbb{N}$ (or $\mathbb{Z}$). To break tradition, we will call this a theorem (instead of a lemma):

Theorem 1.7.1. (**Euclid's lemma**) *If p is a prime and $p \mid ab$, then either $p \mid a$ or $p \mid b$.* Euclid's lemma

Proof. TODO

The above generalizes easily (by induction) to the following:

Corollary 1.7.1. *If p is a prime and $p \mid a_1 a_2 \cdots a_n$, then p divides at least one of the $a_1, \ldots, a_n$.*

Proof. We will prove this by strong induction on the number of terms in $a_1, ..., a_n$. Hence let $P(n)$ be the above statement, i.e., $P(n)$ is the statement that if a prime divides the product of n terms, then p divides at least one of the terms.

The case of $n = 2$ is Euclid's lemma. Hence $P(2)$ holds. This is the base case of our induction.

Now assume $P(k)$ is true for $2 \leq k \leq n$. Let p be a prime divide $a_1 a_2 \cdots a_n a_{n+1}$. Therefore p divides $a_1 a_2 \cdots a_{n-1} b$ (there are $n-1$ terms) where $b = a_n a_{n+1}$. Since $P(n)$ is true, p divides at least one of $a_1, ..., a_{n-1}, b$. Since $P(2)$ is true, p divides $b = a_n a_{n+1}$ if p divides a_n or a_{n+1}. Hence p divides at least one of $a_1, ..., a_{n+1}$. Therefore $P(n+1)$ holds. □

Fundamental Theorem of Arithmetic

Theorem 1.7.2. (Euclid's **Fundamental Theorem of Arithmetic**) *Every positive integer > 1 can be written as a unique product of primes up to permutation of the prime factors. This means*

(a) If $n > 1$ is an integer, then n can be written as a product of primes.
(b) If n is written as two products of primes:

$$n = p_1 p_2 \cdots p_k = q_1 q_2 \cdots q_\ell$$

where p_i and q_j are primes arranged in ascending order, i.e.,

$$p_1 \leq p_2 \leq \cdots \leq p_k$$
$$q_1 \leq q_2 \leq \cdots \leq q_\ell$$

then $k = \ell$ and

$$p_1 = q_1, \quad p_2 = q_2, \quad \cdots, \quad p_k = q_k,$$

Proof. TODO

The statement of the Fundamental Theorem of Arithmetic can include the case of $n = 1$ if we accept that the product of an empty set of integers is 1:

$$\prod_{p \in \{\}} p = 1$$

Proposition 1.7.3. *Let* $a = \prod_{p \in P} p^{a_p}$, $b = \prod_{p \in P} p^{b_p}$ *and* $c = \prod_{p \in P} p^{c_p}$ *where* P *is a finite set of primes. Then*

(a) $c = ab \implies c_p = a_p + b_p$.
(b) $a \mid b \implies a_p \leq b_p$ *for all* $p \in P$.
(c) $c = \gcd(a, b) \implies c_p = \min(a_p, b_p)$.

The above assumes the easily proven facts that

$$\prod_{p \in P} p^{a_p} \prod_{p \in P} p^{b_p} = \prod_{p \in P} p^{a_p + b_p}$$

(by commutativity of $\cdot$) and

$$\prod_{p \in P} p^{a_p} = \prod_{p \in P} p^{b_p} \implies a_p = b_p \text{ for all } p \in P$$

by uniqueness of prime factorization from the Fundamental Theorem of Arithmetic. Here, P is a set of distinct primes.

Proposition 1.7.4. *If* $n > 1$ *is not a prime, then there is a prime factor* p *such that* $p \leq \sqrt{n}$.

Proof. TODO □

Therefore a very simple primality test algorithm for n is the following:

```
ALGORITHM: BRUTE-FORCE-PRIMALITY-TEST
INPUT: n
OUTPUT: true if n is prime. If n < 2, false is returned.

if n < 2: return false

d = 2
while d <= sqrt(n):
    if n % d == 0:
        return false
    d = d + 1
return true
```

In terms of n, the runtime is $O(\sqrt{n})$. However in terms of the bits of n, by Proposition 1.5.2, the number of n is

$$b = \lfloor \log_2(n+1) \rfloor = \log_2(n+1) - \alpha$$

where $0 \leq \alpha < 1$. Hence

$$n = 2^b 2^\alpha - 1$$

Hence in terms of the number of bits of n, the runtime is

$$O(\sqrt{n}) = O(\sqrt{2^b 2^\alpha - 1}) = O(2^{b/2})$$

It is common to denote the number of bits of the input by n. Hence the runtime in number of bits n is $O(2^{n/2})$, i.e., it has **exponential runtime with linear exponent**.

exponential runtime with linear exponent

$O(\sqrt{n})$ is said to be the **pseudo-polynomial runtime** of the algorithm to indicate that the n is the numeric input and not a correct measure of the complexity of the input, which should be in number of bits of the input.

pseudo-polynomial runtime

Exercise 1.7.1. Let $P = \{p_1, ..., p_n\}$ be a set of distint primes. Consider the expression $N(P) = \prod_{p \in P} p + 1$ in Euclid's proof of infinitude of primes. This is sometimes called Euclid's construction. How often is this a prime?

Exercise 1.7.2. Let $P = \{p_1, ..., p_n\}$ be a set of distint primes. Carry out the Euclid's construction on all possible subsets of P. If a Euclid construction is a prime, put that prime into P. If it does not, put the smallest prime factor into P. Repeat. For instance if you start with $P = \{\}$, you'll get 2 and the new P is $\{2\}$. Next you'll get $P = \{2, 3\}$. The Euclid constructions you get from P are $2, 3, 4, 7$. So the next P is $\{2, 3, 7\}$. At the next stage you get $2, 3, 4, 8, 7, 15, 22, 43$. This means that the next P is $\{2, 3, 7, 43\}$. Etc. Does your P always grow? Notice that your P's so far does not capture 5. When, if at all, will 5 appear?

Exercise 1.7.3. In the above, if an Euclid construction does not give you a prime, you take the smallest prime factor. What if you pick the largest prime factor?

The following are some DIY exercises for self-study on famous unsolved problems in number theory. You might want to write programs to check on the conjectures.

Exercise 1.7.4. Are there infinitely many primes p such that $p + 2$ is also a

prime? If p and $p+2$ are both primes, then they are called **twin primes**. The **twin prime conjecture** states that there are infinitely many twin primes. (See https://en.wikipedia.org/wiki/Twin_prime.) Write a program that prints $p, p+2$ if both are primes. Print the time elapsed between the discovery of pairs of twin primes.

twin primes

twin prime conjecture

Exercise 1.7.5. Can even positive integer be written as the sum of two primes? When the two primes are > 2, then the sum of these two primes is even. The **Goldbach conjecture** states that every positive integer can be written as the sum of two primes. (See https://en.wikipedia.org/wiki/Goldbach%27s_conjecture.) Write a program that prints $n \geq 2$ as a sum of two primes as n iterates from 2 to a huge positive integer N entered by the user.

Goldbach conjecture

Exercise 1.7.6. A positive integer is a **perfect** number if it is the sum of its positive divisors strictly less than itself. For instance 6 is perfect since $6 = 1 + 2 + 3$. 28 is also perfect since $28 = 1 + 2 + 4 + 7 + 14$. Euclid knew that if p is prime and $2^p - 1$ is a prime, then $2^{p-1}(2^p - 1)$ is an even perfect number. If p is a prime and $2^p - 1$ is a also a prime, then $2^p - 1$ is called a **Mersenne prime**. Almost 2000 years after Euclid, Euler proved the converse, that every even perfect number must be of the form $2^{p-1}(2^p - 1)$ where $2^p - 1$ is a Mersenne prime. There are two famous unsolved problems in number theory on perfect numbers:

1. Are there odd perfect numbers?
2. Are there infinitely many perfect numbers?

(See https://en.wikipedia.org/wiki/Perfect_number.) There is an ongoing search for primes and Mersenne primes using computers. At this point (2023), the top 8 largest known primes are all Mersenne primes. (See https://en.wikipedia.org/wiki/Great_Internet_Mersenne_Prime_Search.) As of Feb 2023, the largest known prime is $2^{82,589,933} - 1$. (See https://en.wikipedia.org/wiki/Largest_known_prime_number.) Write a program that prints perfect numbers, printing the time between pairs of perfect numbers discovered.

Exercise 1.7.7. Leetcode 204
https://leetcode.com/problems/count-primes/
Given an integer n, return the number of prime numbers that are strictly less

than n.

Exercise 1.7.8. Leetcode 507
https://leetcode.com/problems/perfect-number/
A perfect number is a positive integer that is equal to the sum of its positive divisors, excluding the number itself. A divisor of an integer `x` is an integer that can divide `x` evenly. Given an integer `n`, return `true` if `n` is a perfect number, otherwise return `false`.

Exercise 1.7.9. Leetcode 866
https://leetcode.com/problems/prime-palindrome/
Given an integer n, return the smallest prime palindrome greater than or equal to n. An integer is prime if it has exactly two divisors: 1 and itself. Note that 1 is not a prime number. For example, 2, 3, 5, 7, 11, and 13 are all primes. An integer is a palindrome if it reads the same from left to right as it does from right to left. For example, 101 and 12321 are palindromes. The test cases are generated so that the answer always exists and is in the range $[2, 2 * 108]$.

Exercise 1.7.10. Leetcode 1175
https://leetcode.com/problems/prime-arrangements/ Return the number of permutations of 1 to n so that prime numbers are at prime indices (1-indexed.) (Recall that an integer is prime if and only if it is greater than 1, and cannot be written as a product of two positive integers both smaller than it.) Since the answer may be large, return the answer modulo $10^9 + 7$.

Exercise 1.7.11. Leetcode 1362
https://leetcode.com/problems/closest-divisors/
Given an integer `num`, find the closest two integers in absolute difference whose product equals `num + 1` or `num + 2`. Return the two integers in any order.

Exercise 1.7.12. Leetcode 1390
https://leetcode.com/problems/four-divisors/
Given an integer array `nums`, return the sum of divisors of the integers in that array that have exactly four divisors. If there is no such integer in the array, return `0`.

Exercise 1.7.13. Leetcode 2523
https://leetcode.com/problems/closest-prime-numbers-in-range/
Given two positive integers `left` and `right`, find the two integers `num1` and `num2` such that:

- `left <= nums1 < nums2 <= right`
- `nums1` and `nums2` are both prime numbers.
- `nums2 - nums1` is the minimum amongst all other pairs satisfying the above conditions.

Return the positive integer array `ans = [nums1, nums2]`. If there are multiple pairs satisfying these conditions, return the one with the minimum nums1 value or `[-1, -1]` if such numbers do not exist. A number greater than 1 is called prime if it is only divisible by 1 and itself.

Exercise 1.7.14. Leetcode 2521
https://leetcode.com/problems/distinct-prime-factors-of-product-of-array/
Given an array of positive integers `nums`, return the number of distinct prime factors in the product of the elements of `nums`.

Exercise 1.7.15. Leetcode 1071
https://leetcode.com/problems/greatest-common-divisor-of-strings/
For two strings s and t, we say "t divides s" if and only if $s = t + \cdots + t$ (i.e., t is concatenated with itself one or more times). Given two strings `str1` and `str2`, return the largest string `x` such that `x` divides both `str1` and `str2`.

1.8 Multiplicative inverse in $\mathbb{Z}/N$

Quick review: In algebra classes, lots of time is spent on solving equations. You usually start with something like: "Find the roots of $x + b$". This means finding a value for x such that

$$x + a = 0$$

Of course this is dead easy. As long as you have the concept of $-a$ (additive inverse) you just add $-a$ to both sides and voila:

$$\begin{aligned}(x + a) + (-a) &= 0 + (-a)\\ x + (a + (-a)) &= 0 + (-a)\\ x + 0 &= 0 + (-a)\\ x &= 0 + (-a)\\ x &= -a\end{aligned}$$

Next up, you usually see this: "Find the roots of $ax + b$" which is the same as finding a value for x such that

$$ax + b = 0$$

Assuming you are working in $\mathbb{R}$, you would do something like this:

$$\begin{aligned}(ax + b) + (-b) &= 0 + (-b)\\ ax + (b + (-b)) &= 0 + (-b)\\ ax + 0 &= 0 + (-b)\\ ax &= 0 + (-b)\\ ax &= -b\end{aligned}$$

and at this point you would do this:

$$\begin{aligned}ax &= -b\\ a^{-1}(ax) &= a^{-1}(-b)\\ (a^{-1}a)x &= a^{-1}(-b)\\ 1x &= a^{-1}(-b)\\ x &= a^{-1}(-b)\end{aligned}$$

and vóila (again) and you're done.

In the case of real numbers (or even just fractions), if you're given a number a

(which is not zero), you always have another number called a^{-1} (multiplicative inverse) such that

$$a \cdot a^{-1} = 1 = a^{-1} \cdot a$$

The reason why we like this is exactly because it allows us to solve the above equation.

Given a number a, the number $-a$ is called an **additive inverse** if it behaves like this: additive inverse

$$a + (-a) = 0 = (-a) + a$$

The number a^{-1} is called a **multiplicative inverse** of a if it does this: multiplicative inverse

$$a \cdot a^{-1} = 1 = a^{-1}a$$

If a has a multiplicative inverse, we also say that a is invertible.

Definition 1.8.1. We also say that a is a **unit** if a has a multiplicative inverse. unit

Definition 1.8.2. The set of multiplicatively invertible elements of $\mathbb{Z}/N$ is denoted by $(\mathbb{Z}/N)^\times$ or $U(\mathbb{Z}/N)$.

Almost all values in $\mathbb{R}$ are invertible; the only exception being 0. On the other hand, note that most numbers in $\mathbb{Z}$ do not have multiplicative inverses. In fact the only numbers in $\mathbb{Z}$ that have inverses are 1 and -1. So the only units in $\mathbb{Z}$ are $1, -1$.

Proposition 1.8.1. *The only units of $\mathbb{Z}$ are 1, -1. In particular, $1^{-1} = 1$ and $(-1)^{-1} = -1$.*

Proof. The fact $1 \cdot 1 = 1$ follows from the neutral axiom of $\cdot$. Hence $1^{-1} = 1$. $(-1)(-1) = 1$ is from Proposition 1.5.1(d). Hence $(-1)^{-1} = 1$.

Let $a \in \mathbb{Z}$ be invertible. Then $a \cdot a^{-1} = 1$. If $a = 0$, then $a \cdot a^{-1} = 1$ and $a \cdot a^{-1} = 0 \cdot a^{-1} = 0$. In other words $0 = 1$. But this contradicts the nontriviality axiom of $\mathbb{Z}$. Hence $a \neq 0$. Either $a > 0$ or $a < 0$.

First suppose $a > 0$. Note that $a^{-1} > 0$, otherwise $a \cdot a^{-1} < 0$ which contradicts $a \cdot a^{-1} = 1$. Since $a > 0$ and $a^{-1} > 0$, they have prime factorizations. Let p be a prime. Suppose the p–power appearing in the prime factorization of a is p^α

where $\alpha \geq 0$ and the p–power appearing in the prime factorization of a^{-1} is p^{β} where $\alpha \geq 0$. Then the p–power that appears in $a \cdot a^{-1}$ is $p^{\alpha+\beta}$. Note that $a \cdot a^{-1} = 1$ and the p–power that appears in the prime factorization of 1 is p^0. Therefore $p^{\alpha+\beta} = p^0$. Since $\alpha = 0 = \beta$. This implies that p does not appear in the prime factorization of a nor a^{-1}. Since this holds for all primes p, the prime factorization of a is $\prod_{p \in \emptyset} p$, i.e., $a = 1$.

Now suppose $a < 0$. Then $-a > 0$. Then the above case applied to $-a$ implies that $-a = 1$. Hence $-(-a) = -1$, i.e., $a = -1$ by Proposition 1.5.1(b). Hence $a = -1 = a^{-1}$.

Therefore the only units of $\mathbb{Z}$ are $1, -1$. □

Recall from the section on congruences, we have a relation $\equiv \pmod{N}$ that relates values of $\mathbb{Z}$. For instance when $N = 26$,

$$3 \equiv 29 \pmod{26}$$

and

$$3 \equiv -26 \pmod{26}$$

You can think of $\equiv \pmod{26}$ as a kind of equality on $\mathbb{Z}$ so that, in mod 26, 3 and 29 are the same and 3 and -26 are the same. In this context, write $\mathbb{Z}/N$ for the set

$$\mathbb{Z}/N = \{0, 1, 2, ..., N-1\}$$

We call the set $\mathbb{Z}/\mathbb{N}$ "$\mathbb{Z}$ mod N". Note that this set $\mathbb{Z}/N$ also contains N, but N is "the same as" 0, i.e., N is just another way to write 1 in the sense that

$$N \equiv 1 \pmod{N}$$

Note that $\mathbb{Z}/N$ also has $+$, $\cdot$ and has 0 and 1.

In fact $(\mathbb{Z}/N, +, \cdot, 0, 1)$ satisfies the algebraic properties of $+$, properties of $\cdot$, and distributivity. If $N > 1$, $(\mathbb{Z}/N, +, \cdot, 0, 1)$ also satisfies the nontriviality axiom, i.e., $0 \not\equiv 1 \pmod{N}$. However $(\mathbb{Z}/N, +, \cdot, 0, 1)$ does not satisfy the integrality axiom. Furthermore there's no concept of order $<$ on $\mathbb{Z}/N$ and hence there's no WOP or induction on $\mathbb{Z}/N$.

Later we will define the concept of commutative rings which are exactly sets, each with its own $+$ and $\cdot$ operations, satisfying the properties of $+$ and $\cdot$ and distributivity of $\mathbb{Z}$. $\mathbb{Z}$ and $\mathbb{Z}/N$ are both commutative rings.

The ability for a number in some commutative ring to have a multiplicative inverse (in that ring) is special.

On the other hand, $\mathbb{Z}/N$ ($N > 1$) might contain lots of units. For instance look at $\mathbb{Z}/10$. Is 3 invertible mod 10? In other words is there some x such that

$$3x \equiv 1 \pmod{10}$$

$x \equiv 7 \pmod{10}$ satisfies the above. In other words in mod 10, 7 is the multiplicative inverse of 3.

(You can talk about $\mathbb{Z}/N$ for $N = 1$, but it's not very interesting nor useful. $\mathbb{Z}/1$ has only one value, i.e., 0 which behaves like the additive identity element 0 and the multiplicative identity element 1.)

Here's the obvious brute algorithm to find the multiplicative inverse of an integer mod N:

```
ALGORITHM: brute-force-inverse
INPUT: a, N
OUTPUT: x such that a * x % N is 1

for x = 1, 2, 3, ..., N - 1:
    if a * x % N == 1:
        return x
return None
```

Note that in the above algorithm we need not test 0.

Exercise 1.8.1.

(a) Find all the units and their multiplicative inverses in $\mathbb{Z}/10$.
(b) Find all the units and their multiplicative inverses in $\mathbb{Z}/5$.
(c) Find all the units and their multiplicative inverses in $\mathbb{Z}/6$.

Do you see a pattern? □

It turns out that in $\mathbb{Z}/N$, you can easily find all the elements with multiplicative inverses: $a \in \mathbb{Z}/N$ has a multiplicative inverse if $\gcd(a, N) = 1$.

Proposition 1.8.2. *Let a, N be integers where $N > 0$. Then a is (multiplicatively) invertible in $\mathbb{Z}/N$ iff* $\gcd(a, N) = 1$.

If m, n are two integers such that $\gcd(m, n) = 1$, we say that m and n are **coprime**. coprime

Proof. TODO □

The above immediately implies that

$$\begin{aligned}(\mathbb{Z}/N)^\times &= \{a \mid 0 \le a \le N-1,\ \ a \text{ is invertible mod } N\} \\ &= \{a \mid 0 \le a \le N-1,\ \ \gcd(a,N) = 1\}\end{aligned}$$

(Of course $\gcd(0, N) = N > 0$, so we can throw away 0 in the above if $N > 1$.) So the key is the computation of x, y such that

$$1 = \gcd(a, N) = ax + Ny$$

which is achieved by the Extended Euclidean Algorithm. But wait a minute ... I just need the x. So I'll just modify the Extended Euclidean Algorithm slightly.

Here's the (earlier) EEA algorithm, slightly modified, together with a function to compute the multiplicative inverse of $a \pmod N$:

```
ALGORITHM: EEA2 (sort of EEA ... without the d, d0)
INPUTS: a, b
OUTPUTS:  r, c where r = gcd(a, b) = c*a + d*b for some d

a0, b0 = a, b
c0, c = 1, 0
q = a0 // b0
r = a0 - q * b0

while r > 0:
    c, c0 = c0 - q * c, c

    a0, b0 = b0, r
    q = a0 // b0
    r = a0 - q * b0

r = b0
return r, c


ALGORITHM: mod-inverse
INPUTS: a, N
OUTPUT: x such that  (a * x) % N is 1

g, x = EEA2(a, N)
if g == 1:
    return x % N
else:
```

```
    return None
```

Example 1.8.1. Does 135 have an inverse mod 1673? If it does, find it using the Extended Euclidean Algorithm.

Solution.

1. c0, c, q, r: 1, 0, 0, 135
2. c0, c, q, r: 0, 1, 12, 53
3. c0, c, q, r: 1, -12, 2, 29
4. c0, c, q, r: -12, 25, 1, 24
5. c0, c, q, r: 25, -37, 1, 5
6. c0, c, q, r: -37, 62, 4 4
7. c0, c, q, r: 62, -285, 1, 1
8. c0, c, q, r: -285, 347, 4, 0
9. r, c: 1, 347

Therefore $135^{-1} \pmod{1673}$ is 347. And we check:

$$135 \cdot 347 = 34845 = 1 + 28 \cdot 1673 \equiv 1 \pmod{1673}$$

Exercise 1.8.2. Compute the $\gcd(16, 123)$. If it's 1, find x such that $16x \equiv 1 \pmod{123}$. Use the above version of Extended Euclidean Algorithm and compute by hand. When you're done, write a program implementing the above algorithm and check that it gives you the same result. Solve the equation

$$16x + 5 \equiv 0 \pmod{123}$$

i.e. find an integer x such that $0 \leq x < 123$ satisfying the above congruence.

Exercise 1.8.3. In your `Zmod.py`, complete the following methods:

- multiplicative inverse mod N (i.e., `inv`)
- invertibility mod N (i.e., `is_invertible`)
- division (i.e., `__div__`)

1.9 Euler Totient Function

Definition 1.9.1. Let N be a positive integer. $\phi(N)$ is the number of positive integers from 0 to $N-1$ which are coprime to N, i.e.

$$\phi(N) = |\{a \,|\, 0 \leq a \leq N-1, \ \gcd(a,N) = 1\}|$$

Note that you can also view ϕ in this way:

$$\begin{aligned}\phi(N) &= |\{a \mid 0 \leq a \leq N-1, \ \gcd(a,N) = 1\}| \\ &= |\{a \mid 0 \leq a \leq N-1, \ a \text{ is invertible mod } N\}| \\ &= |(\mathbb{Z}/N^\times)|\end{aligned}$$

Recall that $\{a \,|\, 0 \leq a \leq N-1, \ \gcd(a,N) = 1\}$ is the set of units of $\mathbb{Z}/N$ which is denoted by $(\mathbb{Z}/N)^\times$ or $U(\mathbb{Z}/N)$. So the above definition is the same as

$$\phi(N) = |U(\mathbb{Z}/N)|$$

Note that by definition $\phi(1) = 1$. Here are some important properties of ϕ.

Proposition 1.9.1. *Let $n > 0$ be a positive integer.*

(a) *Then*

$$\phi(n) = n \prod_{p|N} \left(1 - \frac{1}{p}\right)$$

where "$\prod_{p|n}$" is "product over all primes p dividing n".

(b) *If m, n are coprime, i.e. $\gcd(m,n) = 1$, then $\phi(mn) = \phi(m)\phi(n)$.*

(c) *If p is a prime and $k > 0$, then $\phi(p^k) = p^{k-1}(p-1) = p^k - p^{k-1}$.*

Proof. TODO □

Let's compute $\phi(10)$. Note that $10 = 2 \cdot 5$ and $\gcd(2,5) = 1$. Therefore using (b) of the above theorem we get

$$\phi(10) = \phi(2^1 \cdot 5^1) = \phi(2^1) \cdot \phi(5^1)$$

since $\gcd(2^1 \cdot 5^1) = 1$. Using (c) of the above theorem I get

$$\phi(10) = \phi(2^1) \cdot \phi(5^1) = (2^1 - 2^{1-1}) \cdot (5^1 - 5^{1-1}) = 1 \cdot 4 = 4$$

Of course you can also use (a) above to get

$$\phi(10) = 10 \cdot (1 - 1/2) \cdot (1 - 1/5) = 10 \cdot \frac{1}{2} \cdot \frac{4}{5} = 4$$

In both cases, we get 4. This means in $\mathbb{Z}/10$, there are 4 elements which are invertible. Running a through $\{0, 1, 2, ..., 9\}$, you'll see that invertible a's are $1, 3, 7, 9$ with inverses $1, 7, 3, 9$ respectively.

Exercise 1.9.1.

(a) Compute $\phi(735)$.
(b) Compute $\phi(900)$.
(c) Compute $\phi(263891)$.

Exercise 1.9.2. (a) Let p be a prime. What is $\phi(2p)$ in terms of p?
(b) How many solutions are there to $\phi(n) = 2n$?
(c) How many solutions are there to $\phi(n) = n/2$?

Exercise 1.9.3. Easy: What is $\phi(pq)$ as an integer expression involving p and q? Can you write it as an expression involving the sum and product of p and q? (i.e., besides constants and operators, your expression contains only $p + q$ and pq).

Exercise 1.9.4.

(a) Solve $\phi(n) = 2$, i.e., find all positive integers n such that $\phi(n) = 2$. (Hint: Write down the prime factorization of $n = p_1^{e_1} \cdots p_g^{e_g}$ and use the equation $\phi(n) = 2$.)
(b) Solve $\phi(n) = 3$.
(b) Solve $\phi(n) = 6$.

Exercise 1.9.5. Prove that

$$\phi(mn) = \phi(m)\phi(n) \cdot \frac{g}{\phi(g)}$$

where $g = \gcd(m, n)$. (Note that the above does *not* assume m, n are coprime. What is the above if m, n are coprime?)

Exercise 1.9.6. * Plot a function of the graph $y = \phi(x)$ for integer values of x running through 1 to 10000. See any pattern? Note that if you want an approximation (for instance in the asymptotics) that's not too difficult. Note the following: There are two ways to compute $\phi(n)$:

(a) $\phi(n) = |\{x \mid 0 \leq x < n-1, \ \gcd(x,n) = 1\}|$ which required Euclidean algorithm in a loop.
(b) $\phi(n) = \phi(p_1^{\alpha_1} \cdots p_g^{\alpha_g}) = (p_1^{\alpha_1} - p_1^{\alpha_1 - 1}) \cdots (p_1^{\alpha_g} - p_1^{\alpha_g - 1})$ which requires prime factorization.

The first method is slow: you need to loop your x and for each x you need to execute the EEA which has a loop. The second method is fast only if you can find the prime factorization of n. Is it possible to find $\phi(n)$ as a formula in n without finding the prime factorization of n? For instance the factorial function $n!$ has this interpolation: If

$$\Gamma(x) = \int_0^\infty t^{x-1} e^{-t} \, dt$$

then $\Gamma(n) = n!$ for positive integers n. Is there a fast interpolation of $\phi(n)$?

Exercise 1.9.7. * Can you find an n such that $\phi(n)$ divides $n + 1$?

Exercise 1.9.8. *

(a) If p is a prime, then $\phi(p) = p - 1$. Prove that if $\phi(n) = n - 1$, then n is a prime.
(b) Can you find an $n > 1$ which is not a prime (i.e. composite) and such that $\phi(n)$ divides $n - 1$? If you can find one, let me know ASAP. Or if you can prove that such as n does not exist, let me know ASAP.

Exercise 1.9.9. *

(a) Can you find some n such that

$$\phi(\phi(n)) = 1$$

(b) Write $\phi^2(n) = \phi(\phi(n))$. Can you find *all* n such that $\phi^2(n) = 1$.
(c) Write ϕ^k to the composition of k Euler ϕ. What about $\phi^3(n) = 1$? Can you find some n satisfying the above equation?

(d) What about $\phi^k(2^n) = 1$? What is the smallest k for $\phi^k(2^n) = 1$?
(e) What about $\phi^k(2^m \cdot 3^n) = 1$? What is the smallest k such that $\phi^k(2^m \cdot 3^n) = 1$?

As an aside, note that ϕ as a function has domain of $\mathbb{N}$. In this case, we say that ϕ is an **arithmetic function**. Furthermore, ϕ satisfies the property that if $\gcd(m, n) = 1$, then $\phi(mn) = \phi(m)\phi(n)$. A function $\mathbb{N} \to \mathbb{C}$ satisfying this property is said to be **multiplicative** . The Euler ϕ function is one of many multiplicative arithmetic functions. Multiplicative functions are extremely important in number theory.

arithmetic function

multiplicative

The following theorem allows you to compute powers in mod p extremely fast:

Theorem 1.9.1. (**Fermat's Little Theorem**). *Let p is a prime number and a be a positive integer not divisible by p. Then*

Fermat's Little Theorem

$$a^{p-1} \equiv 1 \pmod{p}$$

Proof. TODO □

Exercise 1.9.10. Compute r where r is the smallest positive integer satisfying

$$5^{642} \equiv r \pmod{641}$$

Corollary 1.9.1. *Let p be a prime. Then $a^p \equiv a \pmod{p}$.*

Note that the corollary does not require $p \nmid a$. The proof of the corollary is easy. If $p \mid a$, then both sides of the equation is 0 mod p, so the congruence is true. If $p \nmid a$, then Fermat's Little Theorem gives us

$$a^{p-1} \equiv a \pmod{p}$$

on multiplying both sides by a, we get

$$a^p \equiv a \pmod{p}$$

Exercise 1.9.11. What is the remainder of $3^{122436481}$ mod 13?

Note that Fermat's Little Theorem can be used to compute powers very rapidly if you work in mod p where p is a prime. What if you need to work in mod N where N is not a prime? There is a generalization of Fermat's Little Theorem due to Euler. Note that since $p - 1 = \phi(p)$, Fermat's Little Theorem

$$a^{p-1} \equiv 1 \pmod{p}$$

can be stated as

$$a^{\phi(p)} \equiv 1 \pmod{p}$$

This statement actually holds if p is replaced by any positive integer.

Theorem 1.9.2. (**Euler's Theorem**). *Let a and n be positive integers such that $\gcd(a, n) = 1$. Then*

Euler's Theorem

$$a^{\phi(n)} \equiv 1 \pmod{n}$$

Proof. TODO □

Note that for $\mathbb{Z}/N$, a computation of

$$a^k \pmod{N}$$

Euler's Theorem can lower the k if $\gcd(a, N)$. But after you have lowered the k, say to ℓ, you still need to compute $a^\ell \pmod{N}$. See the squaring algorithm in the RSA chapter.

Exercise 1.9.12. Compute r where r is the smallest positive integer satisfying

$$5^{642} \equiv r \pmod{640}$$

Exercise 1.9.13. What is the remainder of $3^{123456789}$ mod 100?

Exercise 1.9.14. What is the hundreds digit of $3^{123456789}$?

Exercise 1.9.15. In your `Zmod.py` complete the following:

- Exponentiation (i.e., `__pow__`)

Note that $x^{-1000} \pmod N$ is $(x^{-1})^{1000} \pmod N$. You want to first check if you can use Euler's Theorem. If it is, use the theorem to lower the exponent to say ℓ. Then use the obvious loop to compute x^ℓ and apply $\pmod N$ as frequently as possible. After you are done with the above, you might want to improve your `__pow__` by *not* use Euler's theorem if the exponent is "small". You can determine for yourself what if "small". For instance you can choose to use Euler's Theorem only when the exponent is greater than 10. (Later in the RSA chapter, we will talk about the squaring method.)

Exercise 1.9.16. Leetcode 372.
https://leetcode.com/problems/super-pow/
Your task is to calculate $a^b \pmod{1337}$ where a is a positive integer and b is an extremely large positive integer given in the form of an array. For instance for $a = 2, b = [1, 0]$, the output is 1024.

Exercise 1.9.17. Leetcode 1015.
https://leetcode.com/problems/smallest-integer-divisible-by-k/
Given a positive integer k, you need to find the length of the smallest positive integer n such that n is divisible by k, and n only contains the digit 1. Return the length of n. If there is no such n, return -1. Note: n may not fit in a 64-bit signed integer.

Exercise 1.9.18. Leetcode 1622.
https://leetcode.com/problems/fancy-sequence/
Write an API that generates fancy sequences using the append, addAll, and multAll operations. Implement the Fancy class:

- `Fancy()` Initializes the object with an empty sequence.
- `void append(val)` Appends an integer val to the end of the sequence.
- `void addAll(inc)` Increments all existing values in the sequence by an integer inc.
- `void multAll(m)` Multiplies all existing values in the sequence by an integer `m`.
- `int getIndex(idx)` Gets the current value at index `idx` (0-indexed) of the sequence modulo 109 + 7. If the index is greater or equal than the length of the sequence, return `-1`.

Exercise 1.9.19. Leetcode 1952

https://leetcode.com/problems/three-divisors/
Given an integer `n`, return `true` if `n` has exactly three positive divisors. Otherwise, return `false`. An integer `m` is a divisor of `n` if there exists an integer `k` such that `n = k * m`.

Exercise 1.9.20. https://acm.timus.ru/problem.aspx?space=1&num=1673
At the end of the previous semester the students of the Department of Mathematics and Mechanics of the Yekaterinozavodsk State University had to take an exam in network technologies. N professors discussed the curriculum and decided that there would be exactly N^2 labs, the first professor would hold labs with numbers $1, N+1, 2N+1, ..., N^2-N+1$, the second one — labs with numbers $2, N+2, 2N+2, ..., N^2-N+2$, etc. N-th professor would hold labs with numbers $N, 2N, 3N, ..., N^2$. The professors remembered that during the last years lazy students didn't attend labs and as a result got bad marks at the exam. So they decided that a student would be admitted to the exam only if he would attend at least one lab of each professor. N roommates didn't know the number of labs and professors in this semester. These students had different diligence: the first student attended all labs, the second one — only labs which numbers were a multiple of two, the third one — only labs which numbers were a multiple of three, etc... At the end of the semester it turned out that only K of these students were admitted to the exam. Find the minimal N which makes that possible.

Input: An integer K ($1 \leq K \leq 2 \cdot 10^9$).

Output: Output the minimal possible N which satisfies the problem statement. If there is no N for which exactly K students would be admitted to the exam, output 0.

Example: Input:8, output:15. Input:3, output:0.

Chapter 2

Classical ciphers

2.1 Shift cipher

Definition 2.1.1. The **shift cipher** (E, D) is given by

$$E(k, x) = x + k \pmod{26}$$

and

$$D(k, x) = x - k \pmod{26}$$

Historically the shift cipher with key $k = 3$ was used by Julius Caesar and is called the **Caesar cipher**.

2.2 Affine cipher

2.3 Vigenère cipher

2.4 Substitution cipher

2.5 Permutation cipher

2.6 Hill cipher

2.7 One-time pad cipher

2.8 Linear feedback shift register

Chapter 3

RSA

3.1 RSA

Now for RSA ...

We only need to work with integers. Why? Because any message M is really just a sequence of bits and you can cut your sequences of bits which can be viewed as an unsigned int, i.e, an integer ≥ 0. So we'll just think of our messages as integers.

So once again suppose Alice wants to send a secret to Bob. The secret is an integer x.

Bob selects two distinct primes p and q. He computes $N = pq$ and selected two positive integers e and d such that

$$ed \equiv 1 \pmod{\phi(N)}$$

In other words e and d and multiplicative inverses of each other in $\mathbb{Z}/\phi(N)$. Furthermore Bob publishes N and e publicly. e is called the **encryption exponent** and d is called the **decryption exponent**.

decryption exponent

We assume that the secret x is less than N because we'll be working in $\mathbb{Z}/N$. (Again if x as a bit sequence is too large for N, then we cut x up into smaller block of bits and send them separately.)

Since N and e are public, Alice can download N and e and then compute

$$E_{(N,e)}(x) = x^e \pmod{N}$$

I'll write $x^e \pmod{N}$ for the least positive remainder of $x^e \bmod N$. She then sends $x^e \pmod{N}$ to Bob.

When Bob received $x^e \pmod N$, he computes

$$D_{(N,d)}(x) = x^{ed} \pmod N$$

What we need to prove is that RSA works, i.e.,

Theorem 3.1.1. *Let p, q be primes and $N = pq$. If $ed \equiv 1 \pmod{\phi(N)}$, then*

$$(x^e)^d \equiv x \pmod N$$

for all integers x.

Proof. Since $ed \equiv 1 \pmod{\phi(N)}$, we have

$$ed = k\phi(N) + 1$$

for some integer k. Therefore

$$x^{ed} = x^{k\phi(N)+1} = (x^{\phi(N)})^k x$$

We now consider several cases based on the possible values of $\gcd(x, N)$. Since $N = pq$, $\gcd(x, N)$ is 1, p, q or pq. since the only possible divisors of $N = pq$ are $1, p, q, pq$. The case of $\gcd(x, N) = p$ and $\gcd(x, N) = q$ are similar.

Case 1: $\gcd(x, N) = 1$. TODO

Case 2: $\gcd(x, N) = N$. TODO

Case 3: $\gcd(x, N) = p$. □

Proposition 3.1.1. *Prove the following:*

(a) *If p and q are distinct primes such that $p \mid a$ and $q \mid a$, then $pq \mid a$.*
(b) *If $x \mid a$ and $y \mid a$ and $\gcd(x, y) = 1$, then $xy \mid a$.*
(c) *If $x \mid a$ and $y \mid a$, then $(xy/\gcd(x, y)) \mid a$.*

(b) is a generalization of (a) and (c) is a generalization of (b). □

Proof. (a) TODO

(b) Since $x \mid a$ and $y \mid a$, we have $xk = a = y\ell$. Hence $y \mid xk$. Since $\gcd(x, y) = 1$, by Extended Euclidean Algorithm, there are integer A, B such

that

$$Ax + By = 1$$

Hence $Axk+Byk = k$. Since y divides Byk and Axk, by linearity of divisibility, y divides k. Hence $ym = k$. Therefore $a = xk = xym$, i.e., $xy \mid a$.

(c) Since $y \mid a$, $y/\gcd(x, y) \mid a$. Since $\gcd(x, y/\gcd(x, y)) = 1$, by (b), $xy/\gcd(x, y) \mid a$. □

Note that RSA (and all public key ciphers) are not meant for encrypting messages like strings (emails, sales receipts, image files, etc.) This is an unfortunate thing that's done in many cryptography textbooks. They are actually used to encrypt/decrypt keys for private ciphers (3DES, AES, etc.) In particular the RSA standard (called PKCS – you can easily find lots of webpages on PKCS) does not include specification on how to break up long messages before encryption and how to reassembly them after decryption. The reason for using RSA (and other public key cipher) is used to setup keys for private key cipher (example: AES) is because AES is much faster than RSA.

OK, let's summarize everything. In the following, I'll write $x \pmod N$ to be the remainder when x is divided by N. There are three steps: Bob has to generate keys, Alice has to encrypt, and Bob has to decrypt.

1. Key Generation:
   a) Bob selects distinct primes p and q.
   b) Bob computes $N = pq$.
   c) Bob computes $\phi(N) = (p-1)(q-1)$.
   d) Bob selects e such that $0 < e < \phi(N)$, $\gcd(e, \phi(n)) = 1$.
   e) Bob computes d such that $ed \equiv 1 \pmod{\phi(N)}$.
   f) Bob publishes (N, e) (the public key) but keeps (N, d) (the private key) to himself.
2. Encryption: Alice obtains the publicly available (N, e) (the public key) and computes
$$E_{(N,e)}(x) = x^e \pmod N$$
and sends it to Bob.
3. Decryption: Bob uses (N, d) (the private key) to compute
$$D_{(N,d)}(x^e \pmod N) = x^{ed} \pmod N$$

Note that the key is made up of

- a **public key** (N, e) and — public key
- a **private key** (N, d) — private key

(N, e) is revealed to the public. (N, d) is kept private. In general

Definition 3.1.1. A **public cipher** is made up of the encryption and decryption functions $E_{\text{pubkey}}, D_{\text{privkey}}$ which depends on the key $k = (\text{pubkey}, \text{privkey})$ which is a 2-tuple made up of the public key and private key. Such a cipher is also called an **asymmetric key cipher**.

public cipher

asymmetric key cipher

Recall that in the case of private (or symmetric) key cipher the encryption and decryption keys are the same.

Exercise 3.1.1. Eve saw Alice sent the ciphertext 230539333248 to Bob. Eve checks Bob's website and found out that his public RSA key is $(N, e) = (100000016300000148701, 7)$. Help Eve compute the plaintext. In fact, compute the private key (N, d). You only need to compute d since N is known. How much time did you use? (Hint: Why is factoring N crucial?) □

File: rsa-implementation-issues.tex

3.2 Implementation issues

For RSA to be a good cryptosystem, the operations involved (key generation, encryption, decryption) must be fast. Furthermore it must be able to sustain all types of attacks. In this section we'll talk about algorithms and performance issues.

File: baby-asymptotic-analysis.tex

3.3 Baby Asymptotic Analysis

The mathematical technology used to measure the speed of an algorithm is called asymptotic analysis. I'm not going to explain everything in asymptotic analysis. I'll only give you enough to move forward. (By the way this area of math was created by people in the area of number theory.)

Let's look at a simple problem first. Suppose you're given two 3-digit numbers to add, say you want to do $123 + 234$. This is what you would do:

```
    1 2 3
+   3 6 9
---------
    4 9 2
---------
```

The mount of work done involves reading two digits for each column, computing the digit sum, which gives two numbers (remember you need to compute the carry too, right?) the sum mod 10 and sum / 10. For instance for the first column (the leftmost one), you do

```
 3 +  9 = 12
12 % 10 = 2
12 / 10 = 1
```

Once that's done you write down:

```
      1
    1 2 3
+   3 6 9
---------
      2
---------
```

Correct? You then go on to the second column and do this:

```
1 + 2 + 6 = 9
9 % 10 = 9
```

```
9 / 10 = 0
```

and you write this:

```
    0 1
    1 2 3
+   3 6 9
---------
      9 2
---------
```

and so on.

Note that what you do is the same for each column. The first column is kind of different because you don't have to worry about any carry at all. However to make the first column like the rest, you can create an initial carry of 0 too:

```
        0
    1 2 3
+   3 6 9
---------

---------
```

Why would you do that? Well ... it make the algorithm more uniform. But in any case for each column, you basically perform the following work:

```
0 + 3 + 9 = 12
12 % 10 = 1
12 / 10 = 1
```

Of course you also have to read the digits (think of this as work done reading the piece of paper), write the digits on the piece of paper. Suppose it takes times t_1 to do all that for each column.

There are 3 columns. Therefore when you're done with all the column operations, you would have used up this amount of time:

$$3 \cdot t_1$$

There was actually one step you did by putting a zero as an initial carry:

```
          0
      1 2 3
+     3 6 9
---------

---------
```

Let's say this takes time t_0. So the total time is

$$3 \cdot t_1 + t_0$$

Don't forget that once you're done with the 3 columns, you have a carry beyond the 3rd column:

```
    0 0 1 0
      1 2 3
+     3 6 9
---------
      4 9 2
---------
```

You can put the 0 down on the fourth column:

```
    0 0 1 0
      1 2 3
+     3 6 9
---------
    0 4 9 2
---------
```

or just leave it blank. Suppose the time to put it down or not put it down is t_2. So the total time is

$$3 \cdot t_1 + t_0 + t_2$$

It should be clear that if you have two n–digit numbers, the time needed is

$$n \cdot t_1 + t_0 + t_2$$

Now someone who writes faster, reads faster, compute addition or quotient or

mod 10 faster might do it in time

$$n \cdot t'_1 + t'_0 + t'_2$$

However the n doesn't go away. In the long run, it's n can controls the growth of this function. If someone were to invent a different addition algorithm that runs with this time:

$$\frac{1}{2}n \cdot t''_1 + t''_0 + t''_2 = n \cdot \frac{t''_1}{2} + t''_0 + t''_2$$

then it's really the same as the previous one. Why? Because all you need to do is to hire someone who can read and write and perform digit addition, digit mod 10, digit / 10 extremely fast so that you t_1 is much smaller than the $\frac{t''_1}{2}$ and $t_0 + t_2$ much smaller than his $t''_0 + t''_2$.

Not only that ... if n is really huge, it's clear that the $n \cdot t_1$ is going to overcome the $t_0 + t_2$ part.

And if we are concerned about the speed of our algorithm, obviously we worry about the case when n (the number of digits) is huge. After all ... who cares that much about addition of 3 digits? What we should worry about is what happens when we apply our method to the addition of two 1000000-digit numbers. If you look at for instance

$$n^2 + 1000000n$$

and

$$n^2$$

when n is small (say 3), the $1000000n$ is huge. But if you think about $n = 10^{1000}$, you see that $n^2 + 1000000n$ and n^2 are actually very close. (Take out your graphing calculator and try zooming out the graphs of $y = x^2$ and $y = x^2 + 1000000x$ to check that I'm not lying.)

This tells us that really the function to focus on when measuring algorithm runtime performance is actually

$$n$$

and not

$$n \cdot t_1 + t_0 + t_2$$

That's the whole point of asymtotic analysis. (Well ... there's a lot more ... but that's enough for *us* ...)

To say that we're ignoring the constants and focusing on the part that controls the growth of the function

$$n \cdot t_1 + t_0 + t_2$$

we write

$$n \cdot t_1 + t_0 + t_2 = O(n)$$

That's called the "big-O of n".

Let's look at what's called "high school multiplication algorithm". Suppose you're given two 3-digit numbers to multiply. Say 123 and 234. You would do this:

```
      1 2 3
  x   2 3 4
  ---------
      4 9 2
    3 6 9
+ 2 4 6
-----------
  2 8 7 8 2
-----------
```

It's easy to see that there are 9 digits to compute in the "mid-section" of the computation:

```
      1 2 3
  x   2 3 4
  ---------
      4 9 2
    3 6 9
+ 2 4 6
-----------

-----------
```

Then the 3 rows are added up.

Using this method, how much time does it take to compute the product of two n–digit integers? There are $n \times n$ numbers to compute in the mid-section. So the midsection requires

$$O(n^2)$$

Note that the numbers you get contain integers of length about $2n$. There

are n such numbers. Adding two $2n$-digit numbers takes $O(2n)$ time. But remember that we ignore constants. So adding two of the $2n$-digit numbers take

$$O(n)$$

times. Given n such numbers, you need to perform $n-1$ additions. So altogether the time is

$$O((n-1)n)$$

Now look at the function

$$(n-1)n$$

This is

$$(n-1)n = n^2 - n$$

The worse part of the function (the part that grows the fastest) is n^2. Therefore

$$O((n-1)n) = O(n^2)$$

Now the computation of the midsection takes $O(n^2)$ and the addition part takes $O(n^2)$. Together it would take $O(2n^2)$. But again we ignore constants. So the time taken is $O(2n^2) = O(n^2)$.

Hence the worse case runtime performance of the "high school multiplication algorithm" is $O(n^2)$.

This standard multiplication algorithm has been used for a very very very very long time.

Can we do better?

Before we leave this section, note that I've already said that addition runs in $O(n)$. You can't do better than that. Why? Well ... you have to at least read the two n–digit numbers, right? Reading them takes about $n+n$ time, which is $2n$, which is $O(n)$. So there's no way you can beat that. Hey ... you have to at least read what to add, right?!?!

File: karatsuba.tex

3.4 Multiplication: Karatsuba algorithm

Skip this section if you have already taken CISS358.

It turns out that there is a better way to multiply integers. This was only discovered very recently in the 1960

The idea is surprisingly simple. Here's a basic formula:

$$(aT+b)(cT+d) = acT^2 + (ad+bc)T + bd$$

For now you can pretend that $T = 10$ and a, b, c, d are digits. For instance to multiply 23 and 45, you can view it as

$$(2 \cdot 10 + 3)(4 \cdot 10 + 5) = (2 \cdot 4)10^2 + (2 \cdot 5 + 3 \cdot 4)10 + 3 \cdot 5$$

You can finish up this computation on your own.

But

$$(aT+b)(cT+d) = acT^2 + (ad+bc)T + bd$$

doesn't really help!!! Viewing $T = 10$ and the a, b, c, d as digits, the above multiplication of two 2-digit numbers, i.e. $(aT+b)$ and $(cT+d)$, we need *four* multiplications

$$ac, \quad ad, \quad bc, \quad bd$$

So it's still essentially an n^2 algorithm. In general if $T(n)$ is runtime of multiplying 2 integers of length n, then

$$T(n) = 4T(n/2) + An + B$$

If $n = 2^k$, then

$$\begin{aligned}
T(2^k) &= 4T(2^{k-1}) + A2^k + B \\
&= 4(4T(2^{k-2}) + A2^{k-1} + B) + A2^k + B \\
&= 4^2T(2^{k-2}) + (4 \cdot 2^{k-1} + 2^k)A + (4+1)B \\
&= 4^2(4T(2^{k-3}) + A2^{k-2} + B) + (4 \cdot 2^{k-1} + 2^k)A + (4+1)B \\
&= 4^3T(2^{k-3}) + (4^2 \cdot 2^{k-2} + 4 \cdot 2^{k-1} + 2^k)A + (4^2 + 4 + 1)B \\
&= ... \\
&= 4^kT(2^{k-k}) + (4^{k-1} \cdot 2^1 + ... + 4^2 \cdot 2^{k-2} + 4 \cdot 2^{k-1} + 2^k)A + (4^{k-1} + ... + 4^2 + 4 + 1)B
\end{aligned}$$

The coefficient for B is $\frac{4^k-1}{4-1}$. The coefficient for A is

$$4^{k-1}\cdot 2^1+...+4^2\cdot 2^{k-2}+4\cdot 2^{k-1}+2^k = 2^{2k-1}+2^{k+2}+2^{k+1}+2^k = 2^k(1+2+...+2^{k-1}) = 2^k(2^k-1)$$

Hence

$$T(2^k) = 4^k T(1) + 2^k(2^k-1)A + \frac{4^k-1}{3}B$$

Hence $T(1) = C$:

$$T(2^k) = 4^k C + 2^k(2^k-1)A + \frac{4^k-1}{3}B$$

Since $n = 2^k$,

$$T(n) = n^2 C + n(n-1)A + \frac{n^2-1}{3}B$$

Clearly $T(n) = O(n^2)$.

But wait ... here's the brilliant (but simple) idea from Karatsuba. Note that

$$\begin{aligned} & (a+b)(c+d) = ac+ad+bc+bd \\ \therefore \quad & (a+b)(c+d) - ac - bd = ad+bc \end{aligned}$$

In other words

$$(aT+b)(cT+d) = (ac)T^2 + [(a+b)(c+d) - (ac) - (bd)]T + (bd)$$

If you count all the operations on the right, you would see that there are only *three* multiplications!!!

So we can do this:

```
1. Compute A = ac          ... 1 multiplication
2. Compute B = bd          ... 1 multiplication
3. Compute C = (a+b)(c+d) ... 2 additions, 1 multiplication
4. Compute D = C - A - B   ... 2 subtractions
5. Output AT^2 + DT + B
```

Note that the runtime for subtraction is like addition. In other words addition and subtraction are "cheap": they are both $O(n)$ where n is the length of the integers.

Practically speaking, how is Karatsuba actually used? Suppose you have to

multiply two 8-digit numbers:

$$12345678 \times 13572468$$

We split them up into this (with $T = 10$)

$$(1234T^4 + 5678) \times (1357T^4 + 2468)$$

by Karatsuba

$$\begin{aligned}(aT^4 + b) \times (cT^4 + d) &= A(T^4)^2 + D(T^4) + B \\ A &= ab \\ B &= bd \\ C &= (a + b)(c + d) \\ D &= C - A - B\end{aligned}$$

Next, we *again* apply Karasuba but this time to the products $ab, bd, (a + b)(c + d)$. Etc. In other words you recursively use Karatsuba until the products involve numbers which are small enough that we can perform them quickly without Karatsuba, If we starting with the multiplication of two 8-digit numbers. We're left with three multiplications of 4-digit numbers. On applying Karasuba to the three multiplications of 4-digits numbers, each multiplication gives rise to 3 2-digit numbers. There are now 3×3 multiplications of 2-digit numbers. Going further, we get $3 \times 3 \times 3$ multiplication of 1-digit numbers.

Here's an example. Suppose we want to multiply

$$1234 \times 1357$$

Step 1: $(1234)(1357)$.

$$\begin{aligned}(1234)(1357) &= (12T^2 + 34) \times (13T^2 + 57) \\ &= A(T^2)^2 + DT^2 + B\end{aligned}$$

$(T = 10)$ where

$$\begin{aligned}a = 12, \;\; b = 34, \;\; c = 13, \;\; d = 57 \\ A = ac = (12)(13) \\ B = bd = (34)(57) \\ C = (a + b)(c + d) = (46)(70) \\ D = C - A - B\end{aligned}$$

Step 2: (12)(34).

$$\begin{aligned}(12)(13) &= (1T^1 + 2) \times (1T^1 + 3) \\ &= A(T^1)^2 + DT^1 + B\end{aligned}$$

where

$$\begin{aligned}&a = 1, \;\; b = 2, \;\; c = 1, \;\; d = 3 \\ &A = ac = (1)(1) = 1 \\ &B = bd = (2)(3) = 6 \\ &C = (a + b)(c + d) = (3)(4) = 12 \\ &D = C - A - B = 12 - 1 - 6 = 5\end{aligned}$$

So

$$(12)(34) = 1(T^1)^2 + 5T^1 + 6 = 100 + 50 + 6 = 156$$

Step 3: (34)(57)

$$(34)(57) = (3T^1 + 4) \times (5T^1 + 7) = A(T^1)^2 + DT^1 + B$$

where

$$\begin{aligned}&a = 3, \;\; b = 4, \;\; c = 5, d = 7 \\ &A = ac = (3)(5) = 15 \\ &B = bd = (4)(7) = 28 \\ &C = (a + b)(c + d) = (7)(12) = 84 \\ &D = C - A - B = 84 - 15 - 28 = 41\end{aligned}$$

So

$$(34)(57) = 15(T^1)^2 + 41T^1 + 28 = 1500 + 410 + 28 = 1938$$

Step 4: (46)(70).

$$(46)(70) = (4T^1 + 6) \times (7T^1 + 0) = A(T^1)^2 + DT^1 + B$$

where

$$\begin{aligned}
&a = 4,\ \ b = 6,\ \ c = 7,\ \ d = 0\\
&A = ac = (4)(7) = 28\\
&B = bd = (6)(0) = 0\\
&C = (a+b)(c+d) = (10)(7) = 70\\
&D = C - A - B = 70 - 28 - 0 = 42
\end{aligned}$$

So

$$(46)(70) = 28(T^1)^2 + 42T^1 + 0 = 2800 + 420 + 0 = 3220$$

Putting Steps 2,3,4 back into Step 1 ...

Continuing Step 1: (1234)(1357).

$$\begin{aligned}
(1234)(1357) &= (12T^2 + 34) \times (13T^2 + 57)\\
&= A(T^2)^2 + DT^2 + B
\end{aligned}$$

where

$$\begin{aligned}
&a = 12,\ \ b = 34,\ \ c = 13,\ \ d = 57\\
&A = (12)(13) = 156 && \text{from Step 1}\\
&B = (34)(57) = 1938 && \text{from Step 2}\\
&C = (46)(70) = 3220 && \text{from Step 3}\\
&D = C - A - B = 3220 - 156 - 1938 = 1126
\end{aligned}$$

and

$$\begin{aligned}
(1234)(1357) &= A(T^2)^2 + DT^2 + B\\
&= 156(T^2)^2 + 1126T^2 + 1938\\
&= 1560000 + 112600 + 1938\\
&= 1674538
\end{aligned}$$

Now if you go back and look for the multiplications you see these:

$$\begin{aligned}(1)(1) &= 1\\(2)(3) &= 6\\(3)(4) &= 12\\(3)(5) &= 15\\(4)(7) &= 28\\(7)(12) &= 84\\(4)(7) &= 28\\(6)(0) &= 0\\(10)(7) &= 70\end{aligned}$$

i.e., $9 = 3^2$ multiplications if we multiply two integers of length $n = 2^2$. High-school multiplication would require $n^2 = (2^2)^2 = 16$ multiplications.

Now let us compute the runtime of Karatsuba's algorithm:

Proposition 3.4.1. *The runtime of Karatsuba is $O(n^{\lg 3}) = O(n^{1.5849\ldots})$ where n is the length of the integers to be multiplied.*

Proof. TODO □

The above result is the same whether you view your integer as an array of decimals or as an array of bits.

It's a good exercise to implement Karatsuba on your own. I did some number theory crunching programming during the summer of my freshman year and a prof gave me a copy of the original paper published by Karatsuba. (Karatsuba's paper has since inspired several improvements.) As you see from the above, the amount of math that you need to know is pretty minimal. Basically the ingredients are

$$(aT + b)(cT + d) = acT^2 + (ad + bc)T + bd$$

(which is not new) and this (which is new):

$$\begin{aligned}(a + b)(c + d) &= ac + ad + bc + bd\\\therefore \quad (a + b)(c + d) - ac - bd &= ad + bc\end{aligned}$$

There are various obvious optimizations which only tweaks the constants of your big-O, but not the big-O itself. For instance if you look at this addition:

$$\begin{aligned}(1234)(1357) &= A(T^2)^2 + DT^2 + B \\ &= 156(T^2)^2 + 1126T^2 + 1938 \\ &= 1560000 + 112600 + 1938 \\ &= 1674538\end{aligned}$$

you'll see that you are adding 1560000 with another number 112600. There are some zeroes in 112600. So you might want to have a function that adds say 1126 to an integer starting at the 100s digit position. This will speed up for instance adding to an integer 123456789123456789, the number 123000000000000:

```
  123456789123456789
+    123000000000000
--------------------

--------------------
        ^
        |
        start column addition here
```

Another thing to note is that our computers are mostly 32 bit machines. If you're using an array of integers to model long integers, you only want to cut up the integers up to a certain point, and not to the digit-level. For instance you might want to model your long integers with arrays of 16-bit integers. Note that the process of cutting up the integers into pieces takes time. In fact when the number of digits is small, highschool multiplication might be faster. Therefore you want to write your code in such as way that if the length of the integers is less than a certain constant, then highschool multiplication is used; if the length is greater than this constant, then Karatsuba is used. That's what I did.

(The language I used long long long time ago to do Karatsuba was Pascal. Each element of the array models 0 to 999. I found through timed testing that if the array has length ≤ 5, then highschool multiplication was actually faster. This is of course machine specific. This explains why when you install some number crunching libraries, before the installation is completed, it will tests the code to tweak such constants for maximum performance.)

OK. Enough hints. You should go ahead and write a long integer package that

incorporates Karatsuba for multiplication.

Exercise 3.4.1. Compute $1122334455667788 \times 8765432187654321$ using Karasuba multiplication to continually breakdown the integers up to integers of length 2; use your calculator to perform multiplication of length 2 integers.

Exercise 3.4.2. Implement a long integer class where multiplication uses Karasuba.

Exercise 3.4.3. The algorithmic analysis above is basically correct but some details are actually missing. Write a research paper on Karatsuba, describing the algorithm in detail, and analyze the runtime performance exactly. Research also on efficient implementation of Karatsuba.

Exercise 3.4.4. * Since the surprising (shocking?) discovery of Karatsuba, several improvements to his algorithm has appeared since the 60s. Write a research paper on the various new-fangled integer multiplication algorithms, analyze and comparison their runtime performance.

File: exponentials-squaring-method.tex

3.5 Exponentiation: the squaring method

See CISS358. Skip this section if you have taken CISS358.

Note that for RSA we need to compute powers. Extremely large powers. The obvious method: If $n > 0$,

$$a^n = a^{n-1} \cdot a$$

has a runtime of $O(n)$ (in fact $\Theta(n)$) of multiplications. If the exponentiation is in mod N, then you take mod N after each multiplication to previous the a^k from becoming too huge, i.e.,

$$a^k = a^{k-1} \cdot a \pmod{N}$$

There's a much faster way to compute exponentiation.

First I will describe it using recursion:

$$a^n = \begin{cases} 1 & n = 0 \\ \left(a^{n/2}\right)^2 & n > 0 \text{ and } n \text{ is even} \\ a \cdot \left(a^{(n-1)/2}\right)^2 & n > 0 \text{ and } n \text{ is odd} \end{cases}$$

Here, $n \geq 0$.

Example 3.5.1. Here's an example to compute 2^{27}

1. $2^{27} = 2 \cdot (2^{13} \cdot 2^{13})$
2. $2^{13} = 2 \cdot (2^6 \cdot 2^6)$
3. $2^6 = (2^3 \cdot 2^3)$
4. $2^3 = 2 \cdot (2^1 \cdot 2^1)$
5. $2^1 = 2 \cdot (2^0 \cdot 2^0) = 2 \cdot (1 \cdot 1)$

It's easy to write an algorithm implementing the above:

```
ALGORITHM: power
INPUTS: a, n where n >= 0

if n == 0:
    return 1
```

```
else:
    if n is even:
        x = power(a, n / 2)
        return x * x
    else:
        x = power(a, (n - 1) / 2)
        return a * x * x
```

The number of recursions is $O(\lg n)$ where the main cost of each recursion step is either one or two multiplications.

This above recursion can be rewritten using a loop. Suppose you want to compute a^x. First you write x as a binary number:

$$x = (x_k \cdots x_0)_2 = x_k 2^k + \ldots + x_1 2^2 + x_0 2^0$$

where each x_i is 0 or 1. As an example suppose we look at

$$x = 27 = (11011)_2 = 1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0$$

Then

$$a^{27} \equiv a^{1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^3 + 1 \cdot 2^1 + 1 \cdot 2^0} \equiv a^{2^4} a^{2^3} a^{2^1} a^{2^0}$$

Notice that the computation of a^{27} on the right depends on $a^{2^0}, a^{2^1}, a^{2^2}, \ldots$ In general $a^{2^{i+1}} = (a^{2^i})^2$. In the following b runs through $a^{2^0}, a^{2^1}, a^{2^2}, \ldots$. Notice that a^{2^i} is included in a^{27} if the i–th bit of x is 1. Instead of pre-computing the bits of x, we can compute the i–th bit iteratively as the exponentiation is computed. In the following p runs through the partial products of $a^{2^0}, a^{2^1}, a^{2^2}, \ldots$, picking up a^{2^i} if the i–th bit of n is 1. In this case, p and b will run through the following values:

$$\begin{aligned}
p &= 1 & b &= a = a^{2^0} \\
p &= p \cdot b = a^{2^0} & b &= b \cdot b = a^{2^1} \\
p &= p \cdot b = a^{2^0} \cdot a^{2^1} & b &= b \cdot b = a^{2^2} \\
& & b &= b \cdot b = a^{2^3} \\
p &= p \cdot b = a^{2^0} \cdot a^{2^1} \cdot a^{2^3} & b &= b \cdot b = a^{2^4} \\
p &= p \cdot b = a^{2^0} \cdot a^{2^1} \cdot a^{2^3} \cdot a^{2^4} & b &= b \cdot b = a^{2^5}
\end{aligned}$$

Here's the algorithm:

```
ALGORITHM: power
INPUTS: a, n where n >= 0
OUTPUT: a^n
```

```
p = 1
b = a

while n is not 0:
    bit = n % 2
    n = n / 2 (integer division)
    if bit == 1:
        p = p * b
    b = b * b

return p
```

And if the exponentiation is done in mod N, then we just mod by N as frequently as we can:

```
ALGORITHM: power-mod
INPUTS: a, n, N where n >= 0 and N is the modulus
OUTPUT: (a^n) % N

p = 1
b = a % N
while n is not 0:
    if n % 2 == 1:
        p = (p * b) % N
    n = n // 2
    b = (b * b) % N
return p
```

The number of iterations is the number of bits of n, i.e., the number of iterations is $\lfloor \lg n + 1 \rfloor$ ($\lg = \log_2$). Each iteration in the worse case scenario involves two mod N multiplications. Hence the runtime time is

$$O((\lg n) \cdot M)$$

where M is the runtime for multiplication of two integers in $\mathbb{Z}/N$. (In the case when we are performing exponentiation without mod N, the length of the integer `p` in the above increases. In that case the factor M for the runtime of multiplication increases.)

To include the case of negative exponent, when a is a real number

$$a^{-n} = (a^{-1})^n$$

and for mod N, if a is invertible,

$$a^{-n} = (a^{-1})^n \pmod{N}$$

Exercise 3.5.1. Leetcode 50.
https://leetcode.com/problems/powx-n/
Implement `pow(x, n)`, which calculates `x` raised to the power `n`.

Exercise 3.5.2. * In the above, the computation of a^x depends on writing x is base 2. What if you write x in base 3? □

Exercise 3.5.3. Implement an exponentiation function using the squaring method. Test it. After you are done, implement an exponentiation function in $\mathbb{Z}/N$ using the squaring method. □

File: inverse-in-modular-arithmetic.tex

3.6 Inverse in modulo arithmetic

In the key generation for RSA, Bob has to compute the multiplicative inverse of e mod $\phi(n)$. This is just the Extended Euclidean Algorithm. (See previous notes).

File: primality-test.tex

3.7 The Prime Number Theorem and finding primes

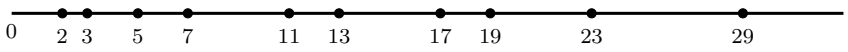
We'll need a way to create huge primes for RSA.

Of course you know Euclid's theorem that says there are infinitely many primes. So we don't have to worry about not finding them. But just because there are infinitely many primes, it does not mean they are everywhere!

Generally, we want to specify how large our primes should be. This is specified by the bit length of the primes. Given the length, one can generate a sequence of bit of that length. Of course that number should be odd. So the least significant bit is set to 1. Call this n. One can than check if n is a prime. If n is not prime, we try $n + 2$, etc.

Does this process take a long time?

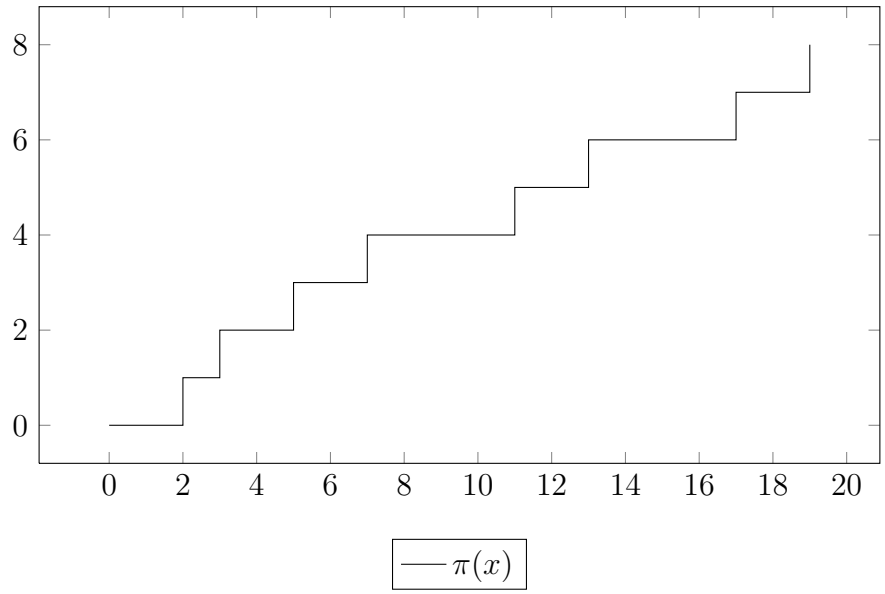
Gauss was the first to realize that even though primes seem to appear randomly on the real line,



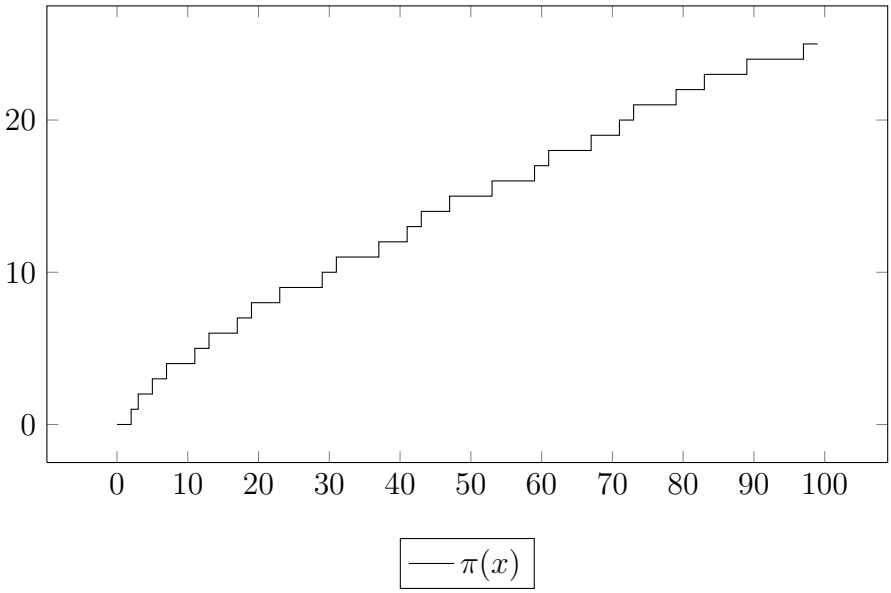
the density of primes seems to follow some law. The density of primes can be defined this way. Let $\pi(x)$ be the number of primes $\leq x$. Then the density of primes up to x is

$$\frac{\pi(x)}{x}$$

Here is the plot of $\pi(x)$ up to $x = 20$:



When the plot is up to $x = 100$ one begin to see that the rough and jagged graph begin to smooth out:



Through analyzing tables of primes, Gauss discovered that $\pi(x)/x$ is **asymptotically equivalent** to $\ln x$ ($\ln = \log_e$)

asymptotically equivalent

$$\frac{\pi(x)}{x} \sim \frac{1}{\ln x}$$

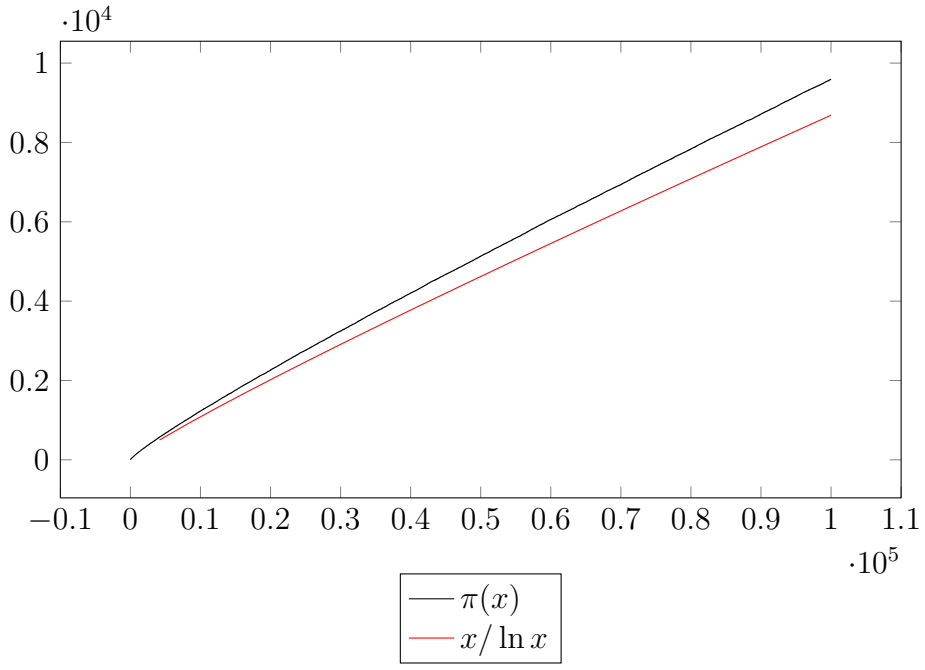
i.e.,

$$\lim_{x\to\infty} \frac{\pi(x)/x}{1/\ln x} = 1$$

Equivalently

$$\pi(x) \sim \frac{x}{\ln x}$$

Here a plot of $\pi(x)$ and $x/\ln x$ up to $x = 10^5$:



The above was first conjectured by Gauss in 1782/3 and finally proven in 1896 by Hadamard and de la Vallée Poussin:

Theorem 3.7.1. (**Prime Number Theorem**)

Prime Number Theorem

$$\pi(x) \sim \frac{x}{\ln x}$$

Among number theorists and researchers in cryptography, the above deep result is known as **PNT**.

PNT

By PNT, when x is large, the density of primes up to x is approximately $1/\ln x$. Using the sieve of Eratosthenes, the number of primes up to $x = 10^5$ is 9592, i.e., $\pi(x)/x = 9592/100000 = 0.09592 = 9.592\%$ which is very close

to $1/\ln x = 1/\ln 10^5 = 0.086858... = 8.6858...\%$. If we search for a prime only among *odd* integers, the chance of finding a prime is $2/\ln x = 0.173717... = 17.3717...\%$.

For modern-day RSA, primes used have approximately 1024-2048 bits. If we choose a bit length of 1024, then $2/\ln 2^{1024} = 0.1408...\%$. Therefore one might find a prime after < 1000 tries among odd integers. Usually one would begin with an integer n with a random sequence of 1024 bits, with least significant bit being 1 (so that n is odd). Then a primality test is used to check if n is prime. We'll see that a probabilistic primality test is used. If n is not a prime, one would then try $n + 2$. Etc.

Next we will look at two very important primality tests: Fermat primality test and Miller–Rabin primality test. Miller–Rabin primality test is the one that is used in the real world. However the main idea in Miller–Rabin primality test is actually Fermat primality test.

Exercise 3.7.1. Write a function `rand_odd_int` that accept L and return an odd positive integer with L random bits. Hint: For python, try this:

```
n = int("0b111", 2) # the "0b" is optional
print(n)
n = int("0b100", 2)
print(n)
```

Try a few more examples to understand what is happening.

Exercise 3.7.2. Write a function `eratosthenes` that accepts an integer n and returns a bool array `isprime` of size n such that `isprime[i]` is True iff `i` is prime.

Exercise 3.7.3. Write a function `primes` that accepts x and returns an array of primes from 2 up to x (inclusive) in ascending order. For instance `primes(10)` return `[2, 3, 5, 7]`.

Exercise 3.7.4. Write a function `write_primes` that accepts `x` and a path `p` and store primes up to `x` at path `p` in comma-separated format. For instance `write_primes(10, 'primes-10.txt')` writes `"2,3,5,7"` to the file `primes-10.txt`. Write another function `read_primes` that accepts a path `p`

and returns a list of primes stored at path `p`. Create a file of primes up to 10,000,000. After you are done with the above make a slight optimization by storing integer in hex. While a decimal (base-10 digit) can store 10 patterns, a hexadecimal can store 16. Try this:

```
i = int("0x1a", 16) # the "0x" is optional
print(i)
s = hex(i)
print(s)
```

(It's even better to store the integer directly in binary format, but that makes the file non-human readable.)

Exercise 3.7.5. Write a function `pi` that accept `x` and returns the number of primes up to `x` (inclusive).

File: fermat-primality-test.tex

3.8 Fermat primality test

Recall that Fermat's Little Theorem says that

$$p \text{ prime}, \ p \nmid a \implies a^{p-1} \equiv 1 \pmod{p}$$

I can also say that if $1 \leq a \leq p-1$,

$$p \text{ prime} \implies a^{p-1} \equiv 1 \pmod{p}$$

And of course we have: if $1 \leq a \leq p-1$,

$$p \text{ not prime} \impliedby a^{p-1} \not\equiv 1 \pmod{p}$$

Replacing "p" by "n", we have the following:

Fermat compositeness test

Proposition 3.8.1. (**Fermat compositeness test**) *If there is some a such that $1 \leq a \leq n-1$ and*

$$a^{n-1} \not\equiv 1 \pmod{n}$$

then n is composite.

Although the proposition above is true in general, when used in an algorithm, there are several cases that we want to remove.

1. Obviously $a = 1$ does not satisfy the hypothesis of the proposition. Therefore the interval for a should be $[2, n-1]$ instead of $[1, n-1]$.
2. Also, the hypothesis does not hold if $a = n-1$ and n is odd. For the case when n is even, either $n = 2$ or $n = 2k$ where $k > 1$. If the goal is to show n is composite, the case of $n = 2$ and $n = 2k$ where $k > 1$ can be checked quickly. Therefore the only useful scenario to use the proposition if now when n is odd and $a \in [2, n-2]$.
3. For the condition "$2 \leq a \leq n-2$" to have any a at at all, we need $2 \leq n-2$, i.e., $n \geq 4$.

Altogether the proposition is useful in an algorithm when $n > 3$ is odd and when $a \in [2, n-2]$.

```
ALGORITHM: Fermat-compositeness-test
INPUTS: n -- number to be tested for compositeness
        t -- number of tries
OUTPUT: "n is composite" if n is composite

if n <= 2: return "n is not composite"
if n % 2 is 0 and n / 2 > 1: return "n is composite"

for i = 1, 2, 3, ..., t:
    pick random a in [2, n - 2]
    if a^(n - 1) ≢ 1 (mod n) return "n is composite"

return "no conclusion"
```

Note that if you return `"no conclusion"`, it means either n or prime or n is composite but unfortunately your random a did not pick an a satisfying $a^{n-1} \not\equiv 1 \pmod{n}$.

Definition 3.8.1. Suppose n is composite and let $a \in [1, n-1]$. If a satisfies

$$a^{n-1} \not\equiv 1 \pmod{n}$$

then a is called a **Fermat witness** for the compositeness of n. Otherwise, if

Fermat witness

$$a^{n-1} \equiv 1 \pmod{n}$$

then a is called a **Fermat liar**. The curious reason for calling such an a a liar will be clarified later.

Fermat liar

Note that the above test tells you that n is composite *without* factoring n. Let's compare the above Fermat compositeness test to other compositeness test.

- Division compositeness test: Randomly pick a such that $2 \leq a \leq \lfloor \sqrt{n} \rfloor$. If a divides n, then n is a composite. If the chance of finding a is low, then one would have to run a through $[2, \lfloor \sqrt{n} \rfloor]$. In general, iteration rather than randomization is used.
- GCD compositeness test: Randomly pick a such that $2 \leq a \leq \lfloor \sqrt{n} \rfloor$, if $\gcd(a, n) > 1$, then n is composite. If the chance of finding a is low, then one would have to run a through $[2, \lfloor \sqrt{n} \rfloor]$.

Note that in both of these tests, a divisor of n is found. The speed of the above three compositeness tests depends on how fast we can find some a satisfying

some condition in the test.

To better understand the condition

$$a^{n-1} \equiv 1 \pmod{n}$$

where $a \in [2, n-2]$, here's a table:

```
 3 1  None 1 ...
 4 0   0.0 0 ..0.
 5 1   1.0 1 ..11.
 6 0   0.0 0 ..000.
 7 1   1.0 1 ..1111.
 8 0   0.0 0 ..00000.
 9 0   0.0 0 ..000000.
10 0   0.0 0 ..0000000.
11 1   1.0 1 ..11111111.
12 0   0.0 0 ..000000000.
13 1   1.0 1 ..1111111111.
14 0   0.0 0 ..00000000000.
15 0  0.17 0 ..001000000100.
16 0   0.0 0 ..0000000000000.
17 1   1.0 1 ..11111111111111.
18 0   0.0 0 ..000000000000000.
19 1   1.0 1 ..1111111111111111.
20 0   0.0 0 ..00000000000000000.
21 0  0.11 0 ..000000100001000000.
22 0   0.0 0 ..000000000000000000.
23 1   1.0 1 ..11111111111111111111.
24 0   0.0 0 ..00000000000000000000.
25 0  0.09 0 ..000001000000000100000.
26 0   0.0 0 ..000000000000000000000.
27 0   0.0 0 ..000000000000000000000000.
28 0  0.08 0 ..000000010000000000000010.
29 1   1.0 1 ..111111111111111111111111.
30 0   0.0 0 ..000000000000000000000000.
31 1   1.0 1 ..111111111111111111111111.
32 0   0.0 0 ..000000000000000000000000.
33 0  0.07 0 ..000000001000000000000100000000.
34 0   0.0 0 ..000000000000000000000000000000.
35 0  0.06 0 ..000010000000000000000000010000.
36 0   0.0 0 ..000000000000000000000000000000.
37 1   1.0 1 ..111111111111111111111111111111.
38 0   0.0 0 ..000000000000000000000000000000.
39 0  0.06 0 ..000000000001000000000010000000000.
40 0   0.0 0 ..000000000000000000000000000000000.
41 1   1.0 1 ..111111111111111111111111111111111.
42 0   0.0 0 ..000000000000000000000000000000000000.
43 1   1.0 1 ..111111111111111111111111111111111111.
44 0   0.0 0 ..000000000000000000000000000000000000000.
```

```
45 0  0.14 0 ..000000100000000101000000101000000001000000.
46 0   0.0 0 ..000000000000000000000000000000000000000000.
47 1   1.0 1 ..111111111111111111111111111111111111111111.
48 0   0.0 0 ..000000000000000000000000000000000000000000.
49 0  0.09 0 ..000000000000000011000000000011000000000000000.
50 0   0.0 0 ..000000000000000000000000000000000000000000000.
```

Here are the descriptions of the columns:

- First column: n
- Third column: 1–prime, 0–composite
- Fifth column: sequence of boolean values for $a^{n-1} \equiv 1 \pmod{n}$ where $a = 0, 1, 2, ..., n-1$, except that if a is not in $[2, n-2]$, the value is "." to indicate "not applicable". The "not applicable" values are $a = 0, 1, n-1$.
- Second column: The percentage of "1"s in the fourth column (not counting the "not applicable" cases).
- Third column: 1–percentage of witnesses is 0%, 0–otherwise.
- Fourth column: 1–if values in fifth column are all 1. (Ignore this for now.)

Of course when n is prime, the sequence of boolean values are all "1"s. Now let us focus on the composite n's.

For $n = 15$:

```
15 0  0.17 0 ..001000000100.
```

there are 10 Fermat witnesses for the compositeness of 15 (i.e., 2, 3, 5, 6, 7, 8, 9, 10, 12, 13) and 2 Fermat liars (i.e., 4, 11). For instance 2 is a Fermat witness since

$$2^{15-1} = 2^{14} = (2^4)^3 \cdot 2^2 = (16)^3 \cdot 4 \equiv 1^3 \cdot 4 = 4 \not\equiv 1 \pmod{16}$$

and 4 is a Fermat liar since

$$4^{15-1} = 4^{14} = (4^2)^7 = (16)^7 \equiv 1^7 = 1 \pmod{16}$$

The percentage of Fermat liars (among integers $a \in [2, 13]$) is 17%. Therefore there are more Fermat witnesses than Fermat liars. In fact, when n is composite, the fact that there are more Fermat witnesses than not seems to be very common. The first time the percentage of Fermat liars is $> 50\%$ is when $n = 561$ (at 57%).

This means that when n is composite and we randomly pick an a such that

$a \in [2, n-2]$, the chance that a is a Fermat witness to n's compositeness seems be to very high. The percentage of Fermat witnesses will be quantified later.

Exercise 3.8.1. Write a program that produces the above table, say up to $n = 10000$. □

Compared to the brute force division compositeness test, suppose $n = pq$ with $p < q$, then in $[2, \lfloor\sqrt{n}\rfloor]$, there's only one divisor. For instance if $n = 11 \cdot 13 = 143$, $[2, \lfloor\sqrt{n}\rfloor] = [2, 11]$. and the only divisor of n in this interval is 11. The percentage of finding a divisor (with one try) in this case is 10%. The chance of finding some a such that $\gcd(a, n) > 1$ is the same. If $n = pq$ where q is much larger than p, then the GCD compositeness test has a better change of finding a good a, because $a = p, 2p, 3p, ...kp$ would work where kp is the largest integer $\leq \sqrt{pq}$. For instance is $n = 2 \cdot 13$, then $a = 2, 4 \leq \left\lfloor\sqrt{2 \cdot 13}\right\rfloor = 5$ works so the number of good as is $2/4 = 50\%$. However if you look at the table above, you will see that for $n = 26$, the percentage of Fermat witness is 100%.

For Fermat compositeness test, from the above table, the entry for $n = 143$ is

```
143 0  0.01 0 ..00000000001000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000100
00000000.
```

There's a 99% chance of finding a Fermat witness (with one try).

Exercise 3.8.2. Let $F_n = 2^{2^n} + 1$. These numbers are called **Fermat numbers** (not Fibonacci numbers). For $n = 5$, $F_5 = 2^{2^5} + 1 = 4294967297$. Fermat thought F_5 is prime, but it is not. (The first 5 Fermat numbers $F_0, F_1, ..., F_4$ are prime.) Euler was the first to realize that F_5 is not prime. Of course neither Fermat nor Euler had computers. Prove that F_5 is not a prime in two ways:

Fermat numbers

(a) Find the smallest Fermat witness for F_5.
(b) Find a prime divisor of F_5 by brute force division.

□

Exercise 3.8.3. Is 2 a Fermat witness for $F_{14} = 2^{2^{14}} + 1$? What about 3? Can you find a divisor of F_{14}? The fact that F_{14} is composite was

first discovered in 1961. It took 50 years before a nontrivial 54-digit divisor of F_{14}, 1169280858730743698290359938345963713403867034233733 13, was found. See https://t5k.org/mersenne/LukeMirror/lit/lit_039s.htm. I think the complete factorization of F_{14} is still unknown. See http://www.prothsearch.com/fermat.html. □

The above Fermat compositeness test tells you when n is a composite. It does not tell you if n is a prime. One can ask if the following is true:

$$a^{n-1} \equiv 1 \pmod{n} \implies n \text{ is prime}$$

if n is odd and $a \in [2, n-2]$. This is clearly not true. For instance in the above data for $n = 143$, you see that there are two Fermat liars, the first being $a = 12$:

$$12^{143-1} = 12^{142} = (12^2)^{71} = 144^{71} \equiv 1^{71} = 1 \pmod{144}$$

The first composite n to have a Fermat liar is $n = 15$, where the liar is $a = 4$:

```
15 0  0.17 0 ..001000000100.
```

$$4^{15-1} = 4^{14} = 16^7 \equiv 1^7 \equiv 1 \pmod{15}$$

In this case, we say that 15 is a **Fermat pseudoprime** and 4 is a **Fermat liar** for 15. (Sometimes 4 is called a **base** for 15.) Why is 15 called a pseudoprime and 4 is a liar? The fact

Fermat liar
Fermat pseudoprime

$$4^{15-1} \equiv 1 \pmod{15}$$

is very similar to Fermat Little Theorem where the modulus is a prime p and $p \nmid a$:

$$a^{p-1} \equiv 1 \pmod{p}$$

4 and 15 are trying to trick you into thinking that 15 is like a prime.

Note that if a is a Fermat liar, then it is invertible:

Proposition 3.8.2.

(a) *If a is a Fermat liar for composite n, then a is invertible mod n, i.e.,* $\gcd(a, n) = 1$.
(b) *Therefore if* $\gcd(a, n) > 1$, *then a cannot be a Fermat liar. In other words, if* $\gcd(a, n) > 1$, *then a is a Fermat witness.*

Proof. (a) TODO.

(b) This follows from (a). □

Therefore for $a \in [0, n-1]$ (although for compositeness testing we are only interested in the case when $n > 3$ is odd and $a \in [2, n-2]$):

1. a is a Fermat witness and $\gcd(a, n) = 1$
2. a is a Fermat witness and $\gcd(a, n) > 1$
3. a is a Fermat liar (and in this case $\gcd(a, n) = 1$)

So in the above table, to be "fair" to Fermat liar, if one is interested in comparing the number of Fermat liars against Fermat witnesses for n, sometimes one can temporarily ignore the a such that $\gcd(a, n) > 1$. If we do this, then there are composite numbers n where *every* a in $[2, n-2]$ such that $\gcd(a, n) = 1$ is a Fermat liar for n. Such an n is called a **Carmichael number**.

Carmichael number

Let me collect all these definitions below.

Definition 3.8.2. Let n be a composite. Then

(a) n is a **Fermat pseudoprime** if n is not a prime and if there is some $a \in [2, n-2]$ such that

Fermat pseudoprime

$$a^{n-1} \equiv 1 \pmod{n}$$

In this case, we say that a is a **Fermat liar** for n. (a is also called a **base** for n.)

Fermat liar

base

(b) n is a **Carmichael number** if every $a \in [2, n-2]$ such that $\gcd(a, n) = 1$ is a Fermat liar for n, i.e., for all such a,

Carmichael number

$$a^{n-1} \equiv 1 \pmod{n}$$

The following is similar to the earlier table:

```
 3 1  None 1 ...
 4 0  None 1 ....
 5 1   1.0 1 ..11.
 6 0  None 1 ......
 7 1   1.0 1 ..1111.
 8 0   0.0 0 ...0.0..
 9 0   0.0 0 ..0.00.0.
10 0   0.0 0 ...0...0..
11 1   1.0 1 ..11111111.
```

```
12 0   0.0 0 .....0.0....
13 1   1.0 1 ..1111111111.
14 0   0.0 0 ...0.0...0.0..
15 0  0.33 0 ..0.1..00..1.0.
16 0   0.0 0 ...0.0.0.0.0.0..
17 1   1.0 1 ..11111111111111.
18 0   0.0 0 .....0.0...0.0....
19 1   1.0 1 ..1111111111111111.
20 0   0.0 0 ...0...0.0.0.0...0..
21 0   0.2 0 ..0.00..1.00.1..00.0.
22 0   0.0 0 ...0.0.0.0...0.0.0.0..
23 1   1.0 1 ..11111111111111111111.
24 0   0.0 0 .....0.0...0.0...0.0....
25 0  0.11 0 ..000.0100.0000.0010.000.
26 0   0.0 0 ...0.0.0.0.0...0.0.0.0.0..
27 0   0.0 0 ..0.00.00.00.00.00.00.0.
28 0   0.2 0 ...0.0...1.0.0.0.0.0...0.1..
29 1   1.0 1 ..111111111111111111111111.
30 0   0.0 0 .......0...0.0...0.0...0......
31 1   1.0 1 ..1111111111111111111111111111.
32 0   0.0 0 ...0.0.0.0.0.0.0.0.0.0.0.0.0..
33 0  0.11 0 ..0.00.00.1..00.00.00..1.00.00.0.
34 0   0.0 0 ...0.0.0.0.0.0.0...0.0.0.0.0.0.0..
35 0  0.09 0 ..000.1.00.000..0000..000.00.1.000.
36 0   0.0 0 .....0.0...0.0...0.0...0.0...0.0....
37 1   1.0 1 ..11111111111111111111111111111111.
38 0   0.0 0 ...0.0.0.0.0.0.0.0...0.0.0.0.0.0.0.0..
39 0  0.09 0 ..0.00.00.00..1.00.00.00.1..00.00.00.0.
40 0   0.0 0 ...0...0.0.0.0...0.0.0.0...0.0.0.0...0..
41 1   1.0 1 ..1111111111111111111111111111111111.
42 0   0.0 0 .....0.....0.0...0.0...0.0...0.0.....0....
43 1   1.0 1 ..11111111111111111111111111111111111111.
44 0   0.0 0 ...0.0.0.0...0.0.0.0.0.0.0.0.0.0...0.0.0.0..
45 0  0.27 0 ..0.0..01..0.00.01.1..00..1.10.00.0..10..0.0.
46 0   0.0 0 ...0.0.0.0.0.0.0.0.0.0...0.0.0.0.0.0.0.0.0.0..
47 1   1.0 1 ..1111111111111111111111111111111111111111.
48 0   0.0 0 .....0.0...0.0...0.0...0.0...0.0...0.0...0.0....
49 0   0.1 0 ..00000.000000.000110.000000.011000.000000.00000.
50 0   0.0 0 ...0...0.0.0.0...0.0.0.0...0.0.0.0...0.0.0.0...0..
```

except that for the fifth column, an entry of "." is used to indicate "not applicable", where "not applicable" is when $a \in [0, 1, n-1]$ or when $\gcd(a, n) > 1$. The fourth column is 1 if all values of $a \in [2, n-1]$ satisfying $\gcd(a, n) = 1$ are Fermat liars. In this case n is a Carmichael number and the value for the third column (percentage of Fermat liars) is $1.0 = 100\%$.

Exercise 3.8.4. Modify your program to produce the above output. □

The first Carmichael number is 561. Here's the row for 561 from the table:

```
561 0   1.0 1 ..1.11.11.1..11.1..11..1.11.11.11..1.11.11.1..11.11.11..1.11
.11.11.1..11.11.1..11.11..1..1.11.11.11.11.11.11.1..11.11.1...1.11.11.11.1
1..1.11.1..11.11.11..1.11.11.11.11.1..11.1..11.11.11..1.11.11.11.11.11.11.
1..11.11.11....11.11.11.11.11..1.1..11.11.11..1.11.11.11.11.11.1..1..11.11
.11..1..1.11.11.11.11.11.1..11.11.11..1.1..11.11.11.11.11....11.11.11..1.1
1.11.11.11.11.11.1..11.11.11..1.11..1.11.11.11.11.1..11.11.11..1.11.1..11.
11.11.11.1...1.11.11..1.11.11.11.11.11.11.1..1..11.11..1.11.11..1.11.11.11
.1..11.11.11..1.11.11.1..11.11.11.1..11..1.11..1.11.11.1.
```

The concept of Carmichael numbers was named after Carmichael because of a 1910 paper he wrote. However the defining property of a Carmichael number was studied as far back as at least 1885. For instance the first 6 Carmichael numbers (including 561) were first discovered by Šimerka in 1885. In his 1910 paper, Carmichael conjectured that there are infinitely many Carmichael numbers. This was not proven until 1994 (see `https://math.dartmouth.edu/~carlp/PDF/paper95.pdf`). Here's wikipedia entry for Carmichael numbers.

Exercise 3.8.5. Šimerka was the first to discover the first 7 Carmichael numbers. Beat him by finding the first 8. To check your work, you can go to Carmichael numbers and look for the first 7 Carmichael numbers discovered by Šimerka. □

The above gives us a probabilistic primality test: if $n > 3$ and we pick $a \in [2, n-2]$ and we have

$$a^{n-1} \equiv 1 \pmod{n}$$

then n is "probably a prime". Why? Because

$$a^{n-1} \equiv 1 \pmod{n}$$

can occur in two ways:

1. n is a prime (the fact that $n \nmid a$ is guaranteed by the fact that n is prime and $a \in [2, n-2]$)
2. n is a Fermat pseudoprime and a is Fermat liar for n

From the first table, we see that most of the "1" occurs when n is a prime. For a fixed n, the more a's we try, the higher the probability that n is prime, unless of course our n has lots of liars. In particular, in the worse scenario, if n is a Carmichael number, the test will say n is very likely a prime when in fact it's not.

Combining the Fermat compositeness test with the "probably prime test", we now have:

Fermat Primality Test

Proposition 3.8.3. (**Fermat Primality Test**) *If there is some a such that $1 \leq a \leq n-1$ and*

$$a^{n-1} \not\equiv 1 \pmod{n}$$

then n is composite. If there is some randomly chosen $a \in [2, n-2]$ such that

$$a^{n-1} \equiv 1 \pmod{n}$$

then n is "probably a prime".

Here's the Fermat primality test algorithm:

```
ALGORITHM: Fermat-primality-test
INPUTS: n -- number to be tested for compositeness/primeness. Assume n >= 2.
        t -- number of tries
OUTPUT: "n is composite" or "n is probably prime" (after t tries)

if n is 2: return "n is prime"
if n % 2 is 0 and n / 2 > 1: return "n is composite"

for i = 1, 2, 3, ..., t:
    pick random a in [2, n - 2]
    if a^n - 1 ≢ 1 (mod n) return "n is composite"

return "n is probably prime"
```

Note that we have not included the handling of $n \leq 1$. In general, primality tests are for handling cases when n is large. We have added a note that n is assumed to be ≥ 2.

Note that the first 2 checks (i.e., n is even) can be omitted:

```
ALGORITHM: Fermat-primality-test
INPUTS: n -- number to be tested for compositeness/primeness. Assume n >= 2.
        t -- number of tries
OUTPUT: "n is composite" or "n is probably prime" (after t tries)

for i = 1, 2, 3, ..., t:
    pick random a in [2, n - 2]
    if a^n - 1 ≢ 1 (mod n) return "n is composite"

return "n is probably prime"
```

Why? Because if $n = 2$, then $[2, n-2]$ is empty, which means that `"n is probably prime"` is returned. And if $n = 2k$ where $k > 1$, then $\gcd(a, n) > 1$ whenever a is even or when $a = k$. This means that more than half of the values in $[2, n-2]$ are Fermat witnesses, so there is a strong likeliheed that `"n is composite"` will be returned especially if $t > 1$.

Note that when given an integer $n \geq 0$, either n is a prime or it is not. The statement "n is probably a prime" should actually be "there's probably find a value in $[2, n-2]$ satisfying some condition". But the phrase "n is probably a prime" has been in use for a long time and it's hard to break the usage.

When comparing number of Fermat witnesses against Fermat liar, we can be a little bit more precise:

Proposition 3.8.4. *Let n be composite. If there is a Fermat witness w for n such that $\gcd(w, n) = 1$, then*

$$|\{a \in [1, n-1] \mid a^{n-1} \not\equiv 1 \pmod{n}\}| > \frac{n-1}{2}$$

Proof. TODO. □

In the above proposition, the a is an integer in $[1, n-1]$. Remember that we usually test $a \in [2, n-2]$. Also, we usually assume $n > 3$ is odd. Note that $1^{n-1} \equiv 1 \pmod{n}$. Hence the above proposition implies

$$|\{a \in [2, n-1] \mid a^{n-1} \not\equiv 1 \pmod{n}\}| > \frac{n-1}{2}$$

If n is odd. Then $n-1$ is even and $(n-1)^{n-1} \equiv (-1)^{n-1} \equiv 1 \pmod{n}$. Hence

$$|\{a \in [2, n-2] \mid a^{n-1} \not\equiv 1 \pmod{n}\}| > \frac{n-1}{2}$$

Also, $(n-1)/2$ is an integer. Therefore

$$|\{a \in [2, n-2] \mid a^{n-1} \not\equiv 1 \pmod{n}\}| \geq \frac{n-1}{2} + 1 = \frac{n+1}{2}$$

The number of integer in $[2, n-2]$ is $n-2-1 = n-3$. Therefore $> 50\%$ of the values in $[2, n-2]$ are Fermat witnesses. If n is even. Then $n-1$ is odd and $(n-1)^{n-1} \equiv (-1)^{n-1} \equiv -1 \pmod{n}$.

$$|\{a \in [2, n-2] \mid a^{n-1} \not\equiv 1 \pmod{n}\}| + 1 > \frac{n-1}{2}$$

In this case $(n-1)/2$ is not an integer.

$$|\{a \in [2, n-2] \mid a^{n-1} \not\equiv 1 \pmod{n}\}| + 1 \geq \frac{n-1}{2} + \frac{1}{2}$$

i.e.,

$$|\{a \in [2, n-2] \mid a^{n-1} \not\equiv 1 \pmod{n}\}| \geq \frac{n-1}{2} - \frac{1}{2} = \frac{n-2}{2}$$

Together,

$$|\{a \in [2, n-2] \mid a^{n-1} \not\equiv 1 \pmod{n}\}| > \begin{cases} \frac{n+1}{2} & \text{if } n \text{ is odd} \\ \frac{n-2}{2} & \text{if } n \text{ is even} \end{cases}$$

In both of the above cases

$$|\{a \in [2, n-2] \mid a^{n-1} \not\equiv 1 \pmod{n}\}| > \frac{n-3}{2}$$

i.e., more than $1/2$ of the values in $[2, n-2]$ are witnesses.

Suppose p is the probability of not finding a Fermat witness a in $[2, n-2]$ for n. Assuming there is a Fermat witness a with $\gcd(a, n) = 1$. From the above, more than half of the a in $[2, n-2]$ are Fermat witnesses for n. This means that $p < 0.5$. The probability of not finding a Fermat witness after t tries is

$$p^t$$

Therefore the probability of finding a Fermat witness is

$$1 - p^t$$

For instance assuming $p = 0.5$, then $t = 8$ tries, if no Fermat witness is found, the probability that n is prime is

$$1 - p^t > 1 - 0.5^8 = 0.99609375$$

If $t = 10$, we have

$$1 - p^t > 1 - 0.5^{10} = 0.9990234375$$

And if $t = 20$, we reach

$$1 - p^t > 1 - 0.5^{10} = 0.9999990463256836$$

Of course all the above is based on the assumption that there is a Fermat

witness a such that $\gcd(a, n) = 1$. You are out of luck if n is a Carmichael number. You can think of Carmichael numbers as extreme failure cases of Fermat primality test.

The question is how common is Carmichael numbers? Carmichael numbers are very rare. At this point, we know that the density of Carmichael numbers is about 1 in 50 trillion $= 5 \times 10^{13}$.

Example 3.8.1. In the following, we obtain a random integer with 10 digits and found a Fermat witness with one try:

```
import random; random.seed()
d = 10
n = random.randrange(10**(d - 1), 10**d)
print(n)
for i in range(20):
    a = random.randrange(2, n - 1)
    b = (pow(a, n - 1, n) != 1)
    print(i, a, b)
    if b: break
```

The output is

```
1151731626
0 869958907 True
```

Fermat primality test says that 1151731626 is composite with only one try. In fact $1151731626 = 2 \cdot 3 \cdot 19 \cdot 10102909$.

Example 3.8.2. Here's another run of the above code where no Fermat witness was found after 20 tries:

```
3584990077
0 3295070400 False
1 3215421426 False
2 262972142 False
3 1050903352 False
4 152804132 False
5 1015451650 False
6 885960417 False
7 720561088 False
8 1694137561 False
9 2065337998 False
10 3345601994 False
```

```
11 1535607663 False
12 3183174792 False
13 1772850385 False
14 2569697199 False
15 948739551 False
16 2148646472 False
17 1640965445 False
18 2024258764 False
19 3229113680 False
```

No Fermat witness was found after 20 tries. Fermat primality test would return "probably a prime". In fact 3584990077 is prime. In this case 3584990077 is small enough that a brute force prime testing by brute force division can be used.

Exercise 3.8.6. Let $n = 18801105946394093459$. Prove that n is composite in three ways:

(a) Use division compositeness test. First try to randomly generate a potential divisor and test it. If it fails, try to do a brute force iteration from 2 to $\sqrt{n}$ to locate a divisor.
(b) Use GCD compositeness test. First try to randomly generate a potential a and test if $\gcd(a, n) > 1$. If it fails, try to do a brute force iterate a from 2 to $\sqrt{n}$ and test if $\gcd(a, n) > 1$.
(c) Use Fermat primality test with $t = 1$.

□

Exercise 3.8.7. Let $n = 58645563317564309847334478714939069495243200674793$. Prove that n is prime or probably prime in three ways:

(a) Use division compositeness test. Do a brute force iteration from 2 to $\sqrt{n}$ to locate a potential divisor.
(b) Use GCD compositeness test. Iterate a from 2 to $\sqrt{n}$ and test if $\gcd(a, n) > 1$.
(c) Use Fermat primality test with $t = 10$.

□

Exercise 3.8.8. How often is 2 a liar? Write a program to print all composite n such that 2 lies for n. Do you think 2 lie for infinitely many Fermat

pseudoprimes? Can you prove it? Compare your experiment with 3, say up to $n = 100000$, how often does 3 lie for n when compared to 2? □

The following is a major theorem on Carmichael numbers:

Theorem 3.8.1. (**Koselt's Theorem** 1899) *Let n be a positive integer. Then n is a Carmichael number iff n is odd, squarefree, and if $p \mid n$, then $p-1 \mid n-1$.* Koselt's Theorem

Proof. Omitted. □

The conditions on n in the above theorem is frequently called **Koselt's criterion**. Koselt's criterion

Example 3.8.3. Consider 561. From the above second table, we know that 561 is a Carmichael. Also, as mentioned earlier Šimerka was the first to discover this Carmichael number. 561 is small and can be easily factorized: $561 = 3 \cdot 11 \cdot 17$. Note that 561 is odd and squarefree. Furthermore $2 \mid 560$, $10 \mid 560$, and $16 \mid 560$. Therefore 561 satisfies Koselt's criteria and therefore 561 is a Carmichael number.

Exercise 3.8.9. Show that 1729 is a Carmichael number.

Although Koselt proved his theorem in 1899, he never mention any Carmichael number in his publications.

Corollary 3.8.1. (Cernick 1939) *If $6k+1$, $12k+1$, and $18k+1$ are all primes, then the product $(6k+1)(12k+1)(18k+1)$ is a Carmichael number.*

Proof. TODO. □

Exercise 3.8.10. Use Corollary 3.8.1 to show that 1729 is a Carmichael number. □

Exercise 3.8.11. Use Corollary 3.8.1 to find your own Carmichael number. Check what you have discovered with known Carmichael numbers on the web. □

Exercise 3.8.12. Implement Fermat's primality test. □

File: miller-rabin-primality-test.tex

3.9 Miller-Rabin primality test

Fermat Little Theorem says this: Let $a \in [2, n-2]$.

$$n \text{ is prime } \implies a^{n-1} \equiv 1 \pmod{n}$$

The basis of Fermat primality test is

$$n \text{ is not prime } \impliedby a^{n-1} \not\equiv 1 \pmod{n}$$

We can a say bit more. Again we have

$$n \text{ is prime } \implies a^{n-1} \equiv 1 \pmod{n}$$

Suppose $n-1 = 2^k m$ with $k \geq 0$ and $2 \nmid m$. Then we have

$$n \text{ is prime } \implies a^{2^k m} \equiv 1 \pmod{n}$$

Now note that

$$a^{2^k m} = (a^m)^{2^k}$$

This can be computed as a sequence of squares. For instance if $k = 3$, then

$$a^{2^3 m} = (a^m)^{2^3} = (((a^m)^2)^2)^2$$

Note that following fact:

Proposition 3.9.1. *If p is a prime and $x^2 \equiv 1 \pmod{p}$ then $x \equiv \pm 1 \pmod{p}$.*

Proof. TODO. □

Applying this proposition to $x^2 = (a^m)^{2^k}$, we have the following fact: if n is prime, and $k > 0$, then

$$(a^m)^{2^k} \equiv 1 \pmod{n} \implies (a^m)^{2^{k-1}} \equiv \pm 1 \pmod{n}$$

And if

$$(a^m)^{2^{k-1}} \equiv 1 \pmod{n}$$

then

$$(a^m)^{2^{k-2}} \equiv \pm 1 \pmod{n}$$

Etc.

All in all, assuming n is prime, writing $n-1$ as $2^k m$ where 2^k is the highest power of 2 dividing $n-1$, then the sequence

$$\begin{aligned}
&(a^m)^{2^0} \pmod{n}\\
&(a^m)^{2^1} \pmod{n}\\
&(a^m)^{2^2} \pmod{n}\\
&\vdots\\
&(a^m)^{2^{k-2}} \pmod{n}\\
&(a^m)^{2^{k-1}} \pmod{n}\\
&(a^m)^{2^k} \pmod{n}
\end{aligned}$$

either the whole sequence is 1s or it ends with a sequence of 1s (of length ≥ 1) and before this sequence there is a -1. For instance if $k=5$, the above sequence is a sequence of 6 numbers and here are some possibilities:

- The last three numbers might be $-1, 1, 1 \pmod{n}$ (the first three are not -1 and not 1).
- Or the last two might be $-1, 1 \pmod{n}$ (the first four are not -1 and not 1).
- Or all 6 might be $1, 1, 1, 1, 1, 1 \pmod{n}$.

Of course if $k=0$, then there is only one number in the sequence and that number is $1 \pmod{n}$. In general, the above is a sequence of $k+1$ numbers and ends with 1s of length ≥ 1 or ends with -1 followed by 1s of length ≥ 1.

Miller-Rabin primality test is similar to Fermat prime test. For an integer n, we compute m and k such that $n-1 = 2^k \cdot m$. We randomly pick an a in

$[2, n-2]$ look at the values

$$\begin{aligned}
&a^m \pmod n\\
&a^{2m} \pmod n\\
&a^{2^2m} \pmod n\\
&a^{2^3m} \pmod n\\
&\vdots\\
&a^{2^{k-2}m} \pmod n\\
&a^{2^{k-1}m} \pmod n\\
&a^{2^km} \pmod n
\end{aligned}$$

and if they are all 1s (i.e., the first is 1) or ends with $-1, 1, 1, ...1$, the algorithm (i.e., there's a -1) returns `"n is probably prime"`. Otherwise it returns `"n is composite"`. The difference is Fermat primality test only looks at the last value.

Example 3.9.1. As an example, note that $n = 561$ is a Carmichael number. Fermat primality test reports n as probably prime even though it is a composite. Using Miller-Rabin primality test, first $n - 1 = 560 = 35 \cdot 2^4$. Let us use $a = 2$.

$$\begin{aligned}
2^{35} &\equiv 263 \pmod{561}\\
(2^{35})^2 &\equiv (263)^2 \equiv 166 \pmod{561}\\
(2^{35})^{2^2} &\equiv (166)^2 \equiv 67 \pmod{561}\\
(2^{35})^{2^3} &\equiv (67)^2 \equiv 1 \pmod{561}\\
(2^{35})^{2^4} &\equiv (1)^2 \equiv 1 \pmod{561}
\end{aligned}$$

The sequence is $263, 166, 67, 1, 1$. And we see that n cannot be prime, because if n is prime, from line 3 above

$$(2^{35})^{2^3} \equiv 1 \pmod{561}$$

the previous line should have been $\pm 1 \pmod{561}$, but it is not. Therefore Miller-Rabin primality test will report 561 as composite.

One can define Miller-Rabin pseudoprimes (usually called **strong pseudoprimes**), Miller-Rabin witness, and Miller-Rabin liar. While the failure case of Fermat primality test is on the average less than 1/2, the failure case of

strong pseudoprimes

Miller-Rabin is less than 1/4.

Here's the Miller-Rabin primality test algorithm:

```
ALGORITHM: Miller-Rabin-primality-test
INPUTS: n -- integer to be tested for primeness/compositeness
        t -- number of passes

compute k and odd m such that n - 1 = 2^k * m
for pass = 1, 2, 3, ..., t:
    randomly select a in [2, n - 2]
    if Miller-Rabin-one-pass(n, k, m, a) returns "n is composite":
        return "n is a composite"
return "n is probably a prime"


ALGORITHM: Mill-Rabin-one-pass
INPUTS: n, k, m where n = 2^k * m
        a
OUTPUT: "n is composite" or "n is probably prime"

let b ≡ a^m (mod n)
if b ≡ 1 (mod n):
    return "n is probably prime"
for i = 0, 1, 2, ..., k - 1:
    if b ≡ -1 (mod n):
        return "n is probably prime"
    b ≡ b^2 (mod n)
return "n is composite"
```

Exercise 3.9.1. Are the Miller-Rabin computations for the case of $a = 50$ and $n = 561$? What does Miller-Rabin conclude in this case?

Exercise 3.9.2. Compare the results from Fermat primality test and Miller–Rabin primality test for $n = 1729$.

Exercise 3.9.3. Implement the Miller–Rabin primality test algorithm.

File: monte-carlo.tex

3.10 Monte-Carlo algorithms

As noted earlier, Fermat and Miller–Rabin prime testing algorithm are probabilitic algorithm in the sense that if the return value is `"n is composite"` then you know for sure n is a composite (i.e. not a prime), but if the return value is `"n is probably a prime"`, then n can be either be a prime or a composite. This is also called a **Monte-Carlo algorithm** because the result returned is not guaranteed to be true. It's also called a **false-biased Monte-Carlo algorithm** because the false case (i.e., `"not a prime"` return value) is always correct whereas the true case (i.e., is a prime) is only probabilistically true.

Monte-Carlo algorithm

false-biased Monte-Carlo algorithm

There are actually many primality testing algorithms. Rabin-Miller is only one of many.

By the way for a long time it was thought that primality test is "easy". Note that Miller–Rabin prime testing (and other primality tests) is easy probabilistically. A deterministic polynomial runtime primality test was finally discovered and proven only recently (2002) by a group of computer scientists, Agrawal, Kayal and Saxena, from India. The algorithm is now called the **AKS primality test** algorithm. If you want some fancy automata notation, the AKS algorithm says that

AKS primality test

$$\textsc{Primes} \in \mathrm{P}$$

where Primes denotes the problem (or language) of testing for primeness and P denotes the class of polynomial runtime problems. This P is the same P in the famous "P = NP" problem. AKS is however not used in real-world applications because the runtime is too slow. There is current ongoing research on improving the performance of this algorithm.

In real-world applications of Miller-Rabin prime testing algorithm, to test that a random 2048-bit odd number is a prime, using $t = 10$ rounds is usually more than enough.

In 2007, a 1039 bit integer was factored with the number field sieve using 400 computers over 11 months. Nowadays (2019), primes with 2048 bit length is definitely enough – unless someone discovered a new factoring algorithm. In RSA–speak, when you hear "RSA 1024-bit key", it means the modulus $N = pq$ has 1024 bit. That means the bit length of each of the two primes is about 512.

Exercise 3.10.1. Good research project: Study the AKS algorithm. □

File: carmichael-function.tex

3.11 Carmichael function

multiplicative order

Definition 3.11.1. The **multiplicative order** of $a \bmod n$, if it exists, is the smallest positive integer k such that $a^k \equiv 1 \pmod{n}$.

Some values might not have multiplicative order. For instance 0^k is not $\equiv 1 \pmod{n}$ for all k. Recall that a has a multiplicative inverse mod n iff $a^k \equiv 1 \pmod{n}$ for some $k > 0$.

Recall Euler's theorem: If $\gcd(a, n) = 1$, then

$$a^{\phi(n)} \equiv 1 \pmod{n}$$

Is $\phi(n)$ the best possible in the sense that $\phi(n)$ gives you the smallest for the above to be true?

For instance when $n = 2$, $\phi(2) = 1$ which is the smallest possible positive integer to satisfy

$$a^{\phi(n)} \equiv 1 \pmod{n}$$

For $n = 3$, $\phi(3) = 2$ and

$$\begin{aligned} 1^1 &\equiv 1, \quad 2^1 \equiv 2 \pmod{3} \\ 1^2 &\equiv 1, \quad 2^2 \equiv 1 \pmod{3} \end{aligned}$$

So $\phi(3) = 2$ is the smallest for

$$a^{\phi(n)} \equiv 1 \pmod{n}$$

to be true for both $a = 1$ and $a = 2$. Etc. But when you reach $n = 8$ when $\phi(8) = 4$, if $\gcd(a, 8) = 1$, then $a = 1, 3, 5, 7$ and

$$\begin{aligned} 1^2 &\equiv 1 \pmod{8} \\ 3^2 &\equiv 1 \pmod{8} \\ 5^2 &\equiv 1 \pmod{8} \\ 7^2 &\equiv 1 \pmod{8} \end{aligned}$$

and $2 < 4$ (in fact $2 \mid 4$). In other words 2 satisfies

$$a^2 \equiv 1 \pmod{8}$$

for all a such that $\gcd(a, 8) = 1$. Of course we know from Euler's theorem that

$$a^{\phi(8)} \equiv 1 \pmod 8$$

and $\phi(8) = 4$. Clearly

$$a^2 \equiv 1 \pmod 8 \implies a^4 \equiv 1 \pmod 8$$

Let write $\lambda(8) = 4$. In general:

Definition 3.11.2. Define the **Carmichael function**

Carmichael function

$$\lambda(n)$$

to be the LCM (lowest common multiple) of the multiplicative order of a mod n for all $a \in \{1, 2, ..., n\}$ satisfying $\gcd(a, n)$. The multiplicative order of a is the smallest positive integer k such that $a^k \equiv 1 \pmod n$.

In other words, $\lambda(n)$ is "almost" $\phi(n)$.

Theorem 3.11.1.

(a) $\lambda(mn) = \lambda(m)\lambda(n)$ *if* $\gcd(m, n) = 1$.
(b) *Let* p *be a prime and* $k \geq 0$. *Then*

$$\lambda(p^k) = \begin{cases} \phi(p^k) & \text{if } p > 2 \\ \phi(p^k) & \text{if } p = 2, k = 0, 1, 2 \\ \frac{1}{2}\phi(p^k) & \text{if } p = 2, k \geq 3 \end{cases}$$

Theorem 3.11.2.

(a) *Let* $\gcd(a, n) = 1$, *if* $a^k \equiv 1 \pmod n$, *then* $k \mid \lambda(n)$.
(b) *If*

$$a^k \equiv 1 \pmod n \quad \text{for all } \gcd(a, n) = 1$$

then

$$\lambda(n) \mid k$$

(c) $\lambda(n) \mid \phi(n)$.

File: openssl.tex

3.12 OpenSSL

Exercise 3.12.1. Do this is your bash shell:

```
openssl genpkey -algorithm RSA -out private_key.pem \
    -pkeyopt rsa_keygen_bits:2048
```

And you will get an RSA key. The key is stored in the file `private_key.pem`. The utility you are using is openssl. Here's an example of the file:

```
-----BEGIN PRIVATE KEY-----
MIIEvQIBADANBgkqhkiG9w0BAQEFAASCBKcwggSjAgEAAoIBAQDD4UOgAdmz0G5I
LGmpxx5DNWclrpINB/bH1aLiFk4lxh85gE83UX3dEirn1PaVxSB4qvMr9dY0yZ8G
jd7Yj/BubkOAYhlclWbiRERRWcGigmP/CvJWP7MSarC4sTO4QGv0X6+oj64jkM55
WLApi6jHDprg4Un7LT/IJVZbr2hbmu1D6wPPYN2D1uZpavOskL6+SyYnl5U3EimQ
M9BO5FM7K7yiRDeOFXCHgfUbh5PULZLc1u/vBdr7OWopRwFRTdFdGgAY3cHG3p7d
CLC+vNl7DaN7JsJFSoyPq5ynok1P16l9AdvnHwzVtkEtk2tYaQeZCjtyuM+3jzQ4
zQt1Et7tAgMBAAECggEANaii3tVC7vg9DbZk56ZtStn5PKBaOAkLeGiOqxyTIdPp
P9Y/XRcM1J+ic0mqlxKeN5AU90jr+h/1WVVJ46dipM3AeEdnTS58NaWf1W0yFzOC
8x3rjub6RiRF7wJWk+9J43LG6vUZLhMADMvXzjm87XK5yLrOimk13L0lsA4YF2eZ
d+EN/xaop2Olw76PSQkiVseVGKcvZo6lJMxZAgLMUTX5CmWuOU1z8yR/Lgj9CPo4
KO2iKOhnkePEmR9OiHZ5PRPHb4Wb3CsCniwKqHOxJpAuUVYr532r9yt6rUwotw6E
OadIcW2e5XICGRUddBhQ4Di3tN89qHVOTic6M0jQwQKBgQDvyjl7/CcyC0/86Gvj
DDXpr1m4ePSNMyY+Oy5h5S1FB2KNibNLfZp2VYnNX73R63AZQOxNNjCDfHyiBLb2
NJ7arqjqWH846N0cmSdI7Fpo38kyMKwwq95Zl1PqDhahjoZQSq00iTh2R1YXvGmM
qRSes8aLO3l3A9KRGIRnuMOTUQKBgQDRHyNzfTPw8TNSLU2QFUpmZ/6XeSjtRKxP
5dGocZ4YxLb/5FLKZeQCRZWA8ocbllwRFcuI1VgfaSclBCb8ElS76Qw/wWjaYJ/V
qoZgKCvqtrZKruax2+gB59LjJGRqGf537F3V4qB4QP2tp4glTxiQL9yFr4p8e72R
REgBO5ES3QKBgAT2HjJeiUET0tfcxz6vZf4rzqNufUDeqg/nkZIc987R1EwxaTBK
rQN9yZgiPv8O6+DZ4wnF8UMHNFz11ANMG21S59PRePBogQqycImlukkpODR9pVJs
e/FGnEnfeMBm/ohywxqvLCfmWfWrxFNQvEh8V8NRu7Wmspil9TdgL0vBAoGBAKki
9DteUnpXu1iFx6v3bFtzVRkSJ6Xv2yYsDOyeKG6D/DbvZn7I9idYPFk0z03iyMgQ
xrP/Sezt0XlA6H8MHHh3Py75sWKer+fSqihvlUWbTckNuQy1feq8o3aPYp/mMkiw
Zhyt1XgtqH+hdp4mYQmNjGCb3/ha5LHvdgX0JewJAoGAPT3a1Zb5xPQ6RARQSX2b
Tk6AXHH7vsuYf18c0KyruUAhbQ6CUTqemz4qY5VnlWmORP277ceb9i+NtiRvm4Rd
HoyZtvZvZc0zTtIsxYaFU6pPnnexrNgRC7+jCoAqfHeShJ/fNLiHA/Ffy06S6eQV
Xo8vamqc1SMq2tQegRBEV9s=
-----END PRIVATE KEY-----
```

You can search for a website that decode PEM file data for you and see what is the data stored in this file. Here's one: https://lapo.it/asn1js/. To find out which are the primes, etc., you can check the the spec at IEFT https://tools.ietf.org/html/rfc2313#section-7.2:

```
An RSA private key shall have ASN.1 type RSAPrivateKey:

   RSAPrivateKey ::= SEQUENCE {
     version Version,
     modulus INTEGER, -- n
     publicExponent INTEGER, -- e
     privateExponent INTEGER, -- d
     prime1 INTEGER, -- p
     prime2 INTEGER, -- q
     exponent1 INTEGER, -- d mod (p-1)
     exponent2 INTEGER, -- d mod (q-1)
     coefficient INTEGER -- (inverse of q) mod p }
```

You can also extract the public key from your file:

```
openssl rsa -pubout -in private_key.pem -out public_key.pem
```

which can then be sent to your friend (or a server) for communication. These files are pretty standard and can be use by encryption/decryption programs to perform RSA/AES/3DES/... encryption and decryption. Or you can execute

```
openssl rsa -in private_key.pem -noout -text
```

Either way, this will display a list of 9 integers. The command line gives this output:

```
Private-Key: (2048 bit)
modulus:
    00:c3:e1:43:a0:01:d9:b3:d0:6e:48:2c:69:a9:c7:
    1e:43:35:67:25:ae:92:0d:07:f6:c7:d5:a2:e2:16:
    4e:25:c6:1f:39:80:4f:37:51:7d:dd:12:2a:e7:d4:
    f6:95:c5:20:78:aa:f3:2b:f5:d6:34:c9:9f:06:8d:
    de:d8:8f:f0:6e:6e:4d:00:62:19:5c:95:66:e2:44:
    44:51:59:c1:a2:82:63:ff:0a:f2:56:3f:b3:12:6a:
    b0:b8:b1:3d:38:40:6b:f4:5f:af:a8:8f:ae:23:90:
    ce:79:58:b0:29:8b:a8:c7:0e:9a:e0:e1:49:fb:2d:
    3f:c8:25:56:5b:af:68:5b:9a:ed:43:eb:03:cf:60:
    dd:83:d6:e6:69:6a:f3:ac:90:be:be:4b:26:27:97:
    95:37:12:29:90:33:d0:4e:e4:53:3b:2b:bc:a2:44:
    37:8e:15:70:87:81:f5:1b:87:93:d4:2d:92:dc:d6:
    ef:ef:05:da:fb:d1:6a:29:47:01:51:4d:d1:5d:1a:
    00:18:dd:c1:c6:de:9e:dd:08:b0:be:bc:d9:7b:0d:
    a3:7b:26:c2:45:4a:8c:8f:ab:9c:a7:a2:4d:4f:d7:
    a9:7d:01:db:e7:1f:0c:d5:b6:41:2d:93:6b:58:69:
    07:99:0a:3b:72:b8:cf:b7:8f:34:38:cd:0b:75:12:
    de:ed
publicExponent: 65537 (0x10001)
privateExponent:
    35:a8:a2:de:d5:42:ee:f8:3d:0d:b6:64:e7:a6:6d:
    4a:d9:f9:3c:a0:5a:d0:09:0b:78:68:b4:ab:1c:93:
    21:d3:e9:3f:d6:3f:5d:17:0c:d4:9f:a2:73:49:aa:
    97:12:9e:37:90:14:f7:48:eb:fa:1f:f5:59:55:49:
    e3:a7:62:a4:cd:c0:78:47:67:4d:2e:7c:35:a5:9f:
    d5:6d:32:17:33:82:f3:1d:eb:8e:e6:fa:46:24:45:
    ef:02:56:93:ef:49:e3:72:c6:ea:f5:19:2e:13:00:
    0c:cb:d7:ce:39:bc:ed:72:b9:c8:ba:ce:8a:69:35:
    dc:bd:25:b0:0e:18:17:67:99:77:e1:0d:ff:16:a8:
    a7:63:a5:c3:be:8f:49:09:22:56:c7:95:18:a7:2f:
    66:8e:a5:24:cc:59:02:02:cc:51:35:f9:0a:65:ae:
    d1:4d:73:f3:24:7f:2e:08:fd:08:fa:38:28:ed:a2:
    28:e8:67:91:e3:c4:99:1f:4e:88:76:79:3d:13:c7:
    6f:85:9b:dc:2b:02:9e:2c:0a:a8:7d:31:26:90:2e:
    51:56:2b:e7:7d:ab:f7:2b:7a:ad:4c:28:b7:0e:84:
    39:a7:48:71:6d:9e:e5:72:02:19:15:1d:74:18:50:
    e0:38:b7:b4:df:3d:a8:75:4e:4e:27:3a:33:48:d0:
    c1
prime1:
    00:ef:ca:39:7b:fc:27:32:0b:4f:fc:e8:6b:e3:0c:
    35:e9:af:59:b8:78:f4:8d:33:26:3e:3b:2e:61:e5:
    2d:45:07:62:8d:89:b3:4b:7d:9a:76:55:89:cd:5f:
    bd:d1:eb:70:19:40:ec:4d:36:30:83:7c:7c:a2:04:
    b6:f6:34:9e:da:ae:a8:ea:58:7f:38:e8:dd:1c:99:
```

```
    27:48:ec:5a:68:df:c9:32:30:ac:30:ab:de:59:97:
    53:ea:0e:16:a1:8e:86:50:4a:ad:34:89:38:76:47:
    56:17:bc:69:8c:a9:14:9e:b3:c6:8b:3b:79:77:03:
    d2:91:18:84:67:b8:c3:93:51
prime2:
    00:d1:1f:23:73:7d:33:f0:f1:33:52:2d:4d:90:15:
    4a:66:67:fe:97:79:28:ed:44:ac:4f:e5:d1:a8:71:
    9e:18:c4:b6:ff:e4:52:ca:65:e4:02:45:95:80:f2:
    87:1b:96:5c:11:15:cb:88:d5:58:1f:69:27:25:04:
    26:fc:12:54:bb:e9:0c:3f:c1:68:da:60:9f:d5:aa:
    86:60:28:2b:ea:b6:b6:4a:ae:e6:b1:db:e8:01:e7:
    d2:e3:24:64:6a:19:fe:77:ec:5d:d5:e2:a0:78:40:
    fd:ad:a7:88:25:4f:18:90:2f:dc:85:af:8a:7c:7b:
    bd:91:44:48:01:d3:91:12:dd
exponent1:
    04:f6:1e:32:5e:89:41:13:d2:d7:dc:c7:3e:af:65:
    fe:2b:ce:a3:6e:7d:40:de:aa:0f:e7:91:92:1c:f7:
    ce:d1:d4:4c:31:69:30:4a:ad:03:7d:c9:98:22:3e:
    ff:34:eb:e0:d9:e3:09:c5:f1:43:07:34:5c:f5:d4:
    03:4c:1b:6d:52:e7:d3:d1:78:f0:68:81:0a:b2:70:
    89:a5:ba:49:29:38:34:7d:a5:52:6c:7b:f1:46:9c:
    49:df:78:c0:66:fe:88:72:c3:1a:af:2c:27:e6:59:
    f5:ab:c4:53:50:bc:48:7c:57:c3:51:bb:b5:a6:b2:
    98:a5:f5:37:60:2f:4b:c1
exponent2:
    00:a9:22:f4:3b:5e:52:7a:57:bb:58:85:c7:ab:f7:
    6c:5b:73:55:19:12:27:a5:ef:db:26:2c:0c:ec:9e:
    28:6e:83:fc:36:ef:66:7e:c8:f6:27:58:3c:59:34:
    cf:4d:e2:c8:c8:10:c6:b3:ff:49:ec:ed:d1:79:40:
    e8:7f:0c:1c:78:77:3f:2e:f9:b1:62:9e:af:e7:d2:
    aa:28:6f:95:45:9b:4d:c9:0d:b9:0c:b5:7d:ea:bc:
    a3:76:8f:62:9f:e6:32:48:b0:66:1c:ad:d5:78:2d:
    a8:7f:a1:76:9e:26:61:09:8d:8c:60:9b:df:f8:5a:
    e4:b1:ef:76:05:f4:25:ec:09
coefficient:
    3d:3d:da:d5:96:f9:c4:f4:3a:44:04:50:49:7d:9b:
    4e:4e:80:5c:71:fb:be:cb:98:7f:5f:1c:d0:ac:ab:
    b9:40:21:6d:0e:82:51:3a:9e:9b:3e:2a:63:95:67:
    95:69:8e:44:fd:bb:ed:c7:9b:f6:2f:8d:b6:24:6f:
    9b:84:5d:1e:8c:99:b6:f6:6f:65:cd:33:4e:d2:2c:
    c5:86:85:53:aa:4f:9e:77:b1:ac:d8:11:0b:bf:a3:
    0a:80:2a:7c:77:92:84:9f:df:34:b8:87:03:f1:5f:
    cb:4e:92:e9:e4:15:5e:8f:2f:6a:6a:9c:d5:23:2a:
    da:d4:1e:81:10:44:57:db
```

Here's a translation from the above to our notation:

$$\begin{aligned}
\texttt{modulus:} &\quad N = pq \\
\texttt{publicExponent:} &\quad e \\
\texttt{privateExponent:} &\quad d \\
\texttt{prime1:} &\quad p \\
\texttt{prime2:} &\quad q \\
\texttt{exponent1:} &\quad d \pmod{p-1} \\
\texttt{exponent2:} &\quad e \pmod{q-1} \\
\texttt{coefficient:} &\quad q^{-1} \pmod{p}
\end{aligned}$$

In base 10, the first prime p is

```
168386221992881634970727134992157600824442771211018733163881316415871
609202758509005188823058532708706533796182338020870980037039099058025
872225869715472355018947083739962393036214794584343780828198727685746
868895933854159127737893901361659668312456310417704047627550911349966
038674613775530683936729710 89
```

while the second prime q is

```
146850205873335726353330180166772564286621171971217974278132114803544
633278883088179813065620624727347435602771696568487328866768380168920
948824246710591481125265992053947636669578255218889862032915114553154
712140400684563029286007062102121891779229500755653460774642508504472
364358317500526669535243844 77
```

The modulus N is

```
247275513658878801282328324362350170948439822271881115996466922895517
807260015182099550762034157495802121401603618805024611698302324653663
199621879572079305510117840995606570705929526769807088797227393314380
109903951991268766169296050971494455011008075661234718250081008996571
549718564078664213764634179879033436138140820578780413753205069147148
084939839606509988605383846946473227492791610546564145099648941715024
055673231160683748583516668407723785175342146593463329472390278141765
698337558832260644164913350488796927586698227618400410726766590304319
072291398666071369881349881441299500925076152870541385453
```

You can verify that pq is indeed N.

Exercise 3.12.2. Write a simple python program to convert the HEX data from the above into an integer and print the integer. Combine with the openssl command, write a program that prints the integer values from the PEM file in base 10.

Exercise 3.12.3. You now have everything you need to write a python program to generate RSA keys (both private and public). Allow the user to specify the bit length of the key (that means the bit length of N) and the number of rounds of Miller-Rabin. Default the bit length to 4096 and the number of rounds of Miller-Rabin to 128. You can choose your p and q this way: Assume p and q have $n/2$ bits each where n is the bitlength of N. Put random bits into p and q. Obvously you want p and q to be odd and if you want p to be $n/2$ bits then the $n/2$–order bit can't be 0. You then test if the p and q are prime using Miller-Rabin. You can also hard-code the testing for divisibility with a small number of primes: say the first 1,000,000 primes.

Next, test your RSA by encrypting and decrypting an integer M that is $< N$. (There are other conditions on the various quantities to make your RSA key

generator more secure.)

Exercise 3.12.4. In practice, RSA does not use $e, d \pmod{\phi(pq)}$ but $e, d \pmod{\lambda(pq)}$. where $\lambda(pq) = \text{LCM}(p-1, q-1)$. (Here LCM is "largest common multiple.) Study why.

File: fermat-factorization.tex

3.13 Fermat factorization

Suppose you want to factorize n and you know that n is a difference of two squares:

$$n = x^2 - y^2$$

then right away you know

$$n = (x + y)(x - y)$$

Of course if $x - y$ is 1, the factorization is not helpful at all.

To achieve the goal of writing n as a difference of squares, you can do the following:

- Check if $1^2 - n$ is a square.
- Check if $2^2 - n$ is a square.
- Check if $3^2 - n$ is a square.
- Etc.

Why? Because if $x^2 - n$ is a square, say y^2, then $x^2 - n = y^2$ gives us

$$n = x^2 - y^2$$

Note that if $n = ab$ is odd (which is the case for an RSA modulus), then writing n as a difference of two squares since

$$n = \left(\frac{a+b}{2}\right)^2 - \left(\frac{a-b}{2}\right)^2$$

If n is odd, then a and b are odd and therefore $a + b$ and $a - b$ are both even. Hence every odd n can be expressed as the difference of two squares. What if n is even? Then you can remove 2^k from n where k is maximal to get $n = 2^k n'$ and proceed with the above idea applied to n' if $n' > 1$. In the following, I will assume n is odd.

Now we do not want the factorization $n = n \times 1$. Since we are aiming for $n = (x - y)(x + y)$, one way is to start x at around $\sqrt{n}$. Then y will be close to 0 and therefore $x + y$ and $x - y$ won't be close to 1.

Another thing to note is that if we start x at $\sqrt{n}$, if n is squarefree (which is the case for RSA), we will have $x = \lfloor \sqrt{n} \rfloor < \sqrt{n}$, which means $x^2 - n$ is

negative and therefore cannot be y^2. In this case, we might as well start with $x = \lfloor \sqrt{n} \rfloor + 1$.

To find x, y such that $n = x^2 - y^2$, we try different values of x and then check if

$$x^2 - n$$

is a square:

$$x^2 - n = y^2$$

Note that if n is itself a square (this won't happen for the RSA case), then the above will end with $y = 0$.

```
ALGORITHM: Fermat-Factorization
INPUT: n -- an odd integer

let x = floor(sqrt(n)) + 1
while x * x - n is not a square:
    x = x + 1

y = sqrt(x * x - n)
return (x - y, x + y)
```

Here's an example execution of my code for $n = 11 \cdot 11 \cdot 37$:

```
fermatfactorize: n, x, x*x - n, y, x*x - n - y*y = 4477 67 12 3 3
fermatfactorize: n, x, x*x - n, y, x*x - n - y*y = 4477 68 147 12 3
fermatfactorize: n, x, x*x - n, y, x*x - n - y*y = 4477 69 284 16 28
fermatfactorize: n, x, x*x - n, y, x*x - n - y*y = 4477 70 423 20 23
fermatfactorize: n, x, x*x - n, y, x*x - n - y*y = 4477 71 564 23 35
fermatfactorize: n, x, x*x - n, y, x*x - n - y*y = 4477 72 707 26 31
fermatfactorize: n, x, x*x - n, y, x*x - n - y*y = 4477 73 852 29 11
fermatfactorize: n, x, x*x - n, y, x*x - n - y*y = 4477 74 999 31 38
fermatfactorize: n, x, x*x - n, y, x*x - n - y*y = 4477 75 1148 33 59
fermatfactorize: n, x, x*x - n, y, x*x - n - y*y = 4477 76 1299 36 3
fermatfactorize: n, x, x*x - n, y, x*x - n - y*y = 4477 77 1452 38 8
fermatfactorize: n, x, x*x - n, y, x*x - n - y*y = 4477 78 1607 40 7
fermatfactorize: n, x, x*x - n, y, x*x - n - y*y = 4477 79 1764 42 0
fermatfactorize: factors = 37 121
```

Note that Fermat factorization need not be faster than trial division, but it is specifically crafted to handle the product of two primes which are "close".

I'll give you one optimization:

Exercise 3.13.1. Here's a slight optimization: You need to compute x^2 and then $(x+1)^2$, etc. Clearly $(x+1)^2 = x^2 + 2x + 1$. So if you have x and x^2, what's a faster way to compute $(x+1)^2$? Therefore if you have x and $x^2 - n$, how would you compute $(x+1)^2 - n$?

Exercise 3.13.2. We also need to compute $\lfloor \sqrt{n} \rfloor$, i.e., integer square root of n. How many algorithms can you come up with?

Exercise 3.13.3. Implement Fermat factorization and try to optimize it as much as you can. (There are some questions below on optimization.) Test for many values of n. Make a plot. What is the runtime of Fermat factorization algorithm based on your experiment? You don't have to completely factorize n. Finding a factor is good enough. Can you get roughly $O(n^{1/2})$? $O(n^{1/3})$? $O(n^{1/4})$?

Exercise 3.13.4. Implement brute force trial-by-division until $\sqrt{n}$ for n to find a factor of n. Compare the runtimes for this algorithm against your implementation of Fermat factorization above. Which one is faster? Remember that brute force performs $\lfloor \sqrt{n} \rfloor$ mods in the worse case scenario.

Exercise 3.13.5. If n is an integer, how do you handle the check "n is a square"? Can you think of several ways to do that? What are their runtimes?

Exercise 3.13.6. Suppose z^2 is an integer square that is the largest integer square that is $\leq$ to $x^2 - n$. For instance if $x^2 - n$ is 26, then z^2 is 25 (i.e., z is 5). In other words z^2 is the square that is closest to $x^2 - n$ on the left. Note z^2 will be $x^2 - n$ if $x^2 - n$ is a square. How would you compute the the square closest to $(x+1)^2 - n$ on the left? Is it $(z+1)^2$? $(z+2)^2$? If you can prove, for instance, that it's $(z+2)^2$? then we can save on taking square root, correct? You can assume that you have the values of x and z and n.

Exercise 3.13.7. Base on $x^2 - n$ and $(x+1)^2 - n$, is there a way to guestimate k such that $(x+k)^2 - n$ is a square or closed to a square?

Exercise 3.13.8. Suppose n is an odd prime. What will happen during the execution of Fermat's factorization algorithm with n as input? Would the algorithm terminate? (If not, then it's not really an algorithm!) Can Fermat factorization algorithm be converted to a primality testing algorithm? Would it be possible? Would it be a good idea?

Exercise 3.13.9. Write a parallel program for the Fermat factorization algorithm.

Exercise 3.13.10. What if you test for the condition $n = x^3 - y^3$? Is there are version of factorization using difference of cubes?

Chapter 4

Group theory

4.1 Definitions

The most basic mathematical object is $\mathbb{Z}$. $\mathbb{Z}$ has two operations: addition and multiplication. We first abstract the study of $\mathbb{Z}$ by focusing on just one operation, the $+$.

Definition 4.1.1. $(G, *, e)$ is a **group** if G is a set and $*$ satisfies group

(C) If $x, y \in G$, then $x * y \in G$. In other words $* : G \times G \to G$ is a binary operator.
(A) If $x, y, z \in G$, then $(x * y) * z = x * (y * z)$.
(I) If $x \in G$, then there is some $y \in G$ such that $x * y = e = y * x$. y is called an **inverse** of x. Later we will see that the inverse of x is uniquely determined by x. inverse
(N) If $x \in G$, then $x * e = x = e * x$.

Definition 4.1.2. $(G, *, e)$ is an **abelian group** if $(G, *, e)$ is a group such that if $x, y \in G$, then $x * y = y * x$. In other words, $(G, *, e)$ is an abelian group if $(G, *, e)$ is group and $*$ is a commutative operator. abelian group

The reason for now including the commutativity condition in the definition for groups is because there are many important groups which are not abelian.

Chapter 5

Ring theory

Chapter 6

Field theory

Index

Bibliography

[1] Leslie Lamport, *LaTeX: a document preparation system*, Addison Wesley, Massachusetts, 2nd edition, 1994. (EXAMPLE)