CHAPTER 2

Elliptic Curves

In this chapter we're going to learn about elliptic curves. In Chapter 3, we will combine elliptic curves with finite fields to make elliptic curve cryptography.

Like finite fields, elliptic curves can look intimidating if you haven't seen them before. But again, the actual math isn't very difficult. Most of what you need to know about elliptic curves could have been taught to you after algebra. In this chapter, we'll explore what these curves are and what we can do with them.

Definition

Elliptic curves are like many equations you've seen since pre-algebra. They have y on one side and x on the other, in some form. elliptic curves have a form like this:

$$y^2 = x^3 + ax + b$$

You've worked with other equations that look similar. For example, you probably learned the linear equation back in pre-algebra:

$$y = mx + b$$

You may even remember that m here has the name *slope* and b is the *y-intercept*. You can also graph linear equations, as shown in Figure 2-1.

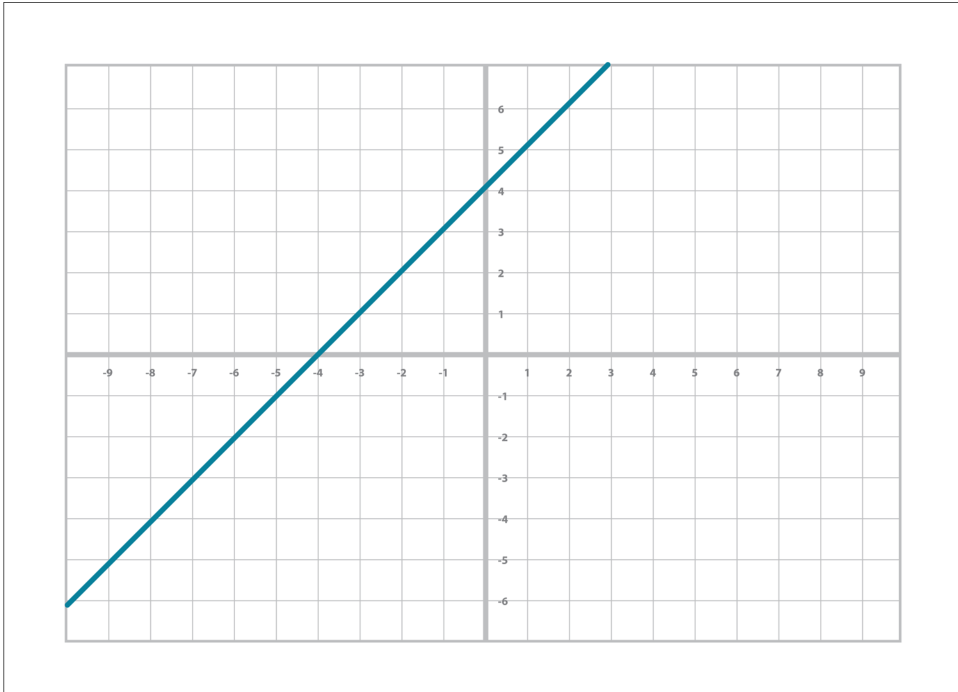


Figure 2-1. Linear equation

Similarly, you're probably familiar with the quadratic equation and its graph (Figure 2-2):

$$y = ax^2 + bx + c$$

And sometime around algebra, you did even higher orders of x—something called the cubic equation and its graph (Figure 2-3):

$$y = ax^3 + bx^2 + cx + d$$

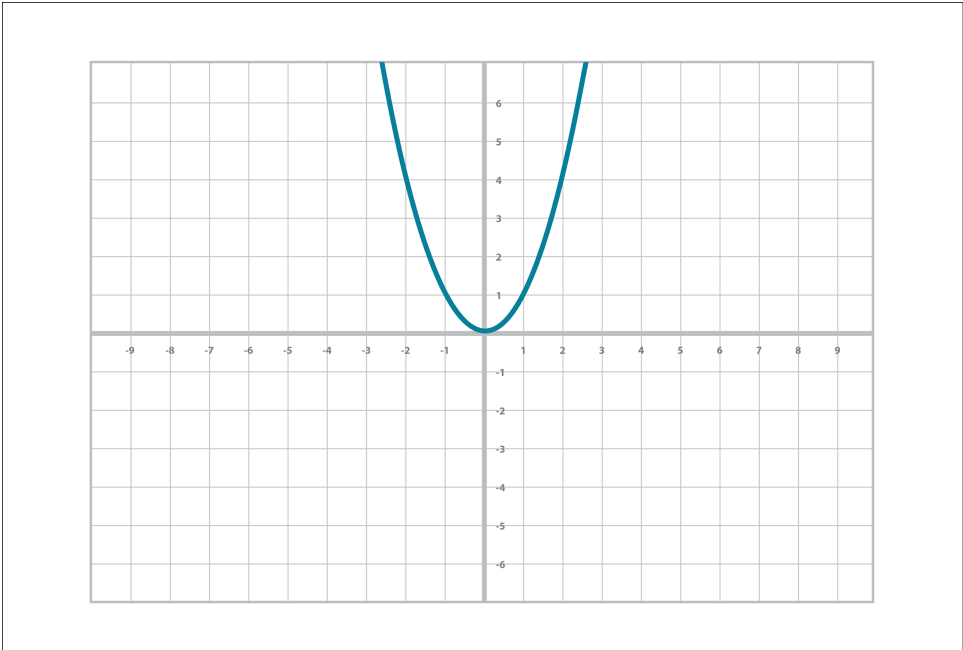


Figure 2-2. Quadratic equation

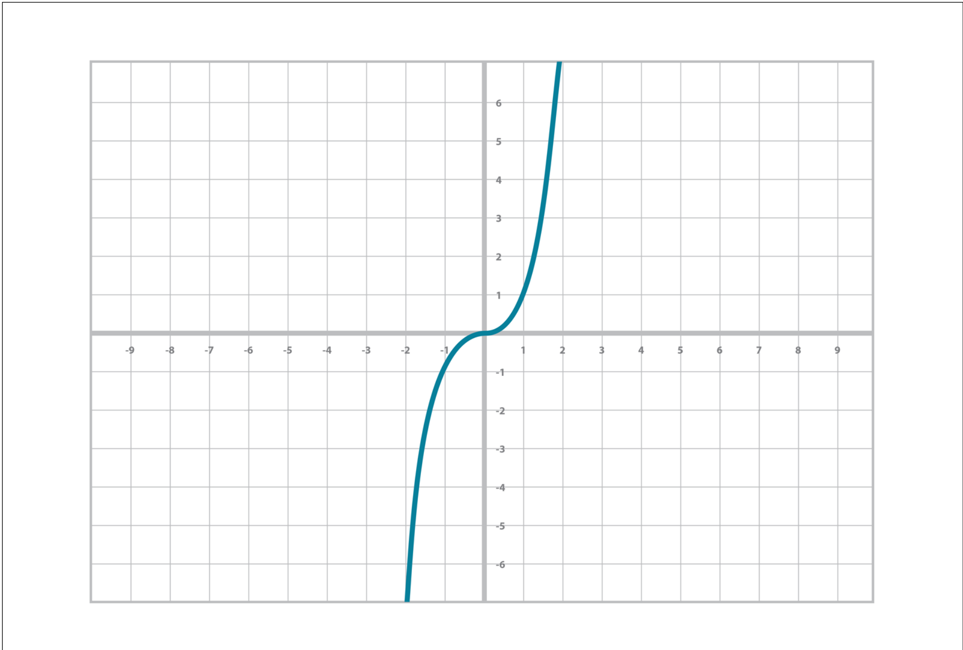


Figure 2-3. Cubic equation

An elliptic curve isn't all that different:

$$y^2 = x^3 + ax + b$$

The only real difference between the elliptic curve and the cubic curve in Figure 2-3 is the y^2 term on the left side. This has the effect of making the graph symmetric over the x-axis, as shown in Figure 2-4.

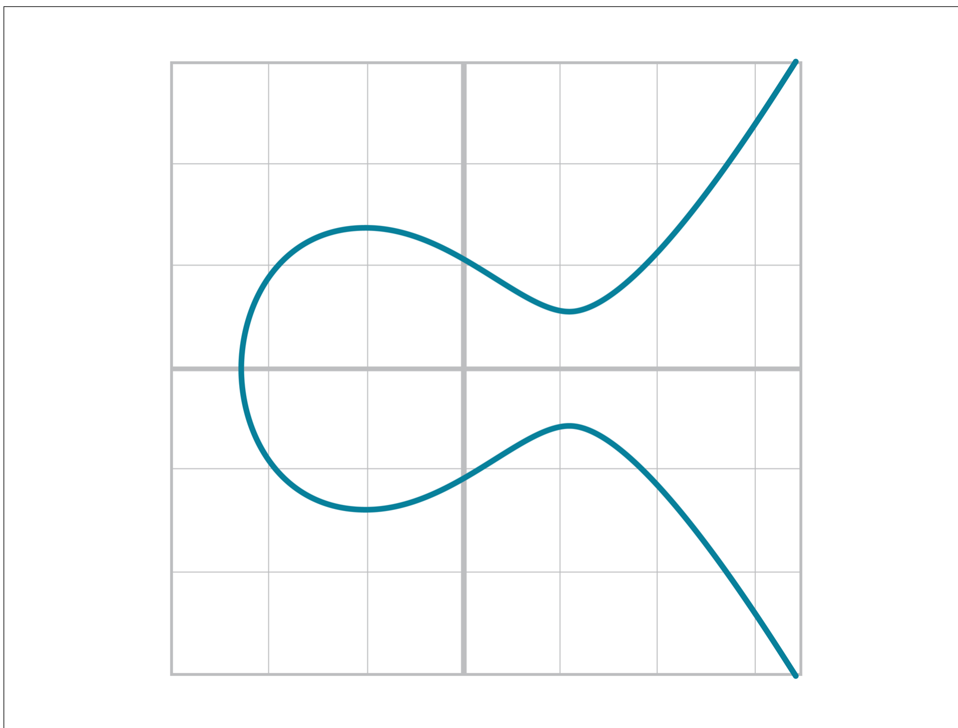


Figure 2-4. Continuous elliptic curve

The elliptic curve is also less steep than the cubic curve. Again, this is because of the y^2 term on the left side. At times, the curve may even be disjoint, as in Figure 2-5.

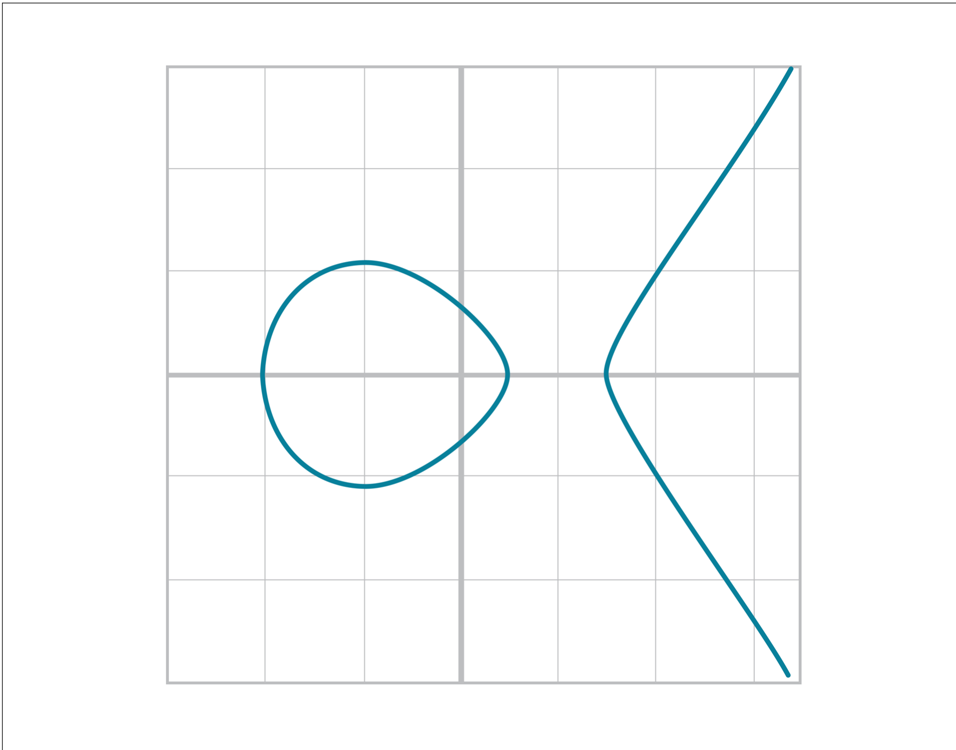


Figure 2-5. Disjoint elliptic curve

If it helps, an elliptic curve can be thought of as taking a cubic equation graph (Figure 2-6), flattening out the part above the x-axis (Figure 2-7), and then mirroring that part below the x-axis (Figure 2-8).

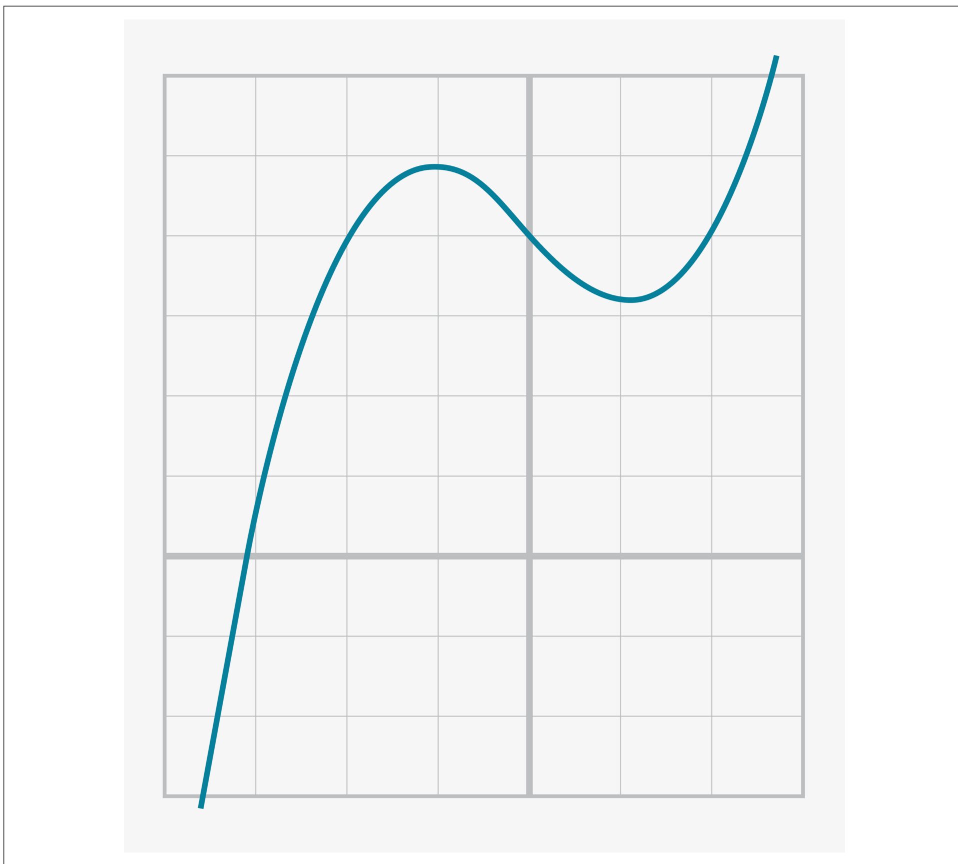


Figure 2-6. Step 1: A cubic equation

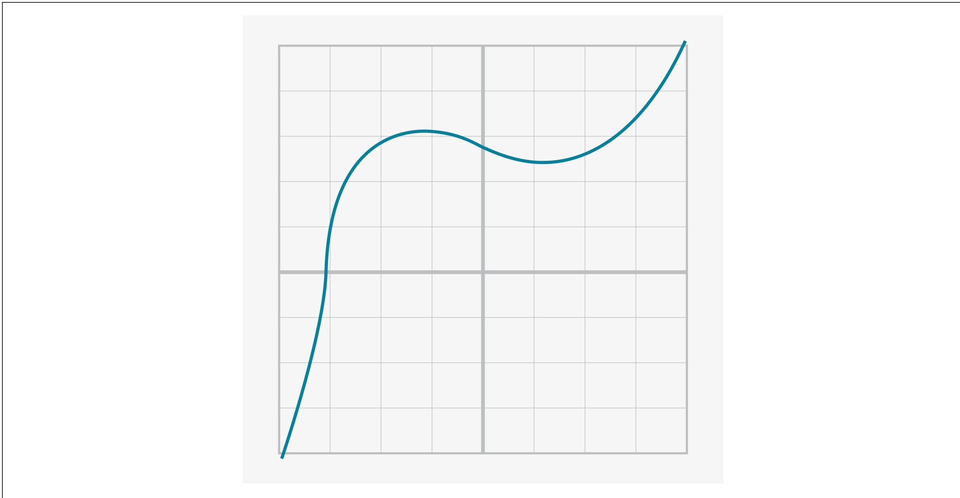


Figure 2-7. Step 2: Stretched cubic equation

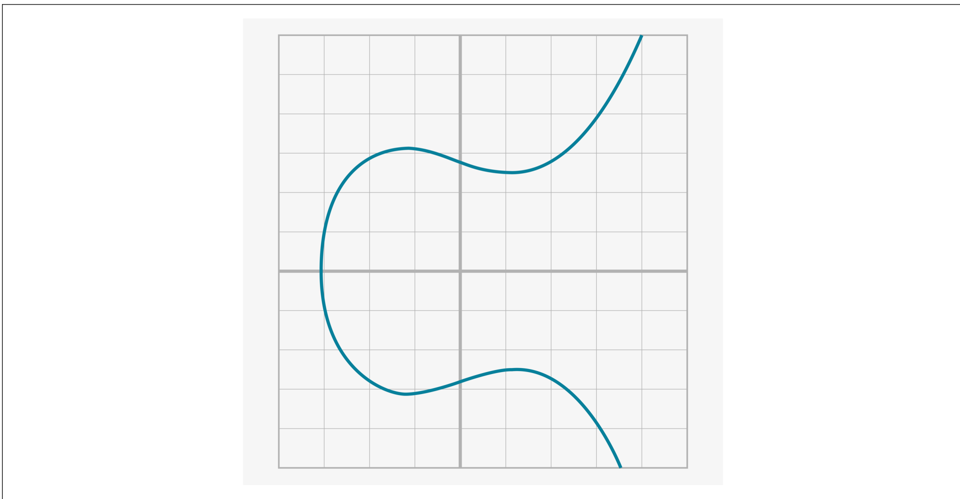


Figure 2-8. Step 3: Reflected over the x-axis

Specifically, the elliptic curve used in Bitcoin is called *secp256k1* and it uses this particular equation:

$$y^2 = x^3 + 7$$

The canonical form is $y^2 = x^3 + ax + b$, so the curve is defined by the constants $a = 0$, $b = 7$. It looks like Figure 2-9.

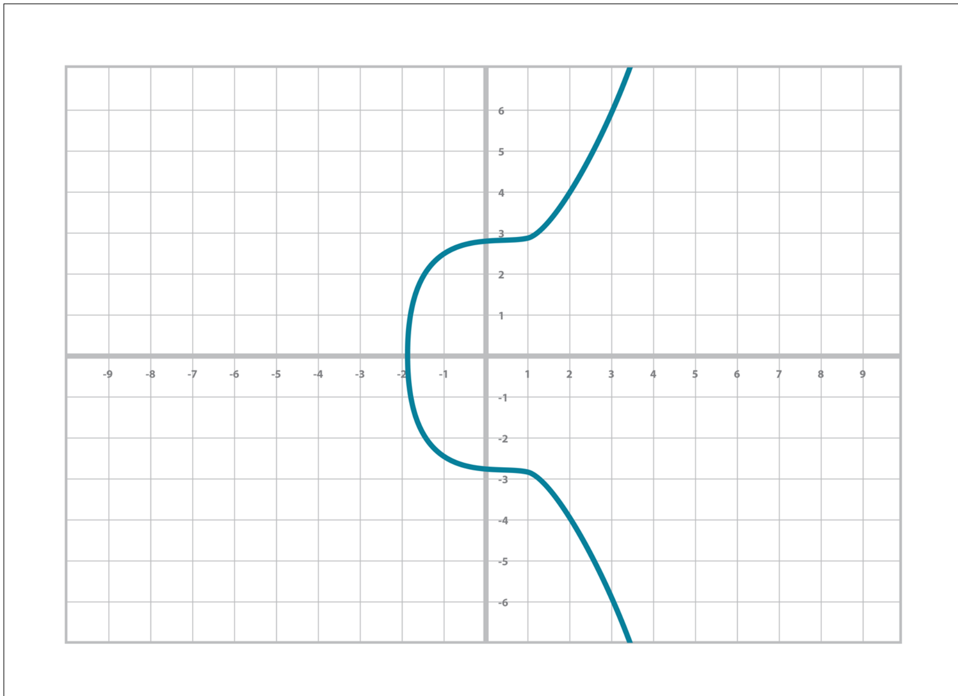


Figure 2-9. secp256k1 curve

Coding Elliptic Curves in Python

For a variety of reasons that will be made clear later, we are not interested in the curve itself, but specific points on the curve. For example, in the curve $y^2 = x^3 + 5x + 7$, we are interested in the coordinate (–1,1). We are thus going to define the class `Point` to be a *point* on a specific curve. The curve has the form $y^2 = x^3 + ax + b$, so we can define the curve with just the two numbers a and b:

```
class Point:

    def __init__(self, x, y, a, b):
        self.a = a
        self.b = b
        self.x = x
        self.y = y
        if self.y**2 != self.x**3 + a * x + b:  ❶
            raise ValueError('({}, {}) is not on the curve'.format(x, y))

    def __eq__(self, other):  ❷
        return self.x == other.x and self.y == other.y \
            and self.a == other.a and self.b == other.b
```

❶ We check here that the point is actually on the curve.

❷ Points are equal if and only if they are on the same curve and have the same coordinates.

We can now create `Point` objects, and we will get an error if the point is not on the curve:

```
>>> from ecc import Point
>>> p1 = Point(-1, -1, 5, 7)
>>> p2 = Point(-1, -2, 5, 7)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "ecc.py", line 143, in __init__
    raise ValueError('({}, {}) is not on the curve'.format(self.x, self.y))
ValueError: (-1, -2) is not on the curve
```

In other words, `__init__` will raise an exception when the point is not on the curve.

Exercise 1

Determine which of these points are on the curve $y^2 = x^3 + 5x + 7$:

$(2,4), (-1,-1), (18,77), (5,7)$

Exercise 2

Write the `__ne__` method for `Point`.

Point Addition

Elliptic curves are useful because of something called *point addition*. Point addition is where we can do an operation on two of the points on the curve and get a third point, also on the curve. This is called *addition* because the operation has a lot of the intuitions we associate with the mathematical operation of addition. For example, point addition is commutative. That is, adding point A to point B is the same as adding point B to point A.

The way we define point addition is as follows. It turns out that for every elliptic curve, a line will intersect it at either one point (Figure 2-10) or three points (Figure 2-11), except in a couple of special cases.

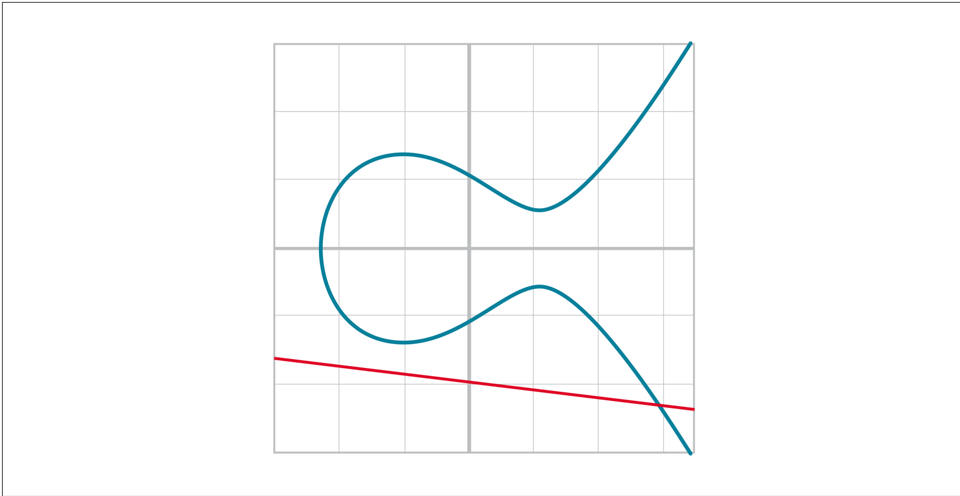


Figure 2-10. Line intersects at only one point

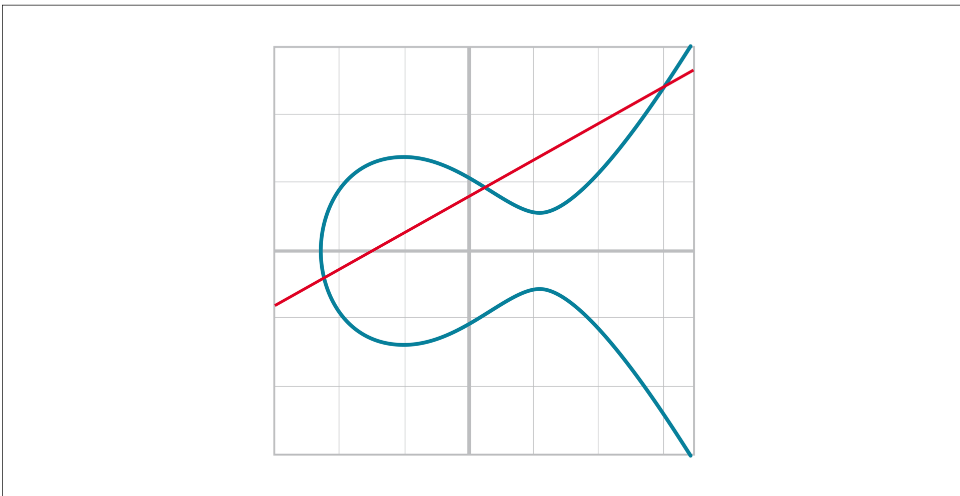


Figure 2-11. Line intersects at three points

The two exceptions are when a line is exactly vertical (Figure 2-12) and when a line is *tangent* to the curve (Figure 2-13).

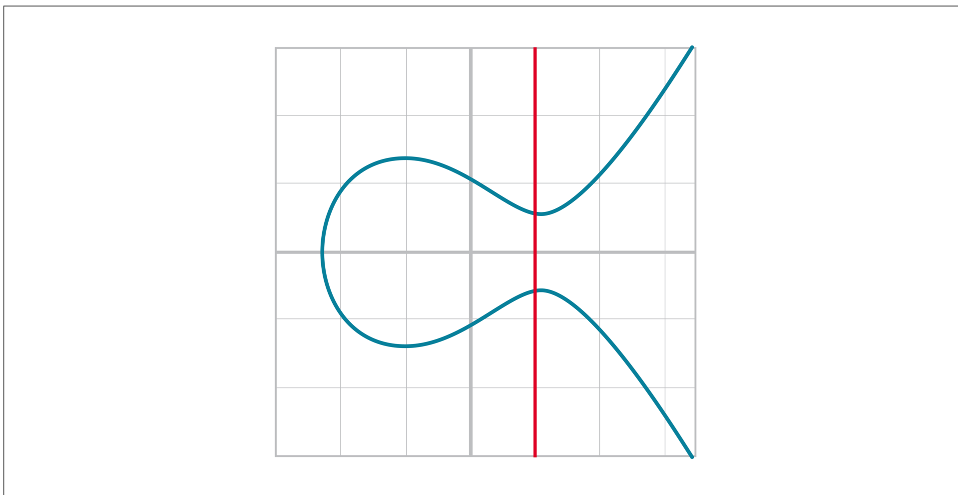


Figure 2-12. Line intersects at two points because it's vertical

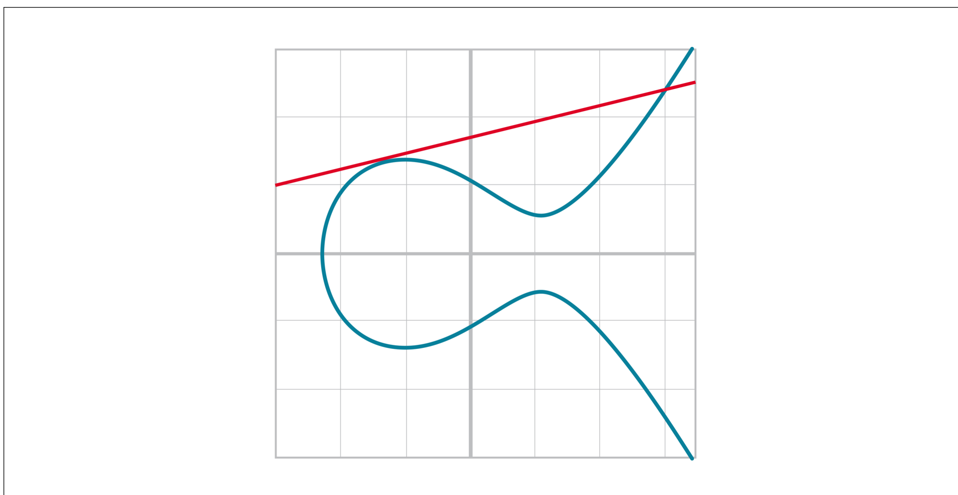


Figure 2-13. Line intersects at two points because it's tangent to the curve

We will come back to these two cases later.

We can define point addition using the fact that lines intersect one or three times with the elliptic curve. Two points define a line, so since that line must intersect the curve one more time, that third point reflected over the x-axis is the result of the point addition.

So, for any two points $P_1 = (x_1,y_1)$ and $P_2 = (x_2,y_2)$, we get $P_1 + P_2$ as follows:

- Find the point intersecting the elliptic curve a third time by drawing a line through P_1 and P_2.
- Reflect the resulting point over the x-axis.

Visually, it looks like Figure 2-14.

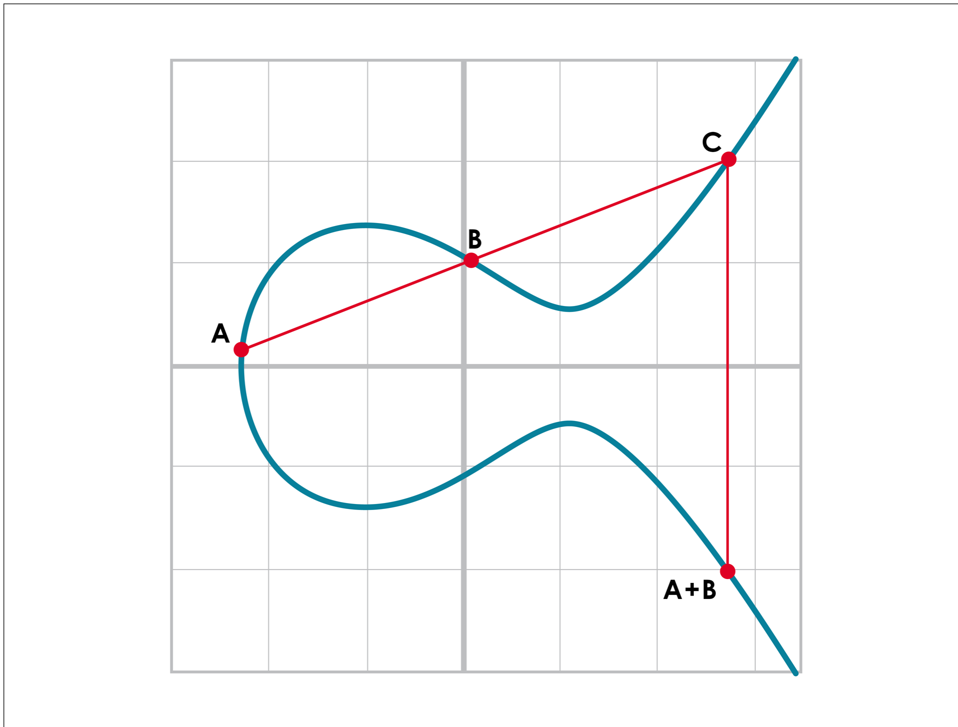


Figure 2-14. Point addition

We first draw a line through the two points we're adding (*A* and *B*). The third intersection point is *C*. We then reflect that point over the x-axis, which puts us at the $A + B$ point in Figure 2-14.

One of the properties that we are going to use is that point addition is not easily predictable. We can calculate point addition easily enough with a formula, but intuitively, the result of point addition can be almost anywhere given two points on the curve. Going back to Figure 2-14, $A + B$ is to the right of both points, $A + C$ would be somewhere between *A* and *C* on the x-axis, and $B + C$ would be to the left of both points. In mathematics parlance, point addition is *nonlinear*.

Math of Point Addition

Point addition satisfies certain properties that we associate with addition, such as:

- Identity
- Commutativity
- Associativity
- Invertibility

Identity here means that there's a zero. That is, there exists some point I that, when added to a point A, results in A:

$$I + A = A$$

We'll call this point the *point at infinity* (reasons for this will become clear in a moment).

This is related to *invertibility*. For some point A, there's some other point $–A$ that results in the identity point. That is:

$$A + (–A) = I$$

Visually, these points are opposite one another over the x-axis on the curve (see Figure 2-15).

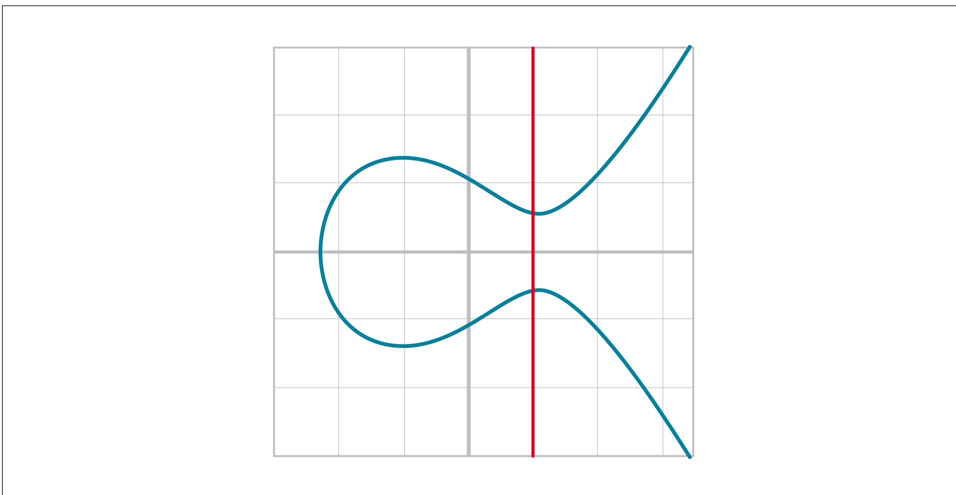


Figure 2-15. Vertical line intersection

This is why we call this point the point at infinity. We have one extra point in the elliptic curve, which makes the vertical line intersect the curve a third time.

Commutativity means that $A + B = B + A$. This is obvious since the line going through A and B will intersect the curve a third time in the same place, no matter the order.

Associativity means that $(A + B) + C = A + (B + C)$. This isn't obvious and is the reason for flipping over the x-axis. This is shown in Figures 2-16 and 2-17.

You can see that in both Figure 2-16 and Figure 2-17, the final point is the same. In other words, we have good reason to believe that $(A + B) + C = A + (B + C)$. While this doesn't prove the associativity of point addition, the visual should at least give you the intuition that this is true.

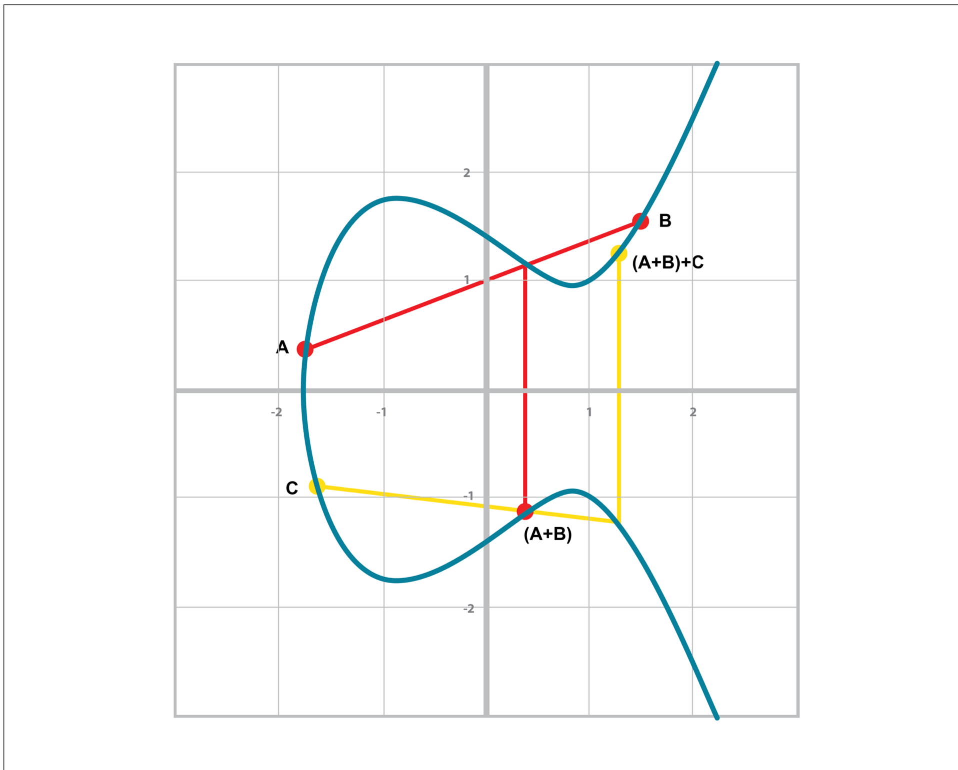


Figure 2-16. (A + B) + C

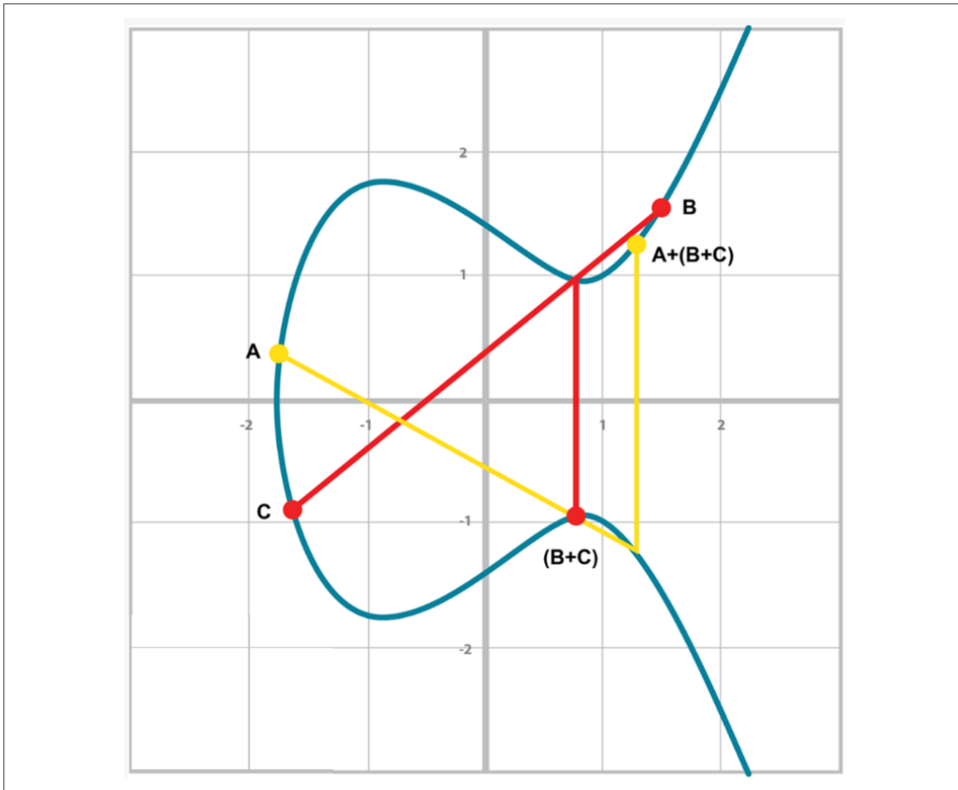


Figure 2-17. A + (B + C)

To code point addition, we're going to split it up into three steps:

1. Where the points are in a vertical line or using the identity point
2. Where the points are not in a vertical line, but are different
3. Where the two points are the same

Coding Point Addition

We first handle the identity point, or point at infinity. Since we can't easily use infinity in Python, we'll use the `None` value instead. What we want is this to work:

```
>>> from ecc import Point
>>> p1 = Point(-1, -1, 5, 7)
>>> p2 = Point(-1, 1, 5, 7)
>>> inf = Point(None, None, 5, 7)
>>> print(p1 + inf)
Point(-1,-1)_5_7
```

```
>>> print(inf + p2)
Point(-1,1)_5_7
>>> print(p1 + p2)
Point(infinity)
```

To make this work, we have to do two things. First, we have to adjust the `__init__` method slightly so it doesn't check that the curve equation is satisfied when we have the point at infinity. Second, we have to overload the addition operator or `__add__` as we did with the `FieldElement` class:

```
class Point:

    def __init__(self, x, y, a, b):
        self.a = a
        self.b = b
        self.x = x
        self.y = y
        if self.x is None and self.y is None:  ❶
            return
        if self.y**2 != self.x**3 + a * x + b:
            raise ValueError('({}, {}) is not on the curve'.format(x, y))

    def __add__(self, other):  ❷
        if self.a != other.a or self.b != other.b:
            raise TypeError('Points {}, {} are not on the same curve'.format
            (self, other))

        if self.x is None:  ❸
            return other
        if other.x is None:  ❹
            return self
```

❶ The x coordinate and y coordinate being `None` is how we signify the point at infinity. Note that the next `if` statement will fail if we don't return here.

❷ We overload the `+` operator here.

❸ `self.x` being `None` means that `self` is the point at infinity, or the additive identity. Thus, we return `other`.

❹ `other.x` being `None` means that `other` is the point at infinity, or the additive identity. Thus, we return `self`.

Exercise 3

Handle the case where the two points are additive inverses (that is, they have the same `x` but a different `y`, causing a vertical line). This should return the point at infinity.

Point Addition for When $x_1 \neq x_2$

Now that we've covered the vertical line, let's examine when the points are different. When we have points where the x's differ, we can add using a fairly simple formula. To help with intuition, we'll first find the slope created by the two points. We can figure this out using a formula from pre-algebra:

$$P_1 = (x_1,y_1),\ P_2 = (x_2,y_2),\ P_3 = (x_3,y_3)$$
$$P_1 + P_2 = P_3$$
$$s = (y_2 - y_1)/(x_2 - x_1)$$

This is the *slope*, and we can use the slope to calculate x_3. Once we know x_3, we can calculate y_3. P_3 can be derived using this formula:

$$x_3 = s^2 - x_1 - x_2$$
$$y_3 = s(x_1 - x_3) - y_1$$

Remember that y_3 is the reflection over the x-axis.

Deriving the Point Addition Formula

Supposing:

$$P_1 = (x_1,y_1),\ P_2 = (x_2,y_2),\ P_3 = (x_3,y_3)$$
$$P_1 + P_2 = P_3$$

We want to know what P_3 is.

Let's start with the fact that the line goes through P_1 and P_2, and has this formula:

$$s = (y_2 - y_1)/(x_2 - x_1)$$
$$y = s(x - x_1) + y_1$$

The second formula is the equation of the line that intersects at both P_1 and P_2. Using this formula and plugging it into the elliptic curve equation, we get:

$$y^2 = x^3 + ax + b$$
$$y^2 = (s(x - x_1) + y_1)^2 = x^3 + ax + b$$

Gathering all the terms, we have this polynomial equation:

$$x^3 - s^2x^2 + (a + 2s^2x_1 - 2sy_1)x + b - s^2x_1{}^2 + 2sx_1y_1 - y_1{}^2 = 0$$

We also know that x_1, x_2, and x_3 are solutions to this equation, thus:

$$(x - x_1)(x - x_2)(x - x_3) = 0$$

$$x^3 - (x_1 + x_2 + x_3)x^2 + (x_1x_2 + x_1x_3 + x_2x_3)x - x_1x_2x_3 = 0$$

From earlier, we know that:

$$x^3 - s^2x^2 + (a + 2s^2x_1 - 2sy\sim1\sim)x + b - s^2x_1^{\ 2} + 2sx\sim1\sim y_1 - y_1^{\ 2} = 0$$

There's a result from what's called Vieta's formula, which states that the coefficients have to equal each other if the roots are the same. The first coefficient that's interesting is the coefficient in front of x^2:

$$-s^2 = -(x_1 + x_2 + x_3)$$

We can use this to derive the formula for x_3:

$$x_3 = s^2 - x_1 - x_2$$

We can plug this into the formula for the line above:

$$y = s(x - x_1) + y_1$$

But we have to reflect over the x-axis, so the right side has to be negated:

$$y_3 = -(s(x_3 - x_1) + y_1) = s(x_1 - x_3) - y_1$$

QED.

Exercise 4

For the curve $y^2 = x^3 + 5x + 7$, what is $(2,5) + (-1,-1)$?

Coding Point Addition for When $x_1 \neq x_2$

We now code this into our library. That means we have to adjust the `__add__` method to handle the case where $x_1 \neq x_2$. We have the formulas:

$$s = (y_2 - y_1)/(x_2 - x_1)$$
$$x_3 = s^2 - x_1 - x_2$$
$$y_3 = s(x_1 - x_3) - y_1$$

At the end of the method, we return an instance of the class `Point` using `self.__class__` to make subclassing easier.

Exercise 5

Write the `__add__` method where $x_1 \neq x_2$.

Point Addition for When $P_1 = P_2$

When the *x* coordinates are the same and the *y* coordinate is different, we have the situation where the points are opposite each other over the x-axis. We know that this means:

$$P_1 = -P_2 \text{ or } P_1 + P_2 = I$$

We've already handled this in Exercise 3.

What happens when $P_1 = P_2$? Visually, we have to calculate the line that's *tangent* to the curve at P_1 and find the point at which the line intersects the curve. The situation looks like Figure 2-18, as we saw before.

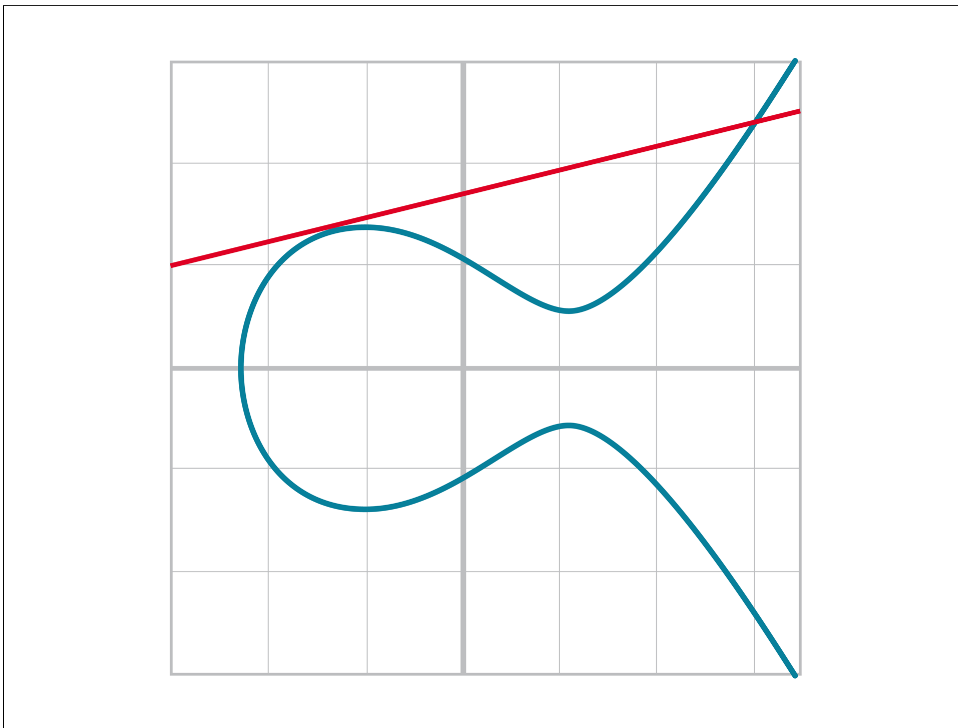


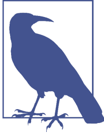
Figure 2-18. Line that's tangent to the curve

Once again, we'll find the slope of the tangent point:

$$P_1 = (x_1,y_1), P_3 = (x_3,y_3)$$
$$P_1 + P_1 = P_3$$
$$s = (3x_1^2 + a)/(2y_1)$$

The rest of the formula goes through as before, except $x_1 = x_2$, so we can combine them:

$$x_3 = s^2 - 2x_1$$
$$y_3 = s(x_1 - x_3) - y_1$$



Deriving the Slope Tangent to the Curve

We can derive the slope of the tangent line using some slightly more advanced math: calculus. We know that the slope at a given point is:

$$dy/dx$$

To get this, we need to take the derivative of both sides of the elliptic curve equation:

$$y^2 = x^3 + ax + b$$

Taking the derivative of both sides, we get:

$$2y\ dy = (3x^2 + a)\ dx$$

Solving for dy/dx, we get:

$$dy/dx = (3x^2 + a)/(2y)$$

That's how we arrive at the slope formula. The rest of the results from the point addition formula derivation hold.

Exercise 6

For the curve $y^2 = x^3 + 5x + 7$, what is $(-1,-1) + (-1,-1)$?

Coding Point Addition for When $P_1 = P_2$

We adjust the `__add__` method to account for this particular case. We have the formulas, and now we implement them:

$$s = (3x_1^2 + a)/(2y_1)$$
$$x_3 = s^2 - 2x_1$$
$$y_3 = s(x_1 - x_3) - y_1$$

Exercise 7

Write the `__add__` method when $P_1 = P_2$.

Coding One More Exception

There is one more exception, and this involves the case where the tangent line is vertical (Figure 2-19).

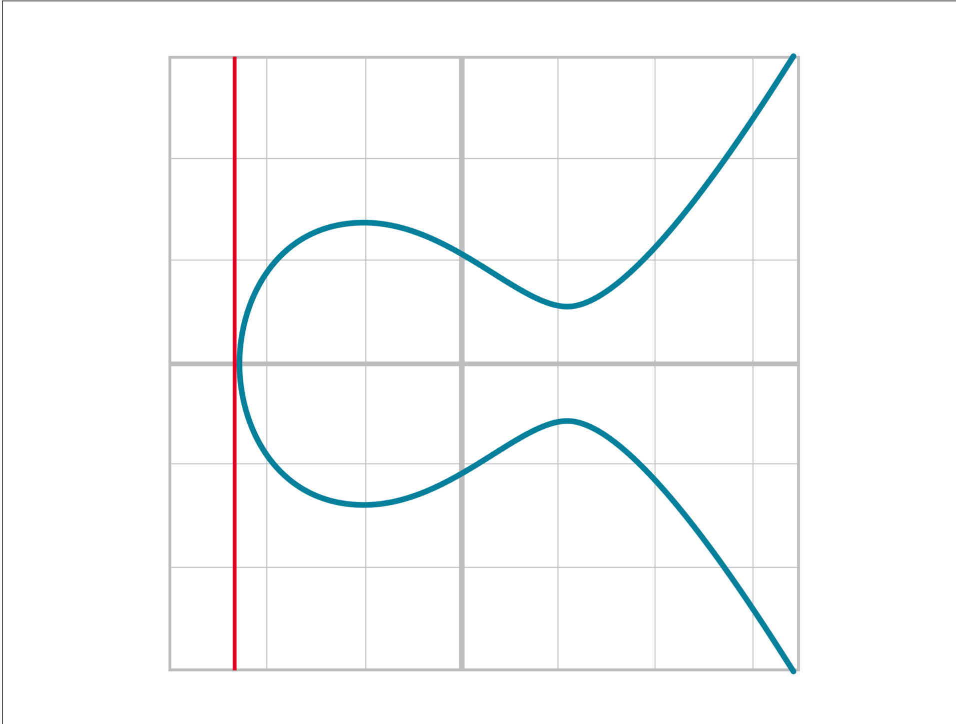


Figure 2-19. Vertical and tangent to the curve

This can only happen if $P_1 = P_2$ and the y coordinate is 0, in which case the slope calculation will end up with a 0 in the denominator.

We handle this with a special case:

```
class Point:
    ...
    def __add__(self, other):
        ...
        if self == other and self.y == 0 * self.x:  ❶
            return self.__class__(None, None, self.a, self.b)
```

❶ If the two points are equal and the *y* coordinate is 0, we return the point at infinity.

Conclusion

We've covered what elliptic curves are, how they work, and how to do point addition. We'll now combine the concepts from the first two chapters to learn elliptic curve cryptography in Chapter 3.

CHAPTER 3
Elliptic Curve Cryptography

The previous two chapters covered some fundamental math. We learned how finite fields work and what an elliptic curve is. In this chapter, we're going to combine the two concepts to learn elliptic curve cryptography. Specifically, we're going to build the primitives needed to sign and verify messages, which is at the heart of what Bitcoin does.

Elliptic Curves over Reals

We discussed in Chapter 2 what an elliptic curve looks like visually because we were plotting the curve over *real* numbers. Specifically, it's not just integers or even rational numbers, but all real numbers. Pi, *sqrt*(2), *e*+7th root of 19, and the like are all real numbers.

This worked because real numbers are also a field. Unlike a *finite* field, there are an *infinite* number of real numbers, but otherwise the same properties hold:

1. If a and b are in the set, $a + b$ and $a \cdot b$ are in the set.
2. 0 exists and has the property $a + 0 = a$.
3. 1 exists and has the property $a \cdot 1 = a$.
4. If a is in the set, $-a$ is in the set, which is defined as the value that makes $a + (-a) = 0$.
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 $a \cdot a^{-1} = 1$.

Clearly, all of these are true: normal addition and multiplication apply for the first part, the additive and multiplicative identities 0 and 1 exist, $-x$ is the additive inverse, and $1/x$ is the multiplicative inverse.

Real numbers are easy to plot on a graph. For example, $y^2 = x^3 + 7$ can be plotted like Figure 3-1.

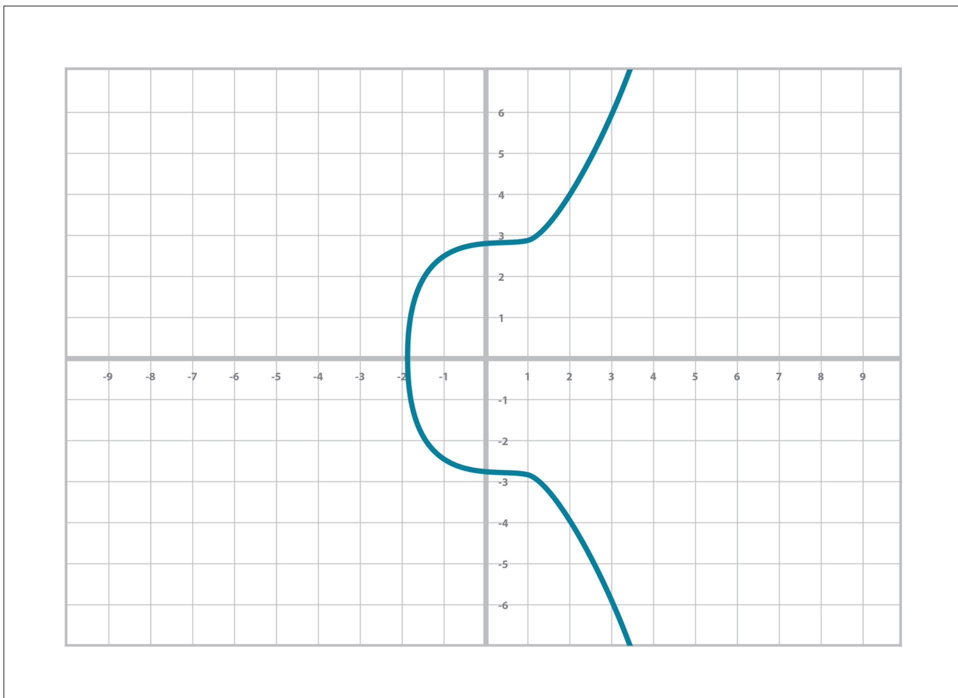


Figure 3-1. secp256k1 over real numbers

It turns out we can use the point addition equations over any field, including the finite fields we learned about in Chapter 1. The only difference is that we have to use the addition/subtraction/multiplication/division as defined in Chapter 1, not the "normal" versions that the real numbers use.

Elliptic Curves over Finite Fields

So what does an elliptic curve over a finite field look like? Let's look at the equation $y^2 = x^3 + 7$ over F_{103}. We can verify that the point (17,64) is on the curve by calculating both sides of the equation:

$$y^2 = 64^2 \% 103 = 79$$
$$x^3 + 7 = (17^3+7) \% 103 = 79$$

We've verified that the point is on the curve using finite field math.

Because we're evaluating the equation over a finite field, the plot of the equation looks vastly different (Figure 3-2).

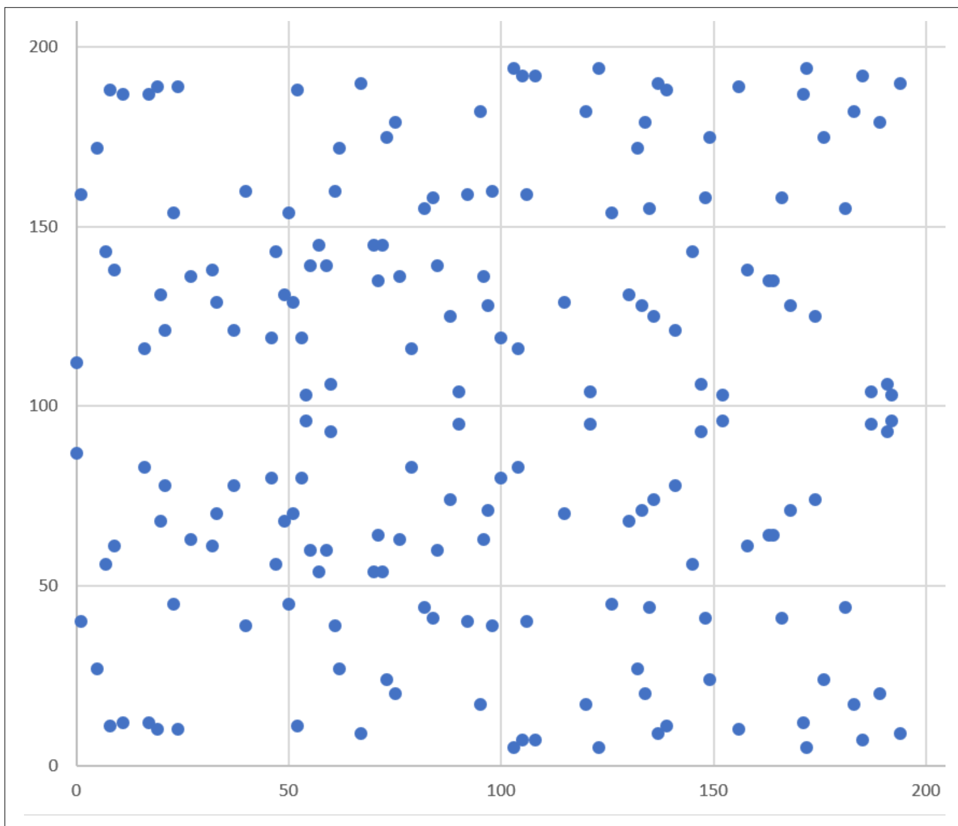


Figure 3-2. Elliptic curve over a finite field

As you can see, it's very much a scattershot of points and there's no smooth curve here. This is not surprising since the points are discrete. About the only pattern is that the curve is symmetric right around the middle, because of the y^2 term. The graph is not symmetric over the x-axis as in the curve over reals, but about halfway up the y-axis due to there not being negative numbers in a finite field.

What's amazing is that we can use the same point addition equations with the addition, subtraction, multiplication, division, and exponentiation as we defined them for finite fields, and everything still works. This may seem surprising, but abstract math has regularities like this despite being different from the traditional modes of calculation you may be familiar with.

Exercise 1

Evaluate whether these points are on the curve $y^2 = x^3 + 7$ over F_{223}:

(192,105), (17,56), (200,119), (1,193), (42,99)

Coding Elliptic Curves over Finite Fields

Because we defined an elliptic curve point and defined the `+`, `-` ,`*` and `/` operators for finite fields, we can combine the two classes to create elliptic curve points over a finite field:

```
>>> from ecc import FieldElement, Point
>>> a = FieldElement(num=0, prime=223)
>>> b = FieldElement(num=7, prime=223)
>>> x = FieldElement(num=192, prime=223)
>>> y = FieldElement(num=105, prime=223)
>>> p1 = Point(x, y, a, b)
>>> print(p1)
Point(192,105)_0_7 FieldElement(223)
```

When initializing `Point`, we will run through this part of the code:

```
class Point:

    def __init__(self, x, y, a, b):
        self.a = a
        self.b = b
        self.x = x
        self.y = y
        if self.x is None and self.y is None:
            return
        if self.y**2 != self.x**3 + a * x + b:
            raise ValueError('({}, {}) is not on the curve'.format(x, y))
```

The addition (`+`), multiplication (`*`), exponentiation (`**`), and not equals (`!=`) operators here use the `__add__`, `__mul__`, `__pow__`, and `__ne__` methods from `FiniteField`, respectively, and *not* the integer equivalents. Being able to do the same equation but with different definitions for the basic arithmetic operators is how we construct an elliptic curve cryptography library.

We've already coded the two classes that we need to implement elliptic curve points over a finite field. However, to check our work, it will be useful to create a test suite. We will do this using the results of Exercise 2:

```
class ECCTest(TestCase):

    def test_on_curve(self):
        prime = 223
        a = FieldElement(0, prime)
```

```
        b = FieldElement(7, prime)
        valid_points = ((192, 105), (17, 56), (1, 193))
        invalid_points = ((200, 119), (42, 99))
        for x_raw, y_raw in valid_points:
            x = FieldElement(x_raw, prime)
            y = FieldElement(y_raw, prime)
            Point(x, y, a, b)  ❶
        for x_raw, y_raw in invalid_points:
            x = FieldElement(x_raw, prime)
            y = FieldElement(y_raw, prime)
            with self.assertRaises(ValueError):
                Point(x, y, a, b)  ❶
```

❶ We pass in `FieldElement` objects to the `Point` class for initialization. This will, in turn, use all the overloaded math operations in `FieldElement`.

We can now run this test like so:

```
>>> import ecc
>>> from helper import run  ❶
>>> run(ecc.ECCTest('test_on_curve'))
.
----------------------------------------------------------------------
Ran 1 test in 0.001s

OK
```

❶ `helper` is a module with some very useful utility functions, including the ability to run unit tests individually.

Point Addition over Finite Fields

We can use all the same equations over finite fields, including the linear equation:

$$y = mx + b$$

It turns out that a “line” in a finite field is not quite what you’d expect (Figure 3-3).

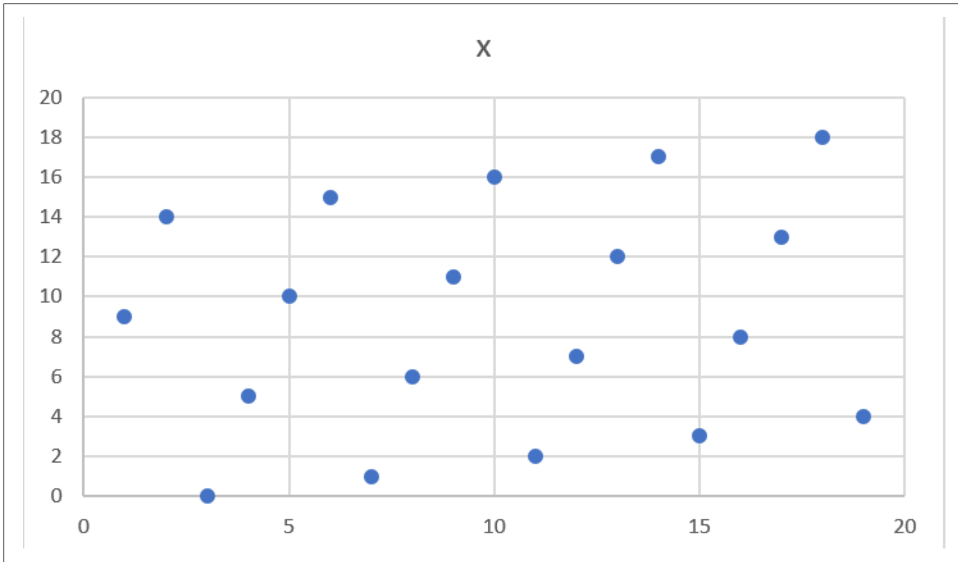


Figure 3-3. Line over a finite field

The equation nevertheless works, and we can calculate what y should be for a given x.

Remarkably, point addition works over finite fields as well. This is because elliptic curve point addition works over all fields! The same exact formulas we used to calculate point addition over reals work over finite fields. Specifically, when $x_1 \neq x_2$:

$$P_1 = (x_1,y_1), P_2 = (x_3,y_3), P_3 = (x_3,y_3)$$
$$P_1 + P_2 = P_3$$
$$s = (y_3 - y_1)/(x_3 - x_1)$$
$$x_3 = s^2 - x_1 - x_3$$
$$y_3 = s(x_1 - x_3) - y_1$$

And when $P_1 = P_2$:

$$P_1 = (x_1,y_1), P_3 = (x_3,y_3)$$
$$P_1 + P_1 = P_3$$
$$s = (3x_1^2 + a)/(2y_1)$$
$$x_3 = s^2 - 2x_1$$
$$y_3 = s(x_1 - x_3) - y_1$$

All of the equations for elliptic curves work over finite fields, which sets us up to create some cryptographic primitives.

Coding Point Addition over Finite Fields

Because we coded `FieldElement` in such a way as to define `__add__`, `__sub__`, `__mul__`, `__truediv__`, `__pow__`, `__eq__`, and `__ne__`, we can simply initialize `Point` with `FieldElement` objects and point addition will work:

```
>>> from ecc import FieldElement, Point
>>> prime = 223
>>> a = FieldElement(num=0, prime=prime)
>>> b = FieldElement(num=7, prime=prime)
>>> x1 = FieldElement(num=192, prime=prime)
>>> y1 = FieldElement(num=105, prime=prime)
>>> x2 = FieldElement(num=17, prime=prime)
>>> y2 = FieldElement(num=56, prime=prime)
>>> p1 = Point(x1, y1, a, b)
>>> p2 = Point(x2, y2, a, b)
>>> print(p1+p2)
Point(170,142)_0_7 FieldElement(223)
```

Exercise 2

For the curve $y^2 = x^3 + 7$ over F_{223}, find:

- (170,142) + (60,139)
- (47,71) + (17,56)
- (143,98) + (76,66)

Exercise 3

Extend `ECCTest` to test for the additions from the previous exercise. Call this `test_add`.

Scalar Multiplication for Elliptic Curves

Because we can add a point to itself, we can introduce some new notation:

$$(170,142) + (170,142) = 2 \cdot (170,142)$$

Similarly, because we have associativity, we can actually add the point again:

$$2 \cdot (170,142) + (170,142) = 3 \cdot (170, 142)$$

We can do this as many times as we want. This is what we call *scalar multiplication*. That is, we have a *scalar* number in front of the point. We can do this because we have defined point addition and point addition is associative.

One property of scalar multiplication is that it's really hard to predict without calculating (see Figure 3-4).

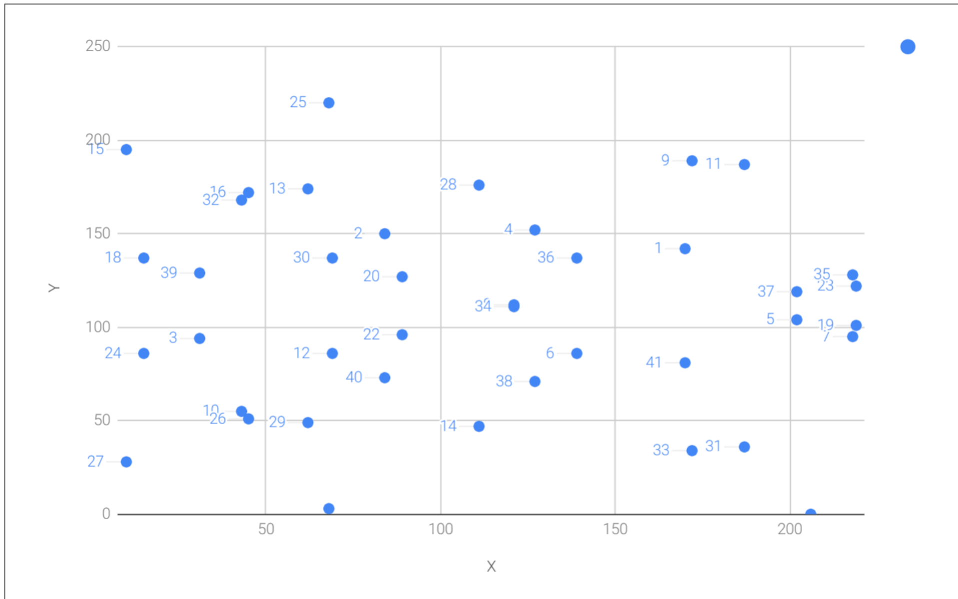


Figure 3-4. Scalar multiplication results for $y^2 = x^3 + 7$ over F_{223} for point (170,142)

Each point is labeled by how many times we've added the point. You can see that this is a complete scattershot. This is because point addition is nonlinear and not easy to calculate. Performing scalar multiplication is straightforward, but doing the opposite, point division, is not.

This is called the *discrete log problem* and is the basis of elliptic curve cryptography.

Another property of scalar multiplication is that at a certain multiple, we get to the point at infinity (remember, the point at infinity is the additive identity or 0). If we imagine a point G and scalar-multiply until we get the point at infinity, we end up with a set:

$$\{ G, 2G, 3G, 4G, \ldots nG \} \text{ where } nG = 0$$

It turns out that this set is called a *group*, and because n is finite, we have a *finite group* (or more specifically, a *finite cyclic group*). Groups are interesting mathematically because they behave well with respect to addition:

$$G + 4G = 5G \text{ or } aG + bG = (a + b)G$$

When we combine the fact that scalar multiplication is easy to do in one direction but hard in the other and the mathematical properties of a group, we have exactly what we need for elliptic curve cryptography.

Why Is This Called the Discrete Log Problem?

You may be wondering why the problem of reversing scalar *multiplication* is referred to as the discrete *log* problem.

We called the operation between the points "addition," but we could easily have called it a point "operation." Typically, a new operation that you define in math is denoted with the dot operator (•). The dot operator is also used for multiplication, and it sometimes helps to think that way:

$$P_1 \cdot P_2 = P_3$$

When you do lots of multiplying, that's the same as exponentiation. Scalar multiplication when we called it "point addition" becomes scalar exponentiation when thinking "point multiplication":

$$P^7 = Q$$

The discrete log problem in this context is the ability to reverse this equation, which ends up being:

$$\log_P Q = 7$$

The log equation on the left has no analytically calculable algorithm. That is, there is no known formula that you can plug in to get the answer generally. This is all a bit confusing, but it's fair to say that we could call the problem the "discrete point division" problem instead of the discrete log problem.

Exercise 4

For the curve $y^2 = x^3 + 7$ over F_{223}, find:

- 2 • (192,105)
- 2 • (143,98)
- 2 • (47,71)
- 4 • (47,71)
- 8 • (47,71)
- 21 • (47,71)

Scalar Multiplication Redux

Scalar multiplication is adding the same point to itself some number of times. The key to making scalar multiplication into public key cryptography is using the fact that scalar multiplication on elliptic curves is very hard to reverse. Note the previous exercise. Most likely, you calculated the point $s \cdot (47,71)$ in F_{223} for s from 1 until 21. Here are the results:

```
>>> from ecc import FieldElement, Point
>>> prime = 223
>>> a = FieldElement(0, prime)
>>> b = FieldElement(7, prime)
>>> x = FieldElement(47, prime)
>>> y = FieldElement(71, prime)
>>> p = Point(x, y, a, b)
>>> for s in range(1,21):
...     result = s*p
...     print('{}*(47,71)=({},{})'.format(s,result.x.num,result.y.num))
1*(47,71)=(47,71)
2*(47,71)=(36,111)
3*(47,71)=(15,137)
4*(47,71)=(194,51)
5*(47,71)=(126,96)
6*(47,71)=(139,137)
7*(47,71)=(92,47)
8*(47,71)=(116,55)
9*(47,71)=(69,86)
10*(47,71)=(154,150)
11*(47,71)=(154,73)
12*(47,71)=(69,137)
13*(47,71)=(116,168)
14*(47,71)=(92,176)
15*(47,71)=(139,86)
16*(47,71)=(126,127)
17*(47,71)=(194,172)
18*(47,71)=(15,86)
19*(47,71)=(36,112)
20*(47,71)=(47,152)
```

If you look closely at the numbers, there's no real discernible pattern to the scalar multiplication. The x coordinates don't always increase or decrease, and neither do the y coordinates. About the only pattern is that between 10 and 11, the x coordinates are equal (10 and 11 have the same x, as do 9 and 12, 8 and 13, and so on). This is due to the fact that $21 \cdot (47,71) = 0$.

Scalar multiplication looks really random, and that's what gives this equation *asymmetry*. An *asymmetric* problem is one that's easy to calculate in one direction, but hard to reverse. For example, it's easy enough to calculate 12 • (47,71). But if we were presented with this:

$$s \cdot (47,71) = (194,172)$$

would we be able to solve for *s*? We can look up the results shown earlier, but that's because we have a small group. We'll see in "Defining the Curve for Bitcoin" on page 58 that when we have numbers that are a lot larger, discrete log becomes an intractable problem.

Mathematical Groups

The preceding math (finite fields, elliptic curves, combining the two) was really to bring us to this point. What we actually want to generate for the purposes of public key cryptography are finite cyclic groups, and it turns out that if we take a generator point from an elliptic curve over a finite field, we can generate a finite cyclic group.

Unlike fields, groups have only a single operation. In our case, point addition is the operation. Groups also have a few other properties, like closure, invertibility, commutativity, and associativity. Lastly, we need the identity.

Let's look at each property, starting with that last one.

Identity

If you haven't guessed by now, the identity is defined as the point at infinity, which is guaranteed to be in the group since we generate the group when we get to the point at infinity. So:

$$0 + A = A$$

We call 0 the point at infinity because visually, it's the point that exists to help the math work out (Figure 3-5).

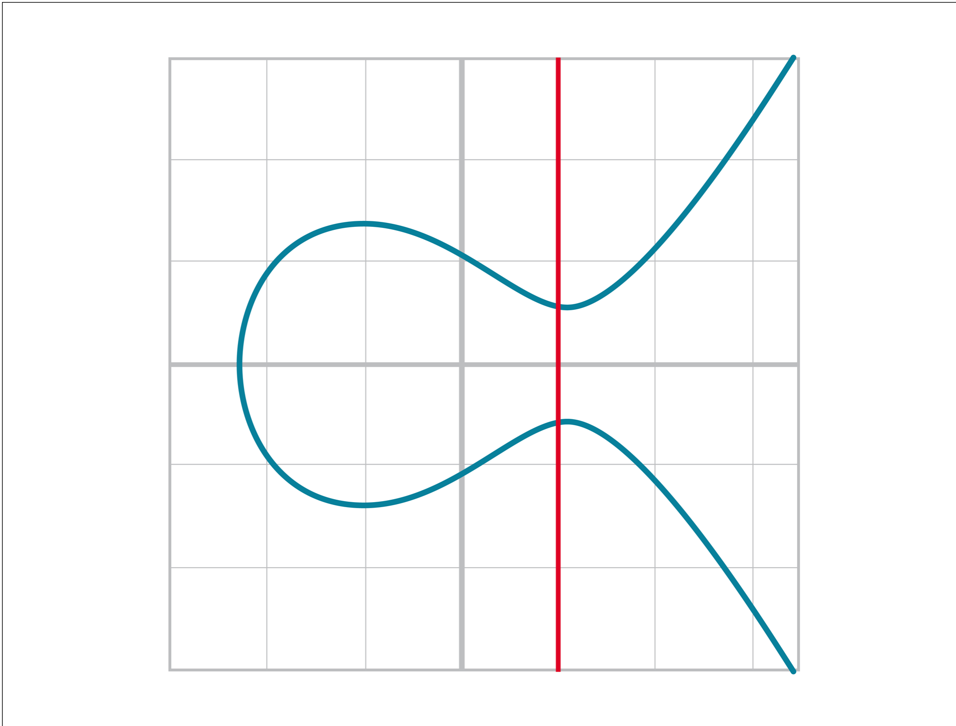


Figure 3-5. Vertical line "intersects" a third time at the point at infinity

Closure

This is perhaps the easiest property to prove since we generated the group in the first place by adding G over and over. Thus, if we have two different elements that look like this:

$$aG + bG$$

We know that the result is going to be:

$$(a + b)G$$

How do we know if this element is in the group? If $a+b < n$ (where n is the order of the group), then we know it's in the group by definition. If $a+b >= n$, then we know $a < n$ and $b < n$, so $a+b < 2n$, so $a+b-n < n$:

$$(a + b - n)G = aG + bG - nG = aG + bG - 0 = aG + bG$$

More generally, $(a + b)G = ((a + b) \% n)G$, where n is the order of the group.

So we know that this element is in the group, proving closure.

Invertibility

Invertibility is easy to depict (Figure 3-6).

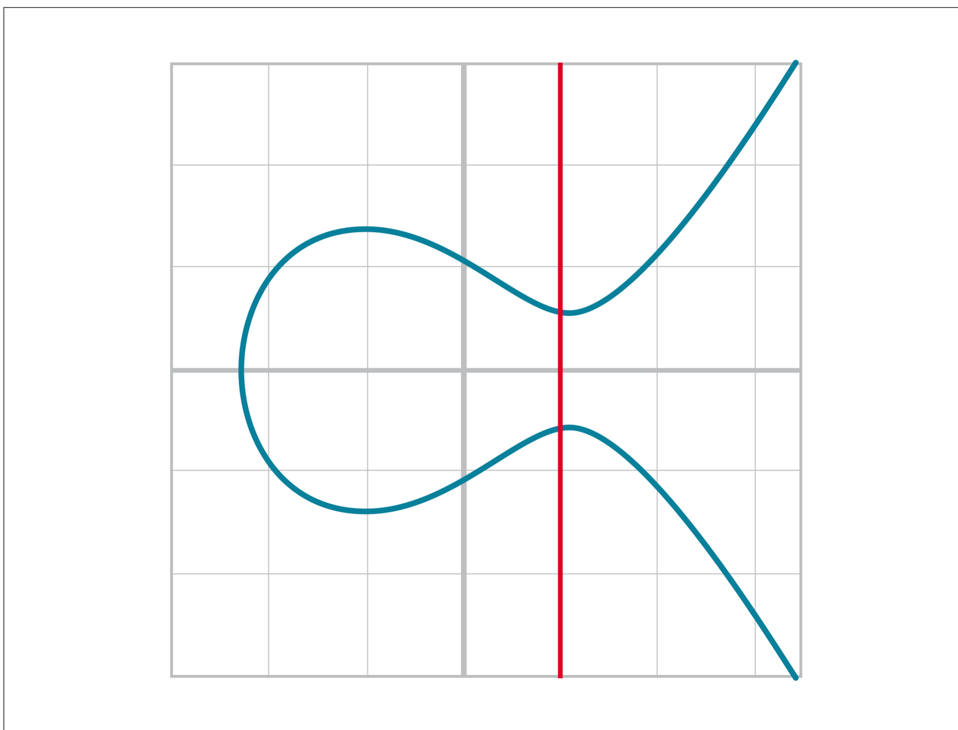


Figure 3-6. Each point is invertible by taking the reflection over the x-axis

Mathematically, we know that if aG is in the group, $(n - a)G$ is also in the group. You can add them together to get $aG + (n - a)G = (a + n - a)G = nG = 0$.

Commutativity

We know from point addition that $A + B = B + A$ (Figure 3-7).

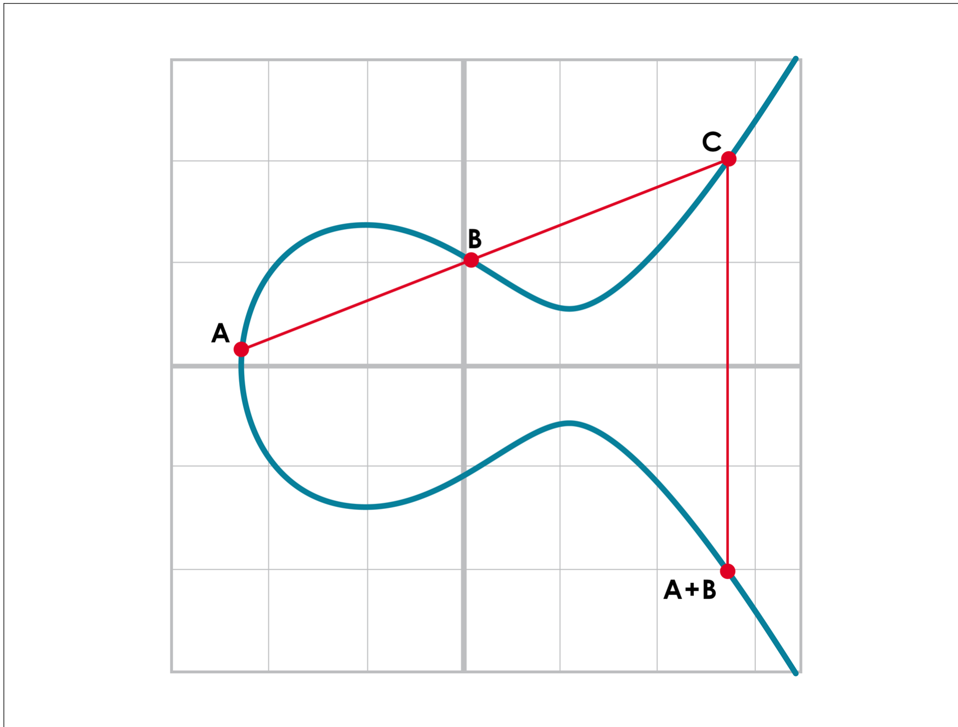


Figure 3-7. The line through the points doesn't change

This means that $aG + bG = bG + aG$, which proves commutativity.

Associativity

We know from point addition that $A + (B + C) = (A + B) + C$ (see Figures 3-8 and 3-9).

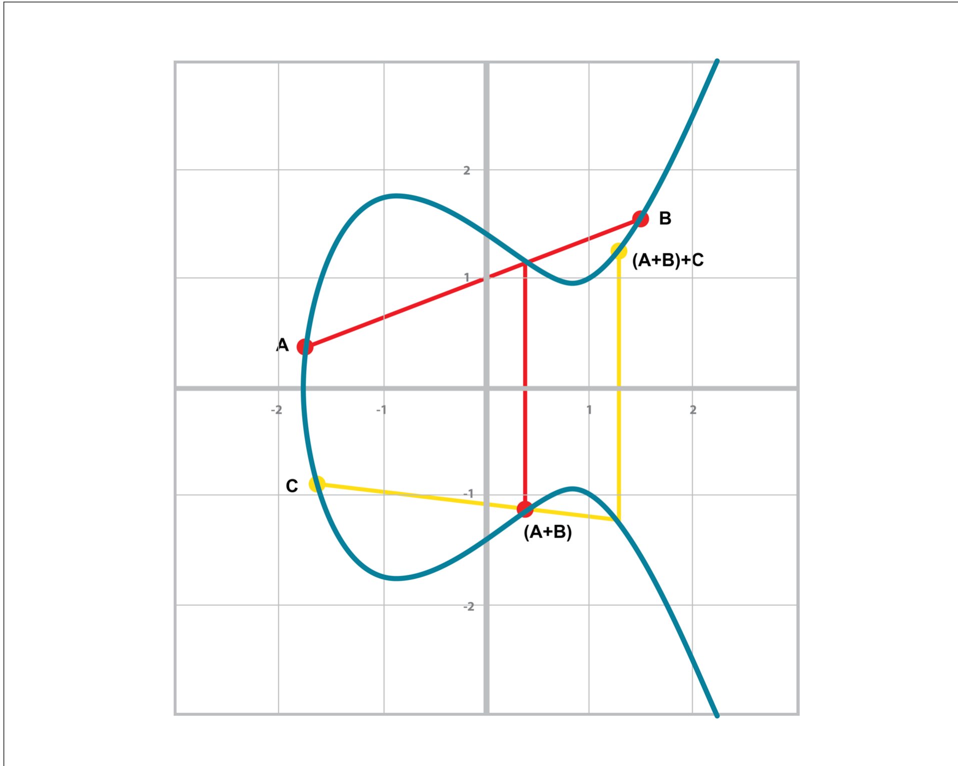


Figure 3-8. (A + B) + C: A + B is computed first before C is added

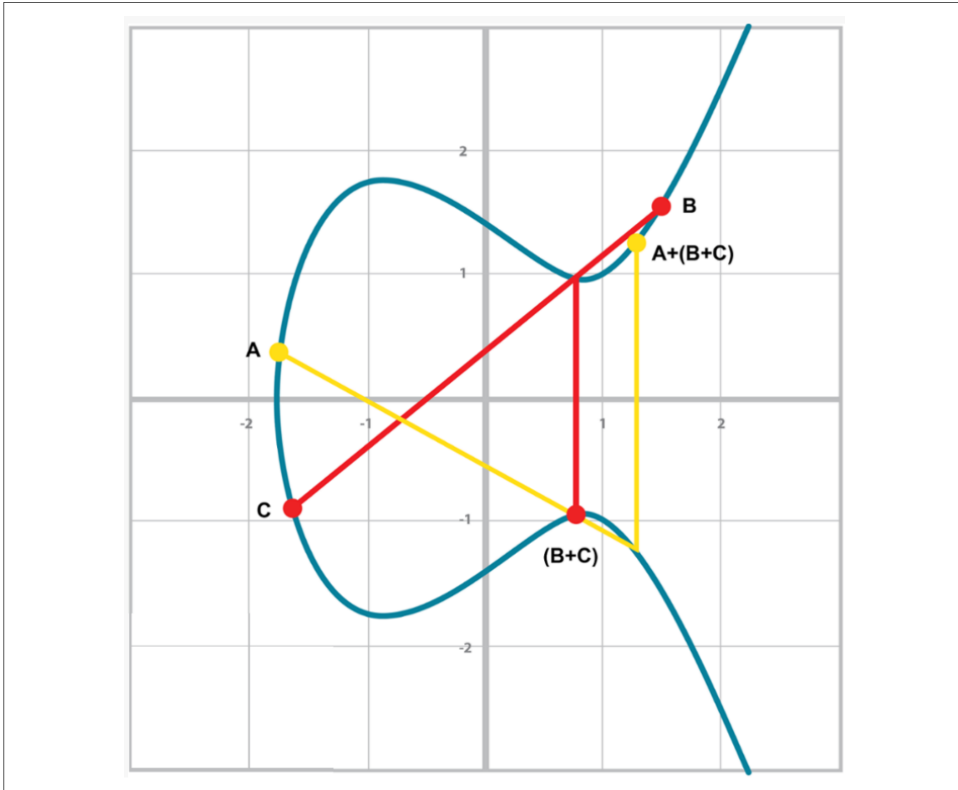


Figure 3-9. A + (B + C): B + C is added first before adding A (note that this results in the same point as in Figure 3-8)

Thus, $aG + (bG + cG) = (aG + bG) + cG$, proving associativity.

Exercise 5

For the curve $y^2 = x^3 + 7$ over F_{223}, find the order of the group generated by (15,86).

Coding Scalar Multiplication

What we're trying to do with Exercise 5 is this:

```
>>> from ecc import FieldElement, Point
>>> prime = 223
>>> a = FieldElement(0, prime)
>>> b = FieldElement(7, prime)
>>> x = FieldElement(15, prime)
>>> y = FieldElement(86, prime)
>>> p = Point(x, y, a, b)
>>> print(7*p)
Point(infinity)
```

We want to be able to scalar-multiply the point with some number. Thankfully, there's a method in Python called `__rmul__` that can be used to override the front multiplication. A naive implementation looks something like this:

```
class Point:
    ...
    def __rmul__(self, coefficient):
        product = self.__class__(None, None, self.a, self.b) ❶
        for _ in range(coefficient): ❷
            product += self
        return product
```

❶ We start the `product` at 0, which in the case of point addition is the point at infinity.

❷ We loop `coefficient` times and add the point each time.

This is fine for small coefficients, but what if we have a very large coefficient—that is, a number that's so large that we won't be able to get out of this loop in a reasonable amount of time? For example, a coefficient of 1 trillion is going to take a really long time.

There's a cool technique called *binary expansion* that allows us to perform multiplication in $\log_2(n)$ loops, which dramatically reduces the calculation time for large numbers. For example, 1 trillion is 40 bits in binary, so we only have to loop 40 times for a number that's generally considered very large:

```
class Point:
    ...
    def __rmul__(self, coefficient):
        coef = coefficient
        current = self ❶
        result = self.__class__(None, None, self.a, self.b) ❷
        while coef:
            if coef & 1: ❸
```

```
            result += current
        current += current  ❹
        coef >>= 1  ❺
    return result
```

❶ `current` represents the point that's at the current bit. The first time through the loop it represents 1 × `self`; the second time it will be 2 × `self`, the third time 4 × `self`, then 8 × `self`, and so on. We double the point each time. In binary the coefficients are 1, 10, 100, 1000, 10000, etc.

❷ We start the result at 0, or the point at infinity.

❸ We are looking at whether the rightmost bit is a 1. If it is, then we add the value of the current bit.

❹ We need to double the point until we're past how big the coefficient can be.

❺ We bit-shift the coefficient to the right.

This is an advanced technique. If you don't understand bitwise operators, think of representing the coefficient in binary and only adding the point where there are 1's.

With `__add__` and `__rmul__`, we can start defining some more complicated elliptic curves.

Defining the Curve for Bitcoin

While we've been using relatively small primes for the sake of examples, we are not restricted to such small numbers. Small primes mean that we can use a computer to search through the entire group. If the group has a size of 301, the computer can easily do 301 computations to reverse scalar multiplication or break discrete log.

But what if we made the prime larger? It turns out that we can choose much larger primes than we've been using. The security of elliptic curve cryptography depends on computers *not* being able to go through an appreciable fraction of the group.

An elliptic curve for public key cryptography is defined with the following parameters:

- We specify the a and b of the curve $y^2 = x^3 + ax + b$.
- We specify the prime of the finite field, p.
- We specify the x and y coordinates of the generator point G.
- We specify the order of the group generated by G, n.

These numbers are known publicly and together form the cryptographic curve. There are many cryptographic curves and they have different security/convenience trade-offs, but the one we're most interested in is the one Bitcoin uses: secp256k1. The parameters for secp256k1 are these:

- $a = 0$, $b = 7$, making the equation $y^2 = x^3 + 7$
- $p = 2^{256} - 2^{32} - 977$
- G_x = 0x79be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798
- G_y = 0x483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8
- n = 0xfffffffffffffffffffffffffffffffebaaedce6af48a03bbfd25e8cd0364141

G_x refers to the x coordinate of the point G and G_y the y coordinate. The numbers starting with 0x are hexadecimal numbers.

There are a few things to notice about this curve. First, the equation is relatively simple. Many curves have a and b values that are much bigger.

Second, p is extremely close to 2^{256}. This means that most numbers under 2^{256} are in the prime field, and thus any point on the curve has x and y coordinates that are expressible in 256 bits each. n is also very close to 2^{256}. This means any scalar multiple can also be expressed in 256 bits.

Third, 2^{256} is a huge number (see sidebar). Amazingly, any number below 2^{256} can be stored in 32 bytes. This means that we can store the private key relatively easily.

How Big is 2^{256}?

2^{256} doesn't seem that big because we can express it succinctly, but in reality, it is an enormous number. To give you an idea, here are some relative scales:

$2^{256} \sim 10^{77}$

- Number of atoms in and on Earth ~ 10^{50}
- Number of atoms in the solar system ~ 10^{57}
- Number of atoms in the Milky Way ~ 10^{68}
- Number of atoms in the universe ~ 10^{80}

A trillion (10^{12}) computers doing a trillion computations every trillionth (10^{-12}) of a second for a trillion years is still less than 10^{56} computations.

Think of finding a private key this way: there are as many possible private keys in Bitcoin as there are atoms in a billion galaxies.

Working with secp256k1

Since we know all of the parameters for secp256k1, we can verify in Python whether the generator point, G, is on the curve $y^2 = x^3 + 7$:

```
>>> gx = 0x79be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798
>>> gy = 0x483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8
>>> p = 2**256 - 2**32 - 977
>>> print(gy**2 % p == (gx**3 + 7) % p)
True
```

Furthermore, we can verify in Python whether the generator point, G, has the order n:

```
>>> from ecc import FieldElement, Point
>>> gx = 0x79be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798
>>> gy = 0x483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8
>>> p = 2**256 - 2**32 - 977
>>> n = 0xfffffffffffffffffffffffffffffffebaaedce6af48a03bbfd25e8cd0364141
>>> x = FieldElement(gx, p)
>>> y = FieldElement(gy, p)
>>> seven = FieldElement(7, p)
>>> zero = FieldElement(0, p)
>>> G = Point(x, y, zero, seven)
>>> print(n*G)
Point(infinity)
```

Since we know the curve we will work in, this is a good time to create a subclass in Python to work exclusively with the parameters for secp256k1. We'll define the equivalent `FieldElement` and `Point` objects, but specific to the secp256k1 curve. Let's start by defining the field we'll be working in:

```
P = 2**256 - 2**32 - 977
...
class S256Field(FieldElement):

    def __init__(self, num, prime=None):
        super().__init__(num=num, prime=P)

    def __repr__(self):
        return '{:x}'.format(self.num).zfill(64)
```

We're subclassing the `FieldElement` class so we don't have to pass in P all the time. We also want to display a 256-bit number consistently by filling 64 characters so we can see any leading zeros.

Similarly, we can define a point on the secp256k1 curve and call it `S256Point`:

```
A = 0
B = 7
...
class S256Point(Point):
```

```
    def __init__(self, x, y, a=None, b=None):
        a, b = S256Field(A), S256Field(B)
        if type(x) == int:
            super().__init__(x=S256Field(x), y=S256Field(y), a=a, b=b)
        else:
            super().__init__(x=x, y=y, a=a, b=b)  ❶
```

❶ In case we initialize with the point at infinity, we need to let x and y through directly instead of using the `S256Field` class.

We now have an easier way to initialize a point on the secp256k1 curve, without having to define `a` and `b` every time like we have to with the `Point` class.

We can also define `__rmul__` a bit more efficiently, since we know the order of the group, n. Since we're coding Python, we'll name this with a capital `N` to make it clear that `N` is a constant:

```
N = 0xfffffffffffffffffffffffffffffffebaaedce6af48a03bbfd25e8cd0364141
...
class S256Point(Point):
    ...
    def __rmul__(self, coefficient):
        coef = coefficient % N  ❶
        return super().__rmul__(coef)
```

❶ We can mod by n because $nG = 0$. That is, every n times we cycle back to zero or the point at infinity.

We can now define G directly and keep it around since we'll be using it a lot going forward:

```
G = S256Point(
    0x79be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798,
    0x483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8)
```

Now checking that the order of G is n is trivial:

```
>>> from ecc import G, N
>>> print(N*G)
S256Point(infinity)
```

Public Key Cryptography

At last, we have the tools that we need to do public key cryptography operations. The key operation that we need is $P = eG$, which is an asymmetric equation. We can easily compute P when we know e and G, but we cannot easily compute e when we know P and G. This is the discrete log problem described earlier.

The difficulty of discrete log will be essential to understanding signing and verification algorithms.

Generally, we call *e* the *private key* and *P* the *public key*. Note here that the private key is a single 256-bit number and the public key is a coordinate (x,y), where x and y are *each* 256-bit numbers.

Signing and Verification

To set up the motivation for why signing and verification exists, imagine this scenario. You want to prove that you are a really good archer, like at the level where you can hit any target you want within 500 yards as opposed to being able to hit any particular target.

Now, if someone could observe you and interact with you, proving this would be easy. Perhaps they would position your son 400 yards away with an apple on his head and challenge you to hit that apple with an arrow. You, being a very good archer, could do this and prove your expertise. The target, if specified by the challenger, makes your archery skill easy to verify.

Unfortunately, this doesn't scale very well. If, for example you wanted to prove this to 10 people, you would have to shoot 10 different arrows at 10 different targets from 10 different challenges. You could try to do something like have 10 people watch you shoot a single arrow, but since they can't all choose the target, they can never be sure that you're not just good at hitting one particular target instead of an arbitrary target. What we want is something that you can do once, that requires no interaction back and forth with the verifiers, but that still proves that you are indeed, a good archer that can hit *any* target.

If, for example, you simply shot an arrow into a target of your choosing, the people observing afterward wouldn't necessarily be convinced. After all, you might have painted the target around wherever your arrow happened to land. So what can you do?

Here's a very clever thing you can do. Inscribe the tip of the arrow with the position of the target that you're hitting ("apple on top of my son's head") and then hit that target with your arrow. Now anyone seeing the target can take an X-ray machine and look at the tip of the embedded arrow and see that the tip indeed says exactly where it was going to hit. The tip clearly had to be inscribed before the arrow was shot, so this can prove you are actually a good archer (provided the actual target isn't just one that you've practiced hitting over and over).

This is the same technique we're using with signing and verification, except what we're proving isn't that we're good archers, but that we know a secret number. We want to prove possession of the secret without revealing the secret itself. We do this by putting the target into our calculation and hitting that target.

Ultimately this is going to be used in transactions, which will prove that the rightful owners of the secrets are spending the bitcoins.

Inscribing the Target

The inscribing of the target depends on the *signature algorithm*, and in our case that algorithm is called the Elliptic Curve Digital Signature Algorithm, or ECDSA for short.

The secret in our case is e satisfying the following:

$$eG = P$$

where P is the public key and e is the private key.

The target that we're going to aim at is a random 256-bit number, k. We then do this:

$$kG = R$$

R is now the target that we're aiming for. In fact, we're only going to care about the x coordinate of R, which we'll call r. You may have guessed already that r here stands for *random*.

We claim at this point that the following equation is equivalent to the discrete log problem:

$$uG + vP = kG$$

where k was chosen randomly, $u,v \neq 0$ can be chosen by the signer, and G and P are known. This is due to the fact that:

$$uG + vP = kG \text{ implies } vP = (k - u)G$$

Since $v \neq 0$, we can divide by the scalar multiple v:

$$P = ((k - u)/v)G$$

If we know e, we have:

$$eG = ((k - u)/v)G \text{ or } e = (k - u)/v$$

This means that any (u,v) combination that satisfies the preceding equation will suffice.

If we don't know e, we'll have to play with (u,v) until $e = (k-u)/v$. If we could solve this with any (u,v) combination, that would mean we'd have solved $P = eG$ while knowing only P and G. In other words, we'd have broken the discrete log problem.

This means to provide a correct u and v, we either have to break the discrete log problem or know the secret e. Since we assume discrete log is hard, we can say e is assumed to be known by the one who came up with u and v.

One subtle thing that we haven't talked about is that we have to incorporate the purpose of our shooting. This is a contract that gets fulfilled as a result of shooting at the target. William Tell, for example, was shooting so that he could save his son (shoot the target and you get to save your son). You can imagine there would be other reasons to hit the target and other "rewards" that the person hitting the target would receive. This has to be incorporated into our equations.

In signature/verification parlance, this is called the *signature hash*. A hash is a deterministic function that takes arbitrary data into data of fixed size. This is a fingerprint of the message containing the intent of the shooter, which anyone verifying the message already knows. We denote this with the letter z. This is incorporated into our $uG + vP$ calculation this way:

$$u = z/s, v = r/s$$

Since r is used in the calculation of v, we now have the tip of the arrow inscribed. We also have the intent of the shooter incorporated into u, so both the reason for shooting and the target that is being aimed at are now part of the equation.

To make the equation work, we can calculate s:

$$uG + vP = R = kG$$
$$uG + veG = kG$$
$$u + ve = k$$
$$z/s + re/s = k$$
$$(z + re)/s = k$$
$$s = (z + re)/k$$

This is the basis of the signature algorithm, and the two numbers in a signature are r and s.

Verification is straightforward:

$$uG + vP \text{ where } u,v \neq 0$$
$$uG + vP = (z/s)G + (re/s)G = ((z + re)/s)G = ((z + re)/((z + re)/k))G = kG = (r,y)$$

Why We Don't Reveal k

At this point, you might be wondering why we don't reveal k and instead reveal the x coordinate of R, or r. If we were to reveal k, then:

$$uG + vP = R$$
$$uG + veG = kG$$
$$kG - uG = veG$$
$$(k - u)G = veG$$
$$(k - u) = ve$$
$$(k - u)/v = e$$

means that our secret would be revealed, which would defeat the whole purpose of the signature. We can, however, reveal R.

It's worth mentioning again: make sure you're using truly random numbers for k, as even accidentally revealing k for a known signature is the equivalent of revealing your secret and losing your funds!

Verification in Depth

Signatures sign some fixed-length value (our "contract")—in our case, something that's 32 bytes. The fact that 32 bytes is 256 bits is not a coincidence, as the thing we're signing will be a scalar for G.

To guarantee that the thing we're signing is 32 bytes, we hash the document first. In Bitcoin, the hashing function is hash256, or two rounds of sha256. This guarantees the thing that we're signing is exactly 32 bytes. We will call the result of the hash the *signature hash*, or z.

The signature that we are verifying has two components, (r,s). r is the x coordinate of some point R that we'll come back to. The formula for s is as above:

$$s = (z+re)/k$$

Keep in mind that we know e ($P = eG$, or what we're proving we know in the first place), we know k ($kG = R$, remember?), and we know z.

We will now construct $R = uG + vP$ by defining u and v this way:

$$u = z/s$$
$$v = r/s$$

Thus:

$$uG + vP = (z/s)G + (r/s)P = (z/s)G + (re/s)G = ((z + re)/s)G$$

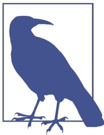
We know $s = (z + re)/k$, so:

$$uG + vP = ((z + re) / ((z + re)/k))G = kG = R$$

We've successfully chosen u and v in such a way as to generate R as we intended. Furthermore, we used r in the calculation of v, proving we knew what R would be. The only way we can know the details of R beforehand is if we know e.

To wit, here are the steps:

1. We are given (r,s) as the signature, z as the hash of the thing being signed, and P as the public key (or public point) of the signer.
2. We calculate $u = z/s$, $v = r/s$.
3. We calculate $uG + vP = R$.
4. If R's x coordinate equals r, the signature is valid.



Why Two Rounds of sha256?

The calculation of z requires two rounds of sha256, or hash256. You may be wondering why there are two rounds when only one is necessary to get a 256-bit number. The reason is for security.

There is a well-known hash collision attack on SHA-1 called a *birthday attack* that makes finding collisions much easier. Google found a SHA-1 collision using some modifications of a birthday attack and a lot of other things in 2017. Using SHA-1 twice, or *double SHA-1*, is the way to defeat or slow down some forms of this attack.

Two rounds of sha256 don't necessarily prevent all possible attacks, but doing two rounds is a defense against some potential weaknesses.

Verifying a Signature

We can now verify a signature using some of the primitives that we have:

```
>>> from ecc import S256Point, G, N
>>> z = 0xbc62d4b80d9e36da29c16c5d4d9f11731f36052c72401a76c23c0fb5a9b74423
>>> r = 0x37206a0610995c58074999cb9767b87af4c4978db68c06e8e6e81d282047a7c6
>>> s = 0x8ca63759c1157ebeaec0d03cecca119fc9a75bf8e6d0fa65c841c8e2738cdaec
>>> px = 0x04519fac3d910ca7e7138f7013706f619fa8f033e6ec6e09370ea38cee6a7574
>>> py = 0x82b51eab8c27c66e26c858a079bcdf4f1ada34cec420cafc7eac1a42216fb6c4
>>> point = S256Point(px, py)
>>> s_inv = pow(s, N-2, N)  ❶
```

```
>>> u = z * s_inv % N  ❷
>>> v = r * s_inv % N  ❸
>>> print((u*G + v*point).x.num == r)  ❹
True
```

❶ Note that we use Fermat's little theorem for $1/s$, since n is prime.

❷ $u = z/s$.

❸ $v = r/s$.

❹ $uG + vP = (r,y)$. We need to check that the x coordinate is r.

Exercise 6

Verify whether these signatures are valid:

```
P = (0x887387e452b8eacc4acfde10d9aaf7f6d9a0f975aabb10d006e4da568744d06c,
     0x61de6d95231cd89026e286df3b6ae4a894a3378e393e93a0f45b666329a0ae34)

# signature 1
z = 0xec208baa0fc1c19f708a9ca96fdeff3ac3f230bb4a7ba4aede4942ad003c0f60
r = 0xac8d1c87e51d0d441be8b3dd5b05c8795b48875dffe00b7ffcfac23010d3a395
s = 0x68342ceff8935ededd102dd876ffd6ba72d6a427a3edb13d26eb0781cb423c4

# signature 2
z = 0x7c076ff316692a3d7eb3c3bb0f8b1488cf72e1afcd929e29307032997a838a3d
r = 0xeff69ef2b1bd93a66ed5219add4fb51e11a840f404876325a1e8ffe0529a2c
s = 0xc7207fee197d27c618aea621406f6bf5ef6fca38681d82b2f06fddbdce6feab6
```

Programming Signature Verification

We already have a class `S256Point`, which is the public point for the private key. We create a `Signature` class that houses the r and s values:

```
class Signature:

    def __init__(self, r, s):
        self.r = r
        self.s = s

    def __repr__(self):
        return 'Signature({:x},{:x})'.format(self.r, self.s)
```

We will be doing more with this class in Chapter 4.

We can now write the `verify` method on `S256Point` based on this:

```
class S256Point(Point):
    ...
    def verify(self, z, sig):
```

```
        s_inv = pow(sig.s, N - 2, N)  ❶
        u = z * s_inv % N  ❷
        v = sig.r * s_inv % N  ❸
        total = u * G + v * self  ❹
        return total.x.num == sig.r  ❺
```

❶ s_inv ($1/s$) is calculated using Fermat's little theorem on the order of the group, n, which is prime.

❷ $u = z/s$. Note that we can mod by n as that's the order of the group.

❸ $v = r/s$. Note that we can mod by n as that's the order of the group.

❹ $uG + vP$ should be R.

❺ We check that the x coordinate is r.

So, given a public key that is a point on the secp256k1 curve and a signature hash, z, we can verify whether a signature is valid or not.

Signing in Depth

Given that we know how verification should work, signing is straightforward. The only missing step is figuring out what k, and thus $R = kG$, to use. We do this by choosing a random k.

The signing procedure is as follows:

1. We are given z and know e such that $eG = P$.
2. Choose a random k.
3. Calculate $R = kG$ and $r = x$ coordinate of R.
4. Calculate $s = (z + re)/k$.
5. Signature is (r,s).

Note that the public key (pubkey) P has to be transmitted to whoever wants to verify it, and z must be known by the verifier. We'll see later that z is computed and P is sent along with the signature.

Creating a Signature

We can now create a signature.

Be Careful with Random Number Generation

Note that using something like the `random` library from Python to do cryptography is generally not a good idea. This library is for teaching purposes only, so please don't use any of the code explained to you here for production purposes.

We do this using some of the primitives that we have:

```
>>> from ecc import S256Point, G, N
>>> from helper import hash256
>>> e = int.from_bytes(hash256(b'my secret'), 'big')  ❶
>>> z = int.from_bytes(hash256(b'my message'), 'big')  ❷
>>> k = 1234567890  ❸
>>> r = (k*G).x.num  ❹
>>> k_inv = pow(k, N-2, N)
>>> s = (z+r*e) * k_inv % N  ❺
>>> point = e*G  ❻
>>> print(point)
S256Point(028d003eab2e428d11983f3e97c3fa0addf3b42740df0d211795ffb3be2f6c52, \
0ae987b9ec6ea159c78cb2a937ed89096fb218d9e7594f02b547526d8cd309e2)
>>> print(hex(z))
0x231c6f3d980a6b0fb7152f85cee7eb52bf92433d9919b9c5218cb08e79cce78
>>> print(hex(r))
0x2b698a0f0a4041b77e63488ad48c23e8e8838dd1fb7520408b121697b782ef22
>>> print(hex(s))
0xbb14e602ef9e3f872e25fad328466b34e6734b7a0fcd58b1eb635447ffae8cb9
```

❶ This is an example of a "brain wallet," which is a way to keep the private key in your head without having to memorize something too difficult. Please don't use this for a real secret.

❷ This is the signature hash, or hash of the message that we're signing.

❸ We're going to use a fixed k here for demonstration purposes.

❹ $kG = (r,y)$, so we take the x coordinate only.

❺ $s = (z + re)/k$. We can mod by n because we know this is a cyclical group of order n.

❻ The public point needs to be known by the verifier.

Exercise 7

Sign the following message with the secret:

```
e = 12345
z = int.from_bytes(hash256('Programming Bitcoin!'), 'big')
```

Programming Message Signing

To program message signing, we now create a `PrivateKey` class, which will house our secret:

```
class PrivateKey:

    def __init__(self, secret):
        self.secret = secret
        self.point = secret * G  ❶

    def hex(self):
        return '{:x}'.format(self.secret).zfill(64)
```

❶ We keep around the public key, `self.point`, for convenience.

We then create the `sign` method:

```
from random import randint
...
class PrivateKey:
...
    def sign(self, z):
        k = randint(0, N)  ❶
        r = (k*G).x.num  ❷
        k_inv = pow(k, N-2, N)  ❸
        s = (z + r*self.secret) * k_inv % N  ❹
        if s > N/2:  ❺
            s = N - s
        return Signature(r, s) ❻
```

❶ `randint` chooses a random integer from `[0,n)`. Please don't use this function in production, because the random number from this library is not nearly random enough.

❷ r is the x coordinate of kG.

❸ We use Fermat's little theorem again, and n, which is prime.

❹ $s = (z + re)/k$.

❺ It turns out that using the low-s value will get nodes to relay our transactions. This is for malleability reasons.

❻ We return a `Signature` object from the class defined earlier.

Importance of a Unique k

There's an important rule in signatures that utilize a random component like we have here: the k needs to be unique per signature. That is, it cannot get reused. In fact, a k that's reused will result in you revealing your secret! Why?

If our secret is e and we are reusing k to sign z_1 and z_2:

$$kG = (r,y)$$
$$s_1 = (z_1 + re) / k, s_2 = (z_2 + re) / k$$
$$s_1/s_2 = (z_1 + re) / (z_2 + re)$$
$$s_1(z_2 + re) = s_2(z_1 + re)$$
$$s_1z_2 + s_1re = s_2z_1 + s_2re$$
$$s_1re - s_2re = s_2z_1 - s_1z_2$$
$$e = (s_2z_1 - s_1z_2) / (rs_1 - rs_2)$$

If anyone sees both signatures, they can use this formula and find our secret! The PlayStation 3 hack back in 2010 was due to the reuse of the k value in multiple signatures.

To combat this, there is a deterministic k generation standard that uses the secret and z to create a unique, deterministic k every time. The specification is in RFC 6979 and the code changes to look like this:

```
class PrivateKey:
...
    def sign(self, z):
        k = self.deterministic_k(z)  ❶
        r = (k * G).x.num
        k_inv = pow(k, N - 2, N)
        s = (z + r * self.secret) * k_inv % N
        if s > N / 2:
            s = N - s
        return Signature(r, s)

    def deterministic_k(self, z):
        k = b'\x00' * 32
        v = b'\x01' * 32
        if z > N:
            z -= N
        z_bytes = z.to_bytes(32, 'big')
        secret_bytes = self.secret.to_bytes(32, 'big')
        s256 = hashlib.sha256
        k = hmac.new(k, v + b'\x00' + secret_bytes + z_bytes, s256).digest()
        v = hmac.new(k, v, s256).digest()
        k = hmac.new(k, v + b'\x01' + secret_bytes + z_bytes, s256).digest()
        v = hmac.new(k, v, s256).digest()
        while True:
            v = hmac.new(k, v, s256).digest()
```

```
            candidate = int.from_bytes(v, 'big')
            if candidate >= 1 and candidate < N:
                return candidate  ❷
            k = hmac.new(k, v + b'\x00', s256).digest()
            v = hmac.new(k, v, s256).digest()
```

❶ We are using the deterministic *k* instead of a random one. Everything else about `sign` remains the same.

❷ This algorithm returns a candidate that's suitable.

A deterministic *k* will be unique with very high probability. This is because sha256 is collision-resistant, and no collisions have been found to date.

Another benefit from a testing perspective is that the signature for a given *z* and the same private key will be the same every time. This makes debugging much easier and unit tests a lot easier to write. In addition, transactions that use deterministic *k* will create the same transaction every time, as the signature will not change. This makes transactions less malleable (more on that in Chapter 13).

Conclusion

We've covered elliptic curve cryptography and can now prove that we know a secret by signing something. We can also verify that the person with the secret actually signed a message. Even if you don't read another page in this book, you've learned to implement what was once considered "weapons-grade munitions". This is a major step in your journey and will be essential for the rest of the book!

We now turn to serializing a lot of these structures so that we can store them on disk and send them over the network.