PROGRAMMING BITCOIN

PROGRAMMING BITCOIN USING RUST LANGUAGE

Abstract. This book will teach you the technology of Bitcoin at a fundamental level. It doesn't cover the monetary, economic, or social dynamics of Bitcoin, but knowing how Bitcoin works under the hood should give you greater insight into what's possible.
There's a tendency to hype Bitcoin and blockchain without really understanding what's going on; this book is meant to be an antidote to that tendency. After all, there are lots of books about Bitcoin, covering the history and the economic aspects and giving technical descriptions. The aim of this book is to get you to understand Bitcoin by coding all of the components necessary for a Bitcoin library. Thelibrary is not meant to be exhaustive or efficient. The aim of the library is to help you
learn.

Introduction

This book will teach you how to program Bitcoin from scratch using the The original Book by Jimmy Song has code written in Python. This book rewrites that code in Rust Programming Language.

What you need to know?

This book assumes you have installed the Rust Programming Language stable toolchain (at least version 1.75) and that you have read through the Rust programming language book. Install the Rust stable toolchain from . You can find the book at .

RECOMMENDATION

Install Sccache crate to improve developer productivity by caching files that don't need to be rebuilt. Install `sccache` from .

1. Finite Fields

One of the most difficult things about learning how to program Bitcoin is knowing where to start. There are so many components that depend on each other that learning one thing may lead you to have to learn another, which in turn may lead you to need to learn something else before you can understand the original thing.

This chapter is going to get you off to a more manageable start. It may seem strange, but we'll start with the basic math that you need to understand elliptic curve cryptography. Elliptic curve cryptography, in turn, gives us the signing and verification algorithms. These are at the heart of how transactions work, and transactions are the atomic unit of value transfer in Bitcoin. By learning about finite fields and elliptic curves first, you'll get a firm grasp of concepts that you'll need to progress logically.

Be aware that this chapter and the next two chapters may feel a bit like you're eating vegetables, especially if you haven't done formal math in a long time. I would encourage you to get through them, though, as the concepts and code presented here will be used throughout the book.

1.1. **Learning Higher-Level Math.**

Learning about new mathematical structures can be a bit intimidating, and in this chapter, I hope to dispel the myth that high-level math is difficult. Finite fields, in particular, don't require all that much more in terms of prior mathematical knowledge than, say, algebra. Think of finite fields as something that you could have learned instead of trigonometry, except that the education system you're a part of decided that trigonometry was more important for you to learn. This is my way of telling you that finite fields are not that hard to learn and require no more background than algebra.

1.2. **Finite Field Definition.**

Mathematically, a finite field is defined as a finite set of numbers and two operations + (addition) and · (multiplication) that satisfy the following:

1. If a and b are in the set, a + b and a · b are in the set. We call this property **closed**.
2. 0 exists and has the property a + 0 = a. We call this the **additive identity**.
3. 1 exists and has the property a · 1 = a. We call this the **multiplicative identity**.
4. If a is in the set, –a is in the set, which is defined as the value that makes a + (–a) = 0. This is what we call the **additive inverse**.
5. If a is in the set and is not 0, a–1 is in the set, which is defined as the value that makes a · a–1 = 1. This is what we call the **multiplicative inverse**.

Let's unpack each of these criteria.

We have a set of numbers that's finite. Because the set is finite, we can designate a number p, which is how big the set is. This is what we call the order of the set.

`#1` says we are closed under addition and multiplication. This means that we have to define addition and multiplication in a way that ensures the results stay in the set. For example, a set containing `{0,1,2}` is not closed under addition, since

$$1 + 2 = 3$$

and 3 is not in the set; neither is

$$2 + 2 = 4$$

Of course we can define addition a little differently to make this work, but using "normal" addition, this set is not closed. On the other hand, the set

$$\{1, 0, 1\}$$

is closed under normal multiplication. Any two numbers can be multiplied (there are nine such combinations), and the result is always in the set.

The other option we have in mathematics is to define multiplication in a particular way to make these sets closed. We'll get to how exactly we define addition and multiplication later in this chapter, but the key concept here is that we can define addition and subtraction differently than the addition and subtraction you are familiar with.

`#2` and `#3` mean that we have the additive and multiplicative identities. That means 0 and 1 are in the set.

`#4` means that we have the additive inverse. That is, if a is in the set, –a is in the set. Using the additive inverse, we can define subtraction.

`#5` means that multiplication has the same property. If a is in the set, a–1 is in the set. That is

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

Using the multiplicative inverse, we can define division. This will be the trickiest to define in a finite field.

1.3. **Defining Finite Sets.**

If the order (or size) of the set is p, we can call the elements of the set,

$$0, 1, 2, ...p-1$$

These numbers are what we call the elements of the set, not necessarily the traditional numbers 0, 1, 2, 3, etc. They behave in many ways like traditional numbers, but have some differences in how we add, subtract, multiply, and so forth.

In math notation the finite field set looks like this:

$$F_p = \{0, 1, 2, ...p-1\}$$

What's in the finite field set are the elements. F_p is a specific finite field called "field of p" or "field of 29" or whatever the size of it is (again, the size is what mathematicians call order). The numbers between the {}s represent what elements are in the field. We name the elements 0, 1, 2, etc. because these names are convenient for our purposes.

A finite field of order 11 looks like this:

$$F_{11} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$$

A finite field of order 17 looks like this:

$$F_{17} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16\}$$

A finite field of order 983 looks like this:

$$F_{983} = \{0, 1, 2, ...982\}$$

Notice the order of the field is always 1 more than the largest element. You might have noticed that the field has a prime order every time. For a variety of reasons that will become clear later, it turns out that fields must have an order that is a power of a prime, and that the finite fields whose order is prime are the ones we're interested in.

1.3.1. Constructing a Finite Field in Rust.

1.3.2. Handling Rust Errors.

Rust language allows us to enforce error handling by returning the `Result` type. Let's create this Result type in Rust types.

```rust
// Import `fmt` method that allows us to format Rust code.
// Using `core` instead of `std` allows us to use this even
// in `no_std` context.
use core::fmt;

// Create a `Result` type that can be reused
// rather than having to type `-> Result<T, BtcError>`
pub type BtcResult<T> = Result<T, BtcError>;

/// Our Error type to handle returning errors
/// as the same type when we use `?`
#[derive(PartialEq, Eq)]
pub enum BtcError {
    /// Error caused by integer overflow
    IntegerOverflow
}

impl fmt::Debug for BtcError {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        let error_as_str = match self {
          Self::IntegerOverflow => "An integer overflow ocurred",
    };

        write!(f, "{}", error_as_str)
    }
}
```

We want to represent each finite field element, so in Rust, we'll be creating a struct that represents a single finite field element. Naturally, we'll name the struct FieldElement. The struct represents an element in a field $\mathbf{F}_{\text{prime}}$. The bare bones of the struct look like this:

```rust
#[derive(Debug, PartialEq, Eq, Clone)]
struct FieldElement {
    num: u32,
    prime: u32,
}

impl FieldElement {
    pub fn new(num: u32, prime: u32) -> Self {
        Self { num, prime }
    }

    pub fn is_within_order(&self) -> BtcResult<&Self> {
        // By using a Rust `u32` we always ensure the number cannot be less than 0
        // and we avoid checking `num ⩾ prime || num < 0`
        if self.num >= self.prime {
            return Err(BtcError::NumMustBeLessThanPrimeOrder);
        } else {
            Ok(self)
        }
    }
}
```

1. First we use `#[derive(Debug, PartialEq, Eq)]` to ensure
- we can print the fields of the struct to the using `Debug` the
- we can compare two `FieldElements` to see if they are equal using the `PartialEq, Eq`. In Rust deriving `PartialEq, Eq` on a struct automatically implements equality operations for fields of a struct if the fields are primitives or if their types already implement `PartialEq, Eq` as part of the standard library.

2. We then create the struct FieldElement with the fields num: u32, prime: u32
3. We then initialize the struct by creating a method new() withing the impl block impl FieldElement where the new() method takes two parameters num: u32, prime: u32 and returns Self which is the struct FieldElement
4. Next we create a method is_within_order() which is used to:
- check if the num field is an element within the order of length defined by the prime field and return an error if the outcome is true. Note that since we are using Rust u32 we do not need to check if the `num` field is less than 0

```
if self.num ⩾ self.prime
```

We have also introduced BtcError::NumMustBeLessThanPrimeOrder as the return type so we add that to our error enum

```
...

pub enum BtcError {
    /// The `num` field must be less than the `prime` field
    NumMustBeLessThanPrimeOrder, // ← Add this variant
    ...
}
```

and then we implement debug for that new element

```
impl fmt::Debug for BtcError {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) → fmt::Result {
        let error_as_str = match self {
            ...
          Self::NumMustBeLessThanPrimeOrder ⇒ "The `self.num` field must be less than the
`self.prime` field
    };

        write!(f, "{}", error_as_str)
    }
}
```

We can then use our FieldElement to perform operations

```
fn main() {
  // Define an element using `FieldElement::new()` method
    let field_element_a = FieldElement::new(7, 13);
    let field_element_b = FieldElement::new(6, 13);
    let field_element_c = FieldElement::new(13, 13);

    // Since we implemented equality comparison
    // using `PartialEq` and `Eq` traits
    // using #[derive(PartialEq, Eq)] we can check for
    // equality between two `FieldElement`s returning
    // true or false
    println!("{}", field_element_a == field_element_b);
    assert_ne!(field_element_a, field_element_b);
    println!("{}", field_element_a == field_element_a);
    assert_eq!(field_element_a, field_element_a);

    // We can check if the `FieldElement` field `num`
    // is within the `order` using the
    // method `is_within_order()`
    println!("{:?}", field_element_a.is_within_order());
    println!("{:?}", field_element_b.is_within_order());
    println!("{:?}", field_element_c.is_within_order());
}
```

1.4. **Modulo Arithmetic.**

One of the tools we can use to make a finite field closed under addition, subtraction, multiplication, and division is something called modulo arithmetic. We can define addition on the finite set using modulo arithmetic, which is something you probably learned when you first learned division.

$$7 \div 3 = 2R1$$

Whenever the division wasn't even, there was something called the "remainder," which is the amount left over from the actual division. We define modulo in the same way. We use the operator % for modulo:

```
7 % 3 = 1
```

Another example:

$$27 \div 7 = 3R6$$

Formally speaking, the modulo operation is the remainder after division of one number by another. Let's look at another example with larger numbers:

```
1747 % 241 = 60
```

If it helps, you can think of modulo arithmetic as "wraparound" or "clock" math. Imagine a problem like this: It is currently 3 o'clock.

What hour will it be 47 hours from now?

The answer is 2 o'clock because:

```
(3 + 47) % 12 = 2
```

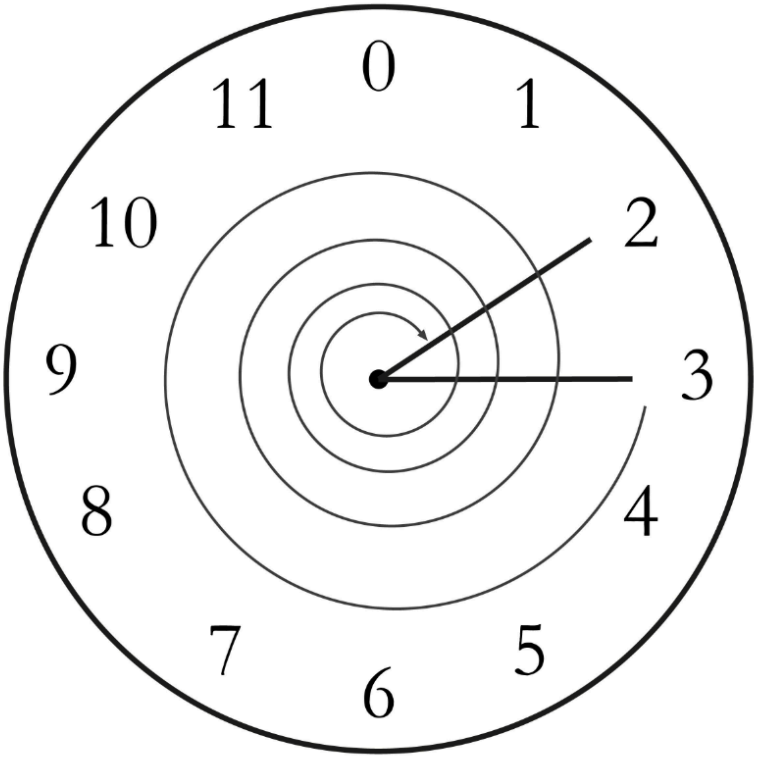

FIGURE 1. Clock going forward 47 hours.

We can also see this as "wrapping around" in the sense that we go past 0 every time we move ahead 12 hours.

We can perform modulo on negative numbers. For example, you can ask:

It is currently 3 o'clock. What hour was it 16 hours ago?

The answer is 11 o'clock:

```
(3 - 16) % 12 = 11
```

The minute hand is also a modulo operation. For example, you can ask:

It is currently 12 minutes past the hour. What minute will it be 843 minutes from now?

It will be 15 minutes past the hour:

```
(12 + 843) % 60 = 15
```

Likewise, we can ask:

It is currently 23 minutes past the hour. What minute will it be 97 minutes from now?

In this case, the answer is 0:

```
(23 + 97) % 60 = 0
```

0 is another way of saying there is no remainder.

The result of the modulo (%) operation for minutes is always between 0 and 59, inclusive. This happens to be a very useful property as even very large numbers can be brought down to a relatively small range with modulo:

```
14738495684013 % 60 = 33
```

We'll be using modulo as we define field arithmetic. Most operations in finite fields use the modulo operator in some capacity.

1.4.1. Modulo Arithmetic in Python.

Rust uses the integer method `rem_euclid()` operator for modulo arithmetic. Here is how the modulo operator is used:

```
println!("{}", 7u32.rem_euclid(3));
```

Remember that in Rust we have to specify the integer type, eg 7.rem_euclid(3) would throw the error:

```
error[E0689]: can't call method `rem_euclid` on ambiguous numeric type `{integer}` ...
you must specify a concrete type for this numeric value, like `i32`
```

We can also use the modulo operator on negative numbers, like this:

```
println!("{}", (-27i64).rem_euclid(13));
```

1.5. **Finite Field Addition and Subtraction.**

Remember that we need to define finite field addition such that we ensure the result is still in the set. That is, we want to make sure that addition in a finite field is closed.

We can use what we just learned, modulo arithmetic, to make addition closed. Let's say we have a finite field of 19:

$$F_{19} = \{0, 1, 2, ...18\}$$

where a, b ϵ F_{19}. Note that the symbol ϵ means "is an element of." In our case, a and b are elements of F_{19}.

Addition being closed means:

$$a +_f b \in F_{19}$$

We denote finite field addition with $+_f$ to avoid confusion with normal integer addition, $+$. If we utilize modulo arithmetic, we can guarantee this to be the case. We can define $a+_f b$ this way:

$$a +_f b = (a + b)\%19$$

For example:

$$7 +_f 8 = (7 + 8)\%19 = 15$$
$$11 +_f 17 = (11 + 17)\%19 = 9$$

and so on.

We take any two numbers in the set, add, and "wrap around" the end to get the sum. We are creating our own addition operator here and the result is a bit unintuitive. After all, $11+_f 17 = 9$ just doesn't look right because we're not used to finite field addition.

More generally, we define field addition this way:

$$a +_f b = (a + b)\%p$$

where $a, b \in F_p$.

We also define the additive inverse this way. $a \in F_p$ implies that $-_f a \in F_p$:

$$-_f a = (-a)\%p$$

Again, for clarity, we use $-_f$ to distinguish field subtraction and negation from integer subtraction and negation. In F_{19}:

$$-_f 9 = (-9)\%19 = 10$$

which means that:

$$9 +_f 10 = 0$$

And that turns out to be true.

Similarly, we can do field subtraction:

$$a -_f b = (a-b)\%p$$

where $a, b \in F_p$. In F_{19}:

$$11 -_f 9 = (11 - 9)\%19 = 2$$
$$6 -_f 13 = (6 - 13)\%19 = 12$$

and so on.

1.5.1. Exercise.

Solve these problems in F_{57} (assume all $+$'s here are $+_f$ and $-$'s here are $-_f$):

1. 44 + 33
2. 9 – 29
3. 17 + 42 + 49
4. 52 – 30 – 38

1.5.2. Coding Addition and Subtraction in Rust.

Rust allows us to override the trait `std::ops::Add` with our own custom implementation that would allow as to add `FieldElement + FieldElement` together. If we try to add two `FieldElements` together without implementing this we get the error:

```
cannot add `FieldElement` to `FieldElement` ... an implementation of `Add` might be missing
for `FieldElement`
```

Let's implement this trait

```
impl std::ops::Add for FieldElement {
    /// We want to return an error if the `order` of both sets is not equal
    type Output = BtcResult<Self>;

    fn add(self, other: Self) -> BtcResult<Self> {
        // We have to ensure that the elements are from the same finite field,
        //otherwise this calculation doesn't have any meaning
        if self.prime != other.prime {
            return Err(BtcError::PrimeOrderMustBeEqual);
        }

        // Addition in a finite field is defined with the modulo operator, as explained
earlier.
        // We return an instance of [Self], which we can conveniently access.
        // However, in Rust we enforce the return type by wrapping our [Self] as part of
the Result using `Ok()`
        Ok(Self {
            num: (self.num + other.num).rem_euclid(self.prime),
            prime: self.prime,
        })
    }
}
```

We are now returning return Err(BtcError::PrimeOrderMustBeEqual); so we add it to our error type by extending the current `BtcError` enum with variant PrimeOrderMustBeEqual

```
...
pub enum BtcError {
    ...

    /// Return this error if we are trying to add
    /// two `FieldElement`s that are not part of
    /// the same order
    PrimeOrderMustBeEqual,
}
```

We also want to implement `fmt::Debug` for the new `PrimeOrderMustBeEqual` enum variant so we extend the `impl fmt::Debug for BtcError` trait

```
impl fmt::Debug for BtcError {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        let error_as_str = match self {
         ...
         Self::PrimeOrderMustBeEqual => "Cannot add two numbers in different Fields. Prime
numbers must be equal in field `self.prime`"
          };
        ...
}
```

1.5.3. Exercise 3.

Write the corresponding std::ops::Sub method that defines the subtraction of two `FieldElement` objects.

```
impl std::ops::Sub for FieldElement {
    /// We want to return an error if the `order` of both sets is not equal
    type Output = BtcResult<Self>;

    fn sub(self, other: Self) -> BtcResult<Self> {
        // We have to ensure that the elements are from the same finite field,
        //otherwise this calculation doesn't have any meaning
        if self.prime != other.prime {
            return Err(BtcError::PrimeOrderMustBeEqual);
        }


        Ok(Self {
            num: (self.num - other.num).rem_euclid(self.prime),
            prime: self.prime,
        })
    }
}
```

1.6. **Finite Field Multiplication and Exponentiation.**

Just as we defined a new addition $(+_f)$ for finite fields that was closed, we can also define a new multiplication for finite fields that's closed. By multiplying the same number many times, we can also define exponentiation. In this section, we'll go through exactly how to define this using modulo arithmetic.

Multiplication is adding multiple times:

$$5 \cdot 3 = 5 + 5 + 5 = 15$$
$$8 \cdot 17 = 8 + 8 + 8 + \ldots \text{(17 total 8's)} \ldots + 8 = 136$$

We can define multiplication on a finite field the same way. Operating in F_{19} once again:

$$5 \cdot_f 3 = 5 +_f 5 +_f 5$$
$$8 \cdot_f 17 = 8 +_f 8 +_f 8 +_f \ldots \text{(17 total 8's)} \ldots +_f 8$$

We already know how to do the right side, and that yields a number within the F_{19} set:

$$5 \cdot_f 3 = 5 +_f 5 +_f 5 = 15 \ \%\ 19 = 15$$
$$8 \cdot_f 17 = 8 +_f 8 +_f 8 +_f \ldots \text{(17 total 8's)} \ldots +_f 8 = (8 \cdot 17) \ \%\ 19 = 136 \ \%\ 19 = 3$$

Note that the second result is pretty unintuitive. We don't normally think of

$$8 \cdot_f 17 = 3$$

but that's part of what's necessary in order to define multiplication to be closed. That is, the result of field multiplication is always in the set $\{0, 1, \ldots p{-}1\}$.

Exponentiation is simply multiplying a number many times:

$$7^3 = 7 \cdot_f 7 \cdot_f 7 = 343$$

In a finite field, we can do exponentiation using modulo arithmetic. In F_{19}:

$$7^3 = 343\%19 = 1$$
$$9^{12} = 7$$

Exponentiation again gives us counterintuitive results. We don't normally think $7^3 = 1$ or $9^{12} = 7$. Again, finite fields have to be defined so that the operations always result in a number within the field.

1.6.1. Exercise.
Solve the following equations in F_{97} (again, assume $\cdot$ and exponentiation are field versions):

- $95 \cdot 45 \cdot 31$
- $17 \cdot 13 \cdot 19 \cdot 44$
- $127 \cdot 7749$

1.6.2. Exercise.
For $k = 1, 3, 7, 13, 18$, what is this set in F_{19}?

$$\{k \cdot 0, k \cdot 1, k \cdot 2, k \cdot 3, ...k \cdot 18\}$$

Do you notice anything about these sets?

Why Fields Are Prime

The answer to the last Exercise is why fields have to have a prime power number of elements. No matter what k you choose, as long as it's greater than 0, multiplying the entire set by k will result in the same set as you started with.

Intuitively, the fact that we have a prime order results in every element of a finite field being equivalent. If the order of the set was a composite number, multiplying the set by one of the divisors would result in a smaller set.

1.6.3. Coding Multiplication in Rust.
Now that we understand what multiplication should be in FieldElement, we want to define the mul() method on struct `FieldElement` that overrides the `std::ops::Mul` trait. We want this to work:

```
let a = FieldElement::new(3, 13);
let b = FieldElement::new(12, 13);
let c = FieldElement::new(10, 13);

assert_eq!((a * b), c);
```

As with addition and subtraction, let's make multiplication work for our class by defining the mul() method on `FieldElement` struct.

```
impl std::ops::Mul for FieldElement {
    /// We want to return an error if the `order` of both sets is not equal
    type Output = BtcResult<Self>;

    fn mul(self, other: Self) -> BtcResult<Self> {
        // We have to ensure that the elements are from the same finite field,
        //otherwise this calculation doesn't have any meaning
        if self.prime != other.prime {
            return Err(BtcError::PrimeOrderMustBeEqual);
        }
```

```
        Ok(Self {
            num: (self.num * other.num).rem_euclid(self.prime),
            prime: self.prime,
        })
    }
}
```

1.6.4. Coding Exponentiation in Rust.

We need to define the exponentiation for FieldElement struct, which in Rust can be implemented by creating a method pow() on the struct. The difference here is that the exponent is not a FieldElement, so it has to be treated a bit differently. We want something like this to work:

```
let a = FieldElement::new(3, 13);
let b = FieldElement::new(1, 13);

let pow_a = a.pow(3).unwrap();
assert_eq!(pow_a, b);
```

Note that because the exponent is an integer, instead of another instance of `FieldElement`, the method receives the variable exponent as an integer. We can code it this way:

```
impl FieldElement {
  pub fn pow(&self, exponent: u32) -> BtcResult<Self> {
        if let Some(exp) = self.num.checked_pow(exponent) {
            let modulo = exp.rem_euclid(self.prime);
            Ok(Self {
                num: modulo,
                prime: self.prime,
            })
        } else {
            Err(BtcError::IntegerOverflow)
        }
    }
}
```

Why don't we force the exponent to be a FieldElement object? It turns out that the exponent doesn't have to be a member of the finite field for the math to work. In fact, if it were, the exponents wouldn't display the intuitive behavior we expect, like being able to add the exponents when we multiply with the same base.

Some of what we're doing now may seem slow for large numbers, but we'll use some clever tricks to improve the performance of these algorithms.

1.6.5. Exercise.

For p = 7, 11, 17, 31, what is this set in F_p?

$$\left\{1^{(p-1)}, 2^{(p-1)}, 3^{(p-1)}, 4^{(p-1)}, ...(p-1)^{(p-1)}\right\}$$

1.7. **Finite Field Division.**

The intuition that helps us with addition, subtraction, multiplication, and perhaps even exponentiation unfortunately doesn't help us quite as much with division.

Because division is the hardest operation to make sense of, we'll start with something that should make sense.

In normal math, division is the inverse of multiplication:

- $7 \cdot 8 = 56$ implies that $56 \div 8 = 7$
- $12 \cdot 2 = 24$ implies that $24 \div 12 = 2$

And so on. We can use this as the definition of division to help us. Note that like in normal math, you cannot divide by 0.

In F_{19}, we know that:

$$3 \cdot_f 7 = 21 \% 19 = 2 \text{ implies that } 2 /_f 7 = 3$$
$$9 \cdot_f 5 = 45 \% 19 = 7 \text{ implies that } 7 /_f 5 = 9$$

This is very unintuitive, as we generally think of $2/_f7$ or $7/_f5$ as fractions, not nice finite field elements. Yet that is one of the remarkable things about finite fields: finite fields are closed under division. That is, dividing any two numbers where the denominator is not 0 will result in another finite field element.

The question you might be asking yourself is, how do I calculate 2/f7 if I don't know beforehand that $3 \cdot_f 7 = 2$? This is indeed a very good question; to answer it, we'll have to use the result from previous Exercise.

In case you didn't get it, the answer is that $n^{(p-1)}$ is always 1 for every p that is prime and every n > 0. This is a beautiful result from number theory called Fermat's little theorem. Essentially, the theorem says:

$$n^{(p-1)} \% p = 1$$

`where p is prime.`

Since we are operating in prime fields, this will always be true.

Fermat's Little Theorem

There are many proofs of this theorem, but perhaps the simplest is using what we saw that these sets are equal:

$$\{1, 2, 3, ...p-2, p-1\} = \{n\%p, 2n\%p, 3n\%p(p-2)n\%p, (p-1)n\%p\}$$

The resulting numbers might not be in the right order, but the same numbers are in both sets. We can then multiply every element in both sets to get this equality:

$$1 \cdot 2 \cdot 3 \cdot ... \cdot (p-2) \cdot (p-1)\%p = n \cdot 2n \cdot 3n \cdot ... \cdot (p-2)n \cdot (p-1)n\%p$$

The left side is the same as (p–1)! % p where ! is the factorial e.g.,

$$5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$$

On the right side, we can gather up all the n`'s and get:

$$(p-1)! \cdot n^{(p-1)}\%p$$

The (p–1)! on both sides cancel, giving us:

$$1 = n^{(p-1)}\%p$$

This proves Fermat's little theorem.

Because division is the inverse of multiplication, we know:

$$\frac{a}{b} = a \cdot_f \left(\frac{1}{b}\right) = a \cdot_f b^{-1}$$

We can reduce the division problem to a multiplication problem as long as we can figure out what b–1 is. This is where Fermat's little theorem comes into play. We know:

$$b^{(p-1)} = 1$$

because p is prime. Thus:

$$b^{-1} = b^{-1} \cdot_f 1 = b^{-1} \cdot_f b^{(p-1)} = b^{(p-2)}$$

or:

$$b^{-1} = b^{(p-2)}$$

In F_{19}, this means practically that $b^{18} = 1$, which means that $b^{-1} = b^{17}$ for all b > 0.

So in other words, we can calculate the inverse using the exponentiation operator. In F_{19}:

$$\frac{2}{7} = 2 \cdot 7^{(19-2)} = 2 \cdot 7^{17} = 465261027974414 \% 19 = 3$$

$$\frac{7}{5} = 7 \cdot 5^{(19-2)} = 7 \cdot 5^{17} = 5340576171875 \% 19 = 9$$

This is a relatively expensive calculation as exponentiating grows very fast. Division is the most expensive operation for that reason. To lessen the expense, we can use the checked_pow() method on an integer in Rust, which does exponentiation and returns `Option::None` in case of integer overflow. In Rust, 7u32.checked_pow(17) is a good example. We then modulo the result.

1.7.1. Exercise.

Solve the following equations in F_{31}:

- $3 / 24$
- 17^{-3}
- $4^{-4} \cdot 11$

1.7.2. Exercise.

Write the corresponding truediv() method on FieldElement struct that defines the division of two field elements.

1.8. **Redefining Exponentiation.**

One last thing that we need to take care of before we leave this chapter is the pow() method, which needs to handle negative exponents. For example, a–3 needs to be a finite field element, but the current code does not take care of this case. We want, for example, something like this to work:

```
let a = FieldElement::new(7, 13);
let b = FieldElement::new(8, 13);
assert!(a.pow(-3).unwrap() == b);
```

Unfortunately, the way we've defined pow() method on FieldElement struct simply doesn't handle negative exponents, because the second parameter of the built-in Rust method checked_pow() on an integer is required to be positive.

Thankfully, we can use some math we already know to solve this. We know from Fermat's little theorem that:

$$a^{p-1} = 1$$

This fact means that we can multiply by a^{p-1} as many times as we want. So, for a^{-3}, we can do:

$$a^{-3} = a^{-3} \cdot a^{p-1} = a^{p-4}$$

To force a number out of being negative, we use rem_euclid() integer method where we subtract 1 from the prime and use it as the parameter for method arguments of the pow() method

```
exponent.rem_euclid((self.prime - 1) as i32) as u32;
```

As a bonus, we can also reduce very large exponents at the same time given that $a^{p-1} = 1$. This will make the pow function not work as hard. Let's change our pow() method on FieldElement to

```
pub fn pow(&self, exponent: i32) -> BtcResult<Self> {
    // Force number out of becoming a negative
    let n = exponent.rem_euclid((self.prime - 1) as i32) as u32;

    if let Some(exp) = self.num.checked_pow(n) {
        let modulo = exp.rem_euclid(self.prime);
        Ok(Self {
            num: modulo,
            prime: self.prime,
        })
    } else {
        Err(BtcError::IntegerOverflow)
    }
}
```

1.9. **Conclusion.** In this chapter we learned about finite fields and how to implement them in Rust. We'll be using finite fields in Chapter 3 for elliptic curve cryptography. We turn next to the other mathematical component that we need for elliptic curve cryptography: elliptic curves

References

URL: