Elliptic Curve Cryptography

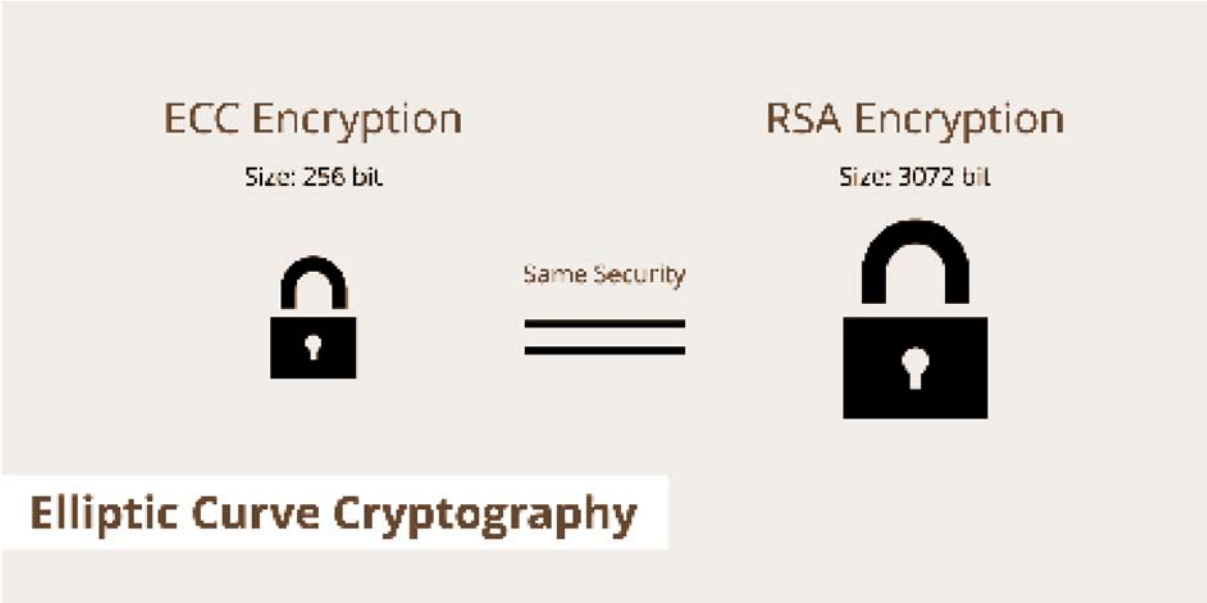
Επιβλέπων καθηγητής
Χρήστος Ξενάκης
Τμήμα Ψηφιακών Συστημάτων

Συνεργασία φοιτητών
Αλίνα Μπαλαούρ
Γιώργος Ανδρής
Βασίλης Ριγγς

Table of Contents

Introduction to ECC

Elliptic Curves Cryptosystems are found in TLS, PGP and SSH. Modern web, IT world, BitCoin and other cryptocurrencies are based on these 3 technologies.



Before ECC become popular, almost all public-key algorithms were based on RSA, DSA, and DH, alternative cryptosystems based on modular arithmetic. RSA and friends are still very important today, and often are used alongside ECC.

Before ECC

All public key algorithms were mainly based on RSA, DSA and DH.

RSA Explanation

Thanks to: http://code.activestate.com/recipes/578838-rsa-a-simple-and-easy-to-read-implementation/

```
################## RSA Implementation ####################

import random
from collections import namedtuple


def get_primes(start, stop): # Return a list of prime numbers in
'range(start, stop)'
    if start >= stop:
          return []

    primes = [2]

    for n in range(3, stop + 1, 2):
        for p in primes:
            if n % p == 0:
                break
        else:
            primes.append(n)

    while primes and primes[0] < start:
        del primes[0]

    return primes


def are_relatively_prime(a, b): # Return True` if a and b are two
relatively prime numbers

    for n in range(2, min(a, b) + 1):
        if a % n == b % n == 0:
            return False
    return True


def make_key_pair(length): # Create a public-private key pair.
            # The key pair is generated from two random prime numbers.
The argument 'length' specifies the bit length of the number 'n'
            # shared between the two keys: the higher, the better.

    if length < 4:
```

```
        raise ValueError('cannot generate a key of length less than 4
(got {!r})'.format(length))

    # First step: find a number 'n' which is the product of two prime
    # numbers ('p' and 'q'). 'n' must have the number of bits specified
    # by 'length', therefore it must be in 'range(n_min, n_max + 1)'.

    n_min = 1 << (length - 1)
    n_max = (1 << length) - 1

    # The key is stronger if 'p' and 'q' have similar bit length. We
choose two prime numbers in 'range(start, stop)'
    # so that the difference of bit lengths is at most 2

    start = 1 << (length // 2 - 1)
    stop = 1 << (length // 2 + 1)
    primes = get_primes(start, stop)

    # Now that we have a list of prime number candidates, randomly
select
    # two so that their product is in 'range(n_min, n_max + 1)'.

    while primes:
        p = random.choice(primes)
        primes.remove(p)
        q_candidates = [q for q in primes

        if n_min <= p * q <= n_max]
            if q_candidates:
                q = random.choice(q_candidates)
                break
            else:
                raise AssertionError("cannot find 'p' and 'q' for a key
of ""length={!r}".format(length))
        stop = (p - 1) * (q - 1) # Second step: choose a number 'e'
lower than '(p - 1) * (q - 1)' which shares no factors with '(p - 1) *
(q - 1)'.

        for e in range(3, stop, 2):
            if are_relatively_prime(e, stop):
                break
            else:
                raise AssertionError("cannot find 'e' with p={!r} ""and
q={!r}".format(p, q))
```

```
        for d in range(3, stop, 2): # Third step: find 'd' such that '(d
* e - 1)' is divisible by '(p - 1) * (q - 1)'
            if d * e % stop == 1:
                break
            else:
                raise AssertionError("cannot find 'd' with p={!r},
q={!r} ""and e={!r}".format(p, q, e))
            return PublicKey(p * q, e), PrivateKey(p * q, d) # Now we
can build and return the public and private keys.


class PublicKey(namedtuple('PublicKey', 'n e')): # Public key which can
be used to encrypt data

      __slots__ = ()
    def encrypt(self, x): # Encrypt the number 'x'

            # The result is a number which can be decrypted only using
the private key

            return pow(x, self.e, self.n)


class PrivateKey(namedtuple('PrivateKey', 'n d')): # Private key which
can be used both to decrypt data

            __slots__ = ()
                        def decrypt(self, x): # Decrypt the number 'x'

                        # The argument 'x' must be the result of the
'encrypt' method of the public key

                        return pow(x, self.d, self.n)


if __name__ == '__main__':
            # Test with known results.
    public = PublicKey(n=2534665157, e=7)
    private = PrivateKey(n=2534665157, d=1810402843)

    assert public.encrypt(123) == 2463995467
    assert public.encrypt(456) == 2022084991
    assert public.encrypt(123456) == 1299565302

    assert private.decrypt(2463995467) == 123
```

```
    assert private.decrypt(2022084991) == 456
    assert private.decrypt(1299565302) == 123456

    # Test with random values.
    for length in range(4, 17):
        public, private = make_key_pair(length)

        assert public.n == private.n
        assert len(bin(public.n)) - 2 == length

        x = random.randrange(public.n - 2)
        y = public.encrypt(x)
        assert private.decrypt(y) == x

        assert public.encrypt(public.n - 1) == public.n - 1
        assert public.encrypt(public.n) == 0

        assert private.decrypt(public.n - 1) == public.n - 1
        assert private.decrypt(public.n) == 0

    import doctest
    doctest.testfile(__file__, globs=globals())
```

Chapter 1

What is an Elliptic Curve? Wolfram MathWorld gives a rather complete definition:

Wolfram MathWorld Definition

Informally, an elliptic curve is a type of **cubic curve** whose solutions are confined to a region of space that is topologically equivalent to a **torus.** The Weierstrass elliptic function (p-function) $P(z;\ g_2,\ g_3)$ describes how to get from this torus to the algebraic form of an elliptic curve.

Formally, an elliptic curve over a field K is a nonsingular cubic curve in two variables, $f(x,\ y) = 0$, with a K-rational point (which may be a point at infinity). The field K is usually taken to be the complex numbers C, reals R, rationals Q, algebraic extensions of Q, p-adic numbers Q_p, or a finite field.

By an appropriate change of variables, a general elliptic curve over a field with field characteristic $\neq 2, 3$, a general cubic curve of type:

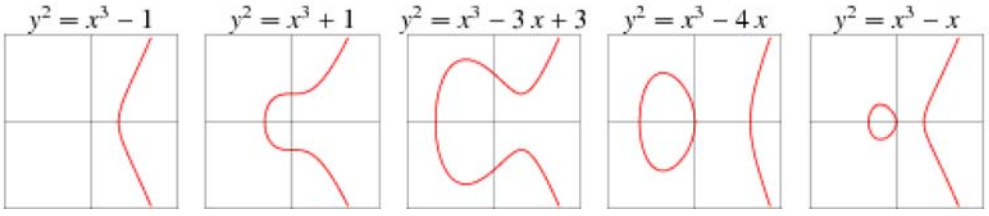
$$Ax^3 + Bx^2y + Cxy^2 + Dy^3 + Ex^2 + Fxy + Gy^2 + Hx + Iy + J = 0 \quad (1)$$

where $A, B, \ldots$ are elements of K, can be written in the form

$$y^2 = x^3 + ax + b \qquad (2)$$

Where the right side of (2) has no repeated factors. Any elliptic curve not of characteristic (field characteristic) 2 or 3, can also be written in Legendre normal form

$$y^2 = x(x-1)(x-\lambda) \text{ (Hartshorne 1999)} \qquad (3)$$



Elliptic curves are illustrated above for various values of a and b. If K has field characteristic three, then the best that can be done is to transform the curve into

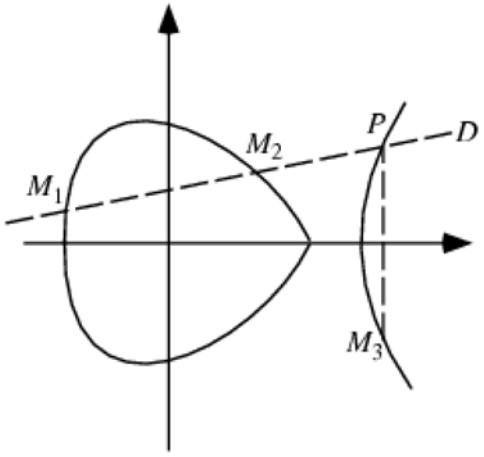
$$y^2 = x^3 + ax^2 + bx + c \qquad (4)$$

The x^2 term cannot be eliminated. If K has field characteristic two, then the situation is even worse. A general form into which an elliptic curve over any K can be transformed is called the Weierstrass form, and is given by:

$$y^2 + ay = x^3 + bx^2 + cxy + dx + e$$,where a, b, c, d, e are elements of K

Luckily Q, R and C fields, all have field characteristic zero. An elliptic curve of the form $y^2 = x^3 + n$ for n being an integer, is known as a Mordell curve.

Whereas conic sections can be parameterized by the rational functions, elliptic curves cannot. The simplest parameterization functions are elliptic functions. Abelian varieties can be viewed as generalizations of elliptic curves.



If the underlying field of an elliptic curve is algebraically closed, then a straight line cuts an elliptic curve at three points (counting multiple roots at points of tangency). If two are known, it is possible to compute the third. If two of the intersection points are K-rational, then so is the third. Mazur and Tate (1973/74) proved that there is no elliptic curve over Q having a rational point of order 13.

Let (x_1, y_1) and (x_2, y_2) be two points on an elliptic curve E with elliptic discriminant

$$\Delta_E = -16(4a^3 + 27b^2) \text{ with } \Delta_E \neq 0$$

A related quantity known as the j-invariant of E is defined as:

$$j(E) = \frac{2^8 3^3 a^3}{4a^3 + 27b^2}$$

Now define

$$\lambda = \begin{cases} \dfrac{y_1 - y_2}{x_1 - x_2} & \text{for } x_1 \neq x_2 \\ \dfrac{3x_1^2 + a}{2y_1} & \text{for } x_1 = x_2 . \end{cases}$$

Then the coordinates of the third point are

$$x_3 = \lambda^2 - x_1 - x_2$$
$$y_3 = \lambda(x_3 - x_1) + y_1$$

For elliptic curves over Q-field, Mordell proved that there are a finite number of integral solutions. The Mordell-Weil theorem says that the group of rational points of an elliptic curve over Q is finitely generated. Let the roots of y^2 be r_1, r_2, r_3. The discriminant is then

$$\Delta = k(r_1 - r_2)^2(r_1 - r_3)^2(r_2 - r_3)^2$$

The amazing Taniyama-Shimura conjecture states that all rational elliptic curves are also modular. This fact is far from obvious, and despite the fact that the conjecture was proposed in 1955, it was not even partially proved until 1995. Even so, Wiles' proof for the semistable case surprised most mathematicians, who had believed the conjecture unassailable. As a side benefit, Wiles' proof of the Taniyama-Shimura conjecture also laid to rest the famous and thorny problem which had baffled mathematicians for hundreds of years, Fermat's last theorem.

Curves with small j-conductors are listed in Swinnerton-Dyer (1975) and Cremona (1997). Methods for computing integral points (points with integral coordinates) are given in Gebel et al. and Stroeker and Tzanakis (1994). The Schoof-Elkies-Atkin algorithm can be used to determine the order of an elliptic curve E/F_p over the finite field F_p.

Web Resource: Weisstein, Eric W. "Elliptic Curve." From MathWorld–A Wolfram

Simplified Definition

In order to fully understand what an elliptic curve is after all, we are going to use a simplified, non-formal definition. So, an elliptic curve will simply be the set of points described by the following equation:

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

Where $4a^3 + 27b^2 \neq 0$ in order to exclude singular curves. The equation above is called as the ***Weierstrass normal form*** for elliptic curves.

Valid elliptic curves

Different shapes for different elliptic curves (b=1, a varying from 2 to -3)

Non valid elliptic curves

Types of singularities: on the left, a curve with a cusp $(y^2 = x^3)$. On the right, a curve with a self-intersection $(y^2 = x^3 - 3x + 2)$.
None of them is a valid elliptic curve.

Depending on the value of a and b, elliptic curves may assume different shapes on the plane. As it can be easily seen and verified, elliptic curves are symmetric about the x-axis.

For our aims, we will also need a point at infinity (aka ideal point) to be part of our curve. From now on, we will denote out point at infinity with the symbol 0 (zero).

If we want to explicitly take into account the point at infinity, we can refine our definition of elliptic curve as follows:

$$\left\{(x,y) \in R^2 \mid y^2 = x^3 + ax + b,\ 4a^3 + 27b^2 \neq 0\right\} \cup \{0\}$$

Groups

A group in mathematics is a set for which we have defined a binary operation that we call "addition" and indicate with the symbol +. In order for the set **G** to be a group, addition must be defined so that it respects the following four properties:

1. **Closure**: if $a,\ b$ are members of **G**, then $a + b$ is a member of **G** as well
2. **Associativity**: $(a + b) + c = a + (b + c)$
3. There exists an **identity element** 0 such that $a + 0 = 0 + a = a$
4. Every element has an **inverse**, that is: for every a there is a b such that $a + b = 0$

Now, if we add a fifth requirement:

5. **Commutativity**: $a + b = b + a$ then the group is called ***abelian* group**.

With the usual notion of addition, the set of integer numbers **Z** is a group (moreover, it's an abelian group). The set of natural numbers **N** however is not a group, as the fourth property can't be satisfied.

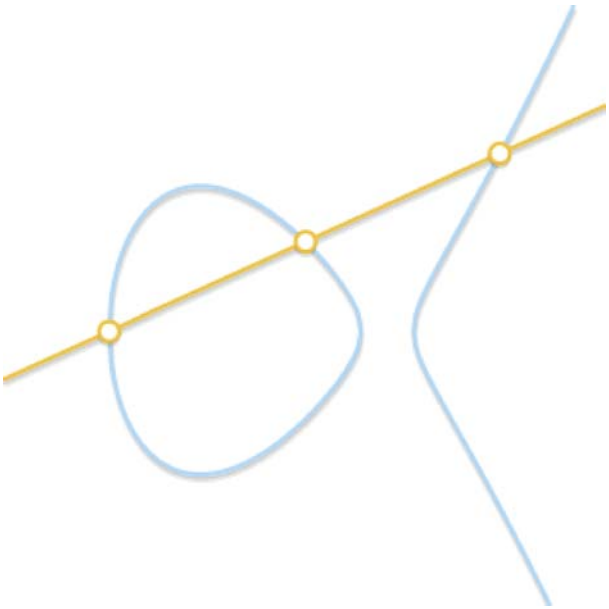
It is convenient enough to use groups, because if we can demonstrate that those four properties hold, we get some other properties for free.

For example: The identity element is unique, also the inverses are unique, that is: for every a there exists only one b such that $a + b = 0$ (and we can write b as $-a$). Either directly or indirectly, these and other facts about groups will be very important for us later.

The group law for elliptic curves

We can define a group over elliptic curves. Specifically:

- The elements of the group are the points of an elliptic curve
- The **identity element** is the point at infinity 0
- The **inverse** of a point P is the one symmetric about the x-axis
- **Addition** is given by the following rule; given three aligned, non-zero points $P, Q \text{ and } R$ their sum is $P + Q + R = 0$



The sum of three aligned points is 0

Note that with the last rule, we only require three aligned points, and three points are aligned without respect to order. This means that, if $P,\ Q \text{ and } R$ are aligned, then

$$P + (Q + R) = Q + (P + R) = R + (P + Q) = ... = 0$$

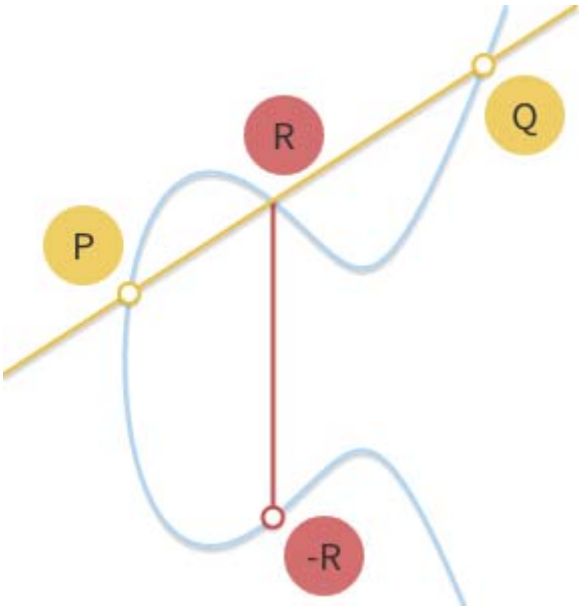
This way, we have intuitively proved that our + operator is both **associative** and **commutative**: we are in an ***abelian*** group.

But how do we actually compute the sum of two arbitrary points?

Geometric addition

Thanks to the fact we are in an *abelian group*, we can write $P+Q+R=0$ as $P+Q=-R$. This equation - in this form, lets us derive a geometric method to compute the sum between two points P, Q:

If we draw a line passing through P and Q, this line will intersect a third point on the curve, which is R (this is simplified by the fact that P, Q, R are aligned). If we take the inverse of this point, $-R$ we have found the result of $P+Q$.

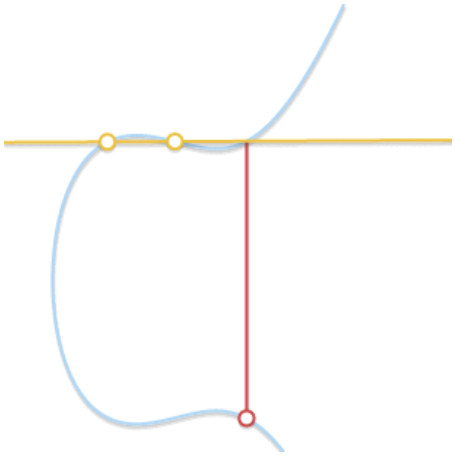


*Draw a line through P and Q. The line intersects a third point R. The point symmetric to it, $-R$ is the result of $P+Q$.

This geometric method works, but needs some refinement. Particularly we need to answer some questions regarding:

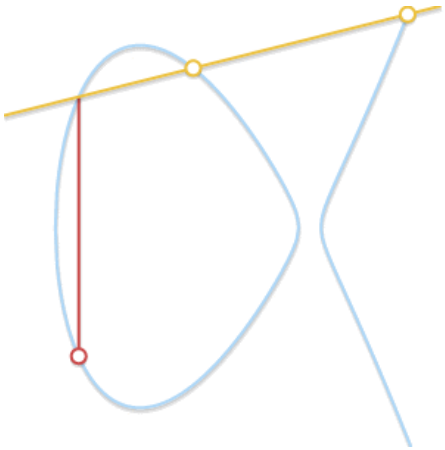
- What if $P=0$ or $Q=0$?
  Certainly we can't draw any line (0 is not on the xy-plane). But given that we defined 0 as the identity element $P+0=P$ and $0+Q=Q$, for any P and Q.
- What if $P=-Q$?
  In this case the line going through the two points is vertical and doesn't intersect any third point. But if P is the inverse of Q, then we have $P+Q=P+(-P)=0$ from the definition of inverse.

- What if $P = Q$?
  In this case there are infinitely many lines passing through the point. Here things start getting a bit more complicated. But consider a point $Q' \neq P$
  What happens if we make Q' approach P, getting closer and closer to it?



  As Q' tends towards P, the line passing through P and Q' becomes tangent to the curve. In the light of this, we can say that $P + P = -R$, where R is the point of intersection between the curve and the line tangent to the curve in P.

- What if $P \neq Q$ but there is no third point R?
  We are in a case very similar to the previous one. In fact, we are in the case where the line passing through P and Q is tangent to the curve.



  Let's assume that P is the tangency point. In the previous case, we would have written $P + P = -Q$. That equation now becomes $P + Q = -P$. If on the other hand Q was the tangency point, the correct equation would have been $P + Q = -Q$.

Algebraic addition

If we want a computer to perform point addition, we need to turn the geometric method into an algebraic one. Transforming the rules described above into a set of equations may seem simple, but actually is can be really dull because it requires solving cubic equations. For this reason, we are going to report only the results.

First things first, we already know that $P + (-P) = 0$ and $P + 0 = 0 + P = P$. In our equations we will forget about those two cases and we will only consider two ***non-zero***, ***non symmetric points*** $P = (x_P, y_P)$ and $Q = (x_Q, y_Q)$.

If P and Q are distinct $(x_P \neq x_Q)$ then the line through them has a slope (κλίση):

$$m = \frac{y_P - y_Q}{x_P - x_Q}$$

The intersection of this line with the elliptic curve is a third point $R = (x_R, y_R)$:

$$x_R = m^2 - x_P - x_Q$$
$$y_R = y_P + m(x_R - x_P)$$

Or equivalently:

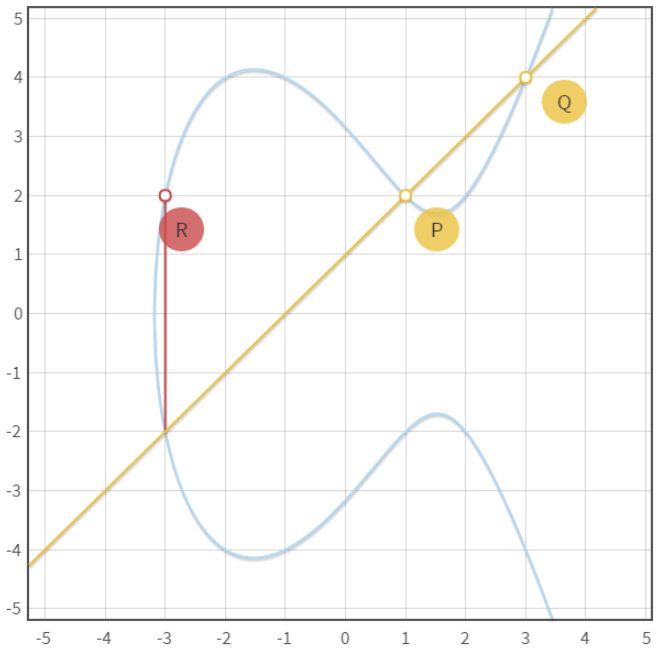
$$y_R = y_Q + m(x_R - x_Q)$$

Hence $(x_P, y_P) + (x_Q, y_Q) = (x_R, -y_R)$ *!remember that* $P + Q = -R$.

If we want to check whether the result is right, we would have had to check whether R belongs to the curve and whether P, Q and R are aligned. Checking whether the points are aligned is of little value, and checking whether R belongs to the curve requires cubic equations, which we are not good at, at all.

What we can do, is go through trial and error and give the points values of: $P = (1,\ 2)$ and $Q = (3,\ 4)$ over the curve $y^2 = x^3 - 7x + 10$, their sum is $P + Q = -R = (-3,\ 2)$. Let's see if our equations agree:

$$m = \frac{y_P - y_Q}{x_P - x_Q} = \frac{2-4}{1-3} = 1$$
$$x_R = m^2 - x_P - x_Q = 1^2 - 1 - 3 = -3$$
$$y_R = y_P + m(x_R - x_P) = 2 + 1(-3 - 1) = -2$$
$$y_R = y_Q + m(x_R - x_Q) = 4 + 1(-3 - 3) = -2$$



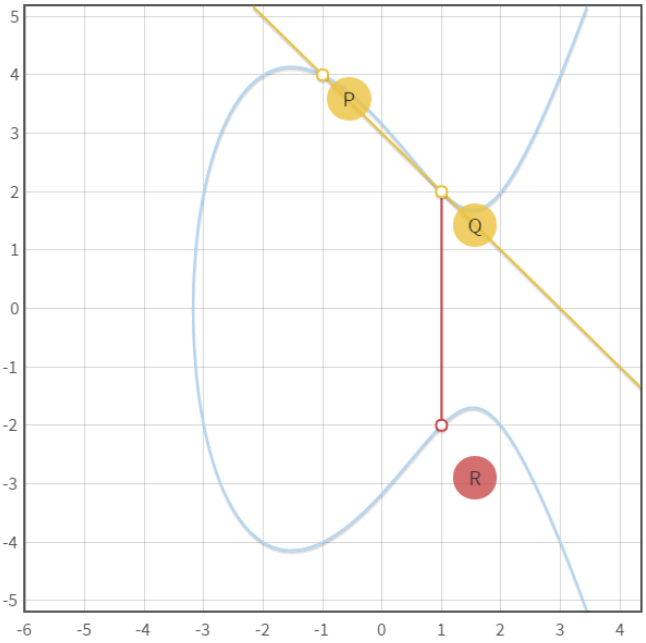
Curve:	a	-7	b	10
P:	x	1	y	2
Q:	x	3	y	4
R = *P* + *Q*:	x	-3	y	2

Point addition over the elliptic curve $y^2 = x^3 - 7x + 10$ in $\mathbb{R}$.

Judging by the results, our assumption was correct. The equations will still work even if one of P or Q is a tangency point.

Let's try assigning the values $P = (-1, 4)$ and $Q = (1, 2)$ at the above equations.

$$m = \frac{y_P - y_Q}{x_P - x_Q} = \frac{4-2}{-1-1} = -1$$
$$x_R = m^2 - x_P - x_Q = (-1)^2 - (-1) - 1 = 1$$
$$y_R = y_P + m(x_R - x_P) = 4 + (-1)(1 + 1) = 2$$



Curve: a -7 b 10
P: x -1 y 4
Q: x 1 y 2
R = P + Q: x 1 y -2

Point addition over the elliptic curve $y^2 = x^3 - 7x + 10$ in $\mathbb{R}$.

We get the same result $P + Q = (1, -2)$ as given from the visual tool (see pictures).

What if $P = Q$ though? The equations for x_R and y_R stay the same, but given that $x_P = x_Q$, we must use a different equation for the slope:

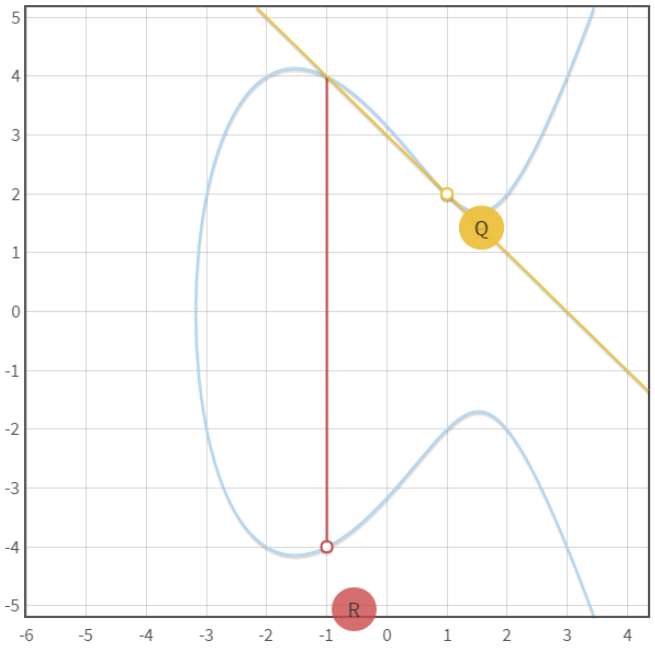
$$m = \frac{3x_P^2 + a}{2y_P}$$

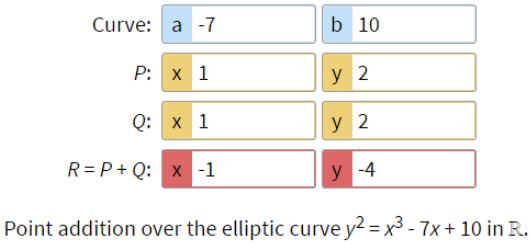
The expression for m is the first derivative of:

$$y_P = \pm\sqrt{x_P^3 + ax_P + b}$$

To prove the validity of this result, it is enough to check that R belongs to the curve and that the line passing through P and R has only two intersections with the curve. Let's try and prove with an example of $P = Q = (1, 2)$

$$m = \frac{3x_P^2 + a}{2y_P} = \frac{3 \cdot 1^2 - 7}{2 \cdot 2} = -1$$
$$x_R = m^2 - x_P - x_Q = (-1)^2 - 1 - 1 = -1$$
$$y_R = y_P + m(x_R - x_P) = 2 + (-1) \cdot (-1 - 1) = 4$$





After all we get that $P + P = -R = (-1, 4)$, which is correct!

Although the procedure to derive them can be really tedious, our equations are pretty compact. This is thanks to Weierstrass normal form: without it, these equations could have been really long and complicated!

Scalar multiplication

Scalar multiplication is defined as:

$$\underbrace{nP = P + P + ... + P}_{\text{n times}}$$

Where n is a natural number. Written in that form it may seem that computing nP requires n additions. If n has k binary digits, then our algorithm would be $O(2^k)$ (O notation /complexity), which is not really good. There exist faster algorithms, such as the “Double and Add” algorithm.

Double and Add

Take $n = 151.$ Its binary representation is 10010111_2 . By analyzing this binary representation into a sum of powers of 2 we get:

$$151 = 1 \cdot 2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 2^7 + 2^4 + 2^2 + 2^1 + 2^0$$

In view of this, we can write:

$$151 \cdot P = 2^7P + 2^4P + 2^2P + 2^1P + 2^0P$$

In a few words, what the double and add algorithm does is:

- ➔ Take P
- ➔ Double it, so that we get $2P$
- ➔ Add $2P$ to P (in order to get the result of $2^1P + 2^0P$)
- ➔ Double $2P$ so that we get 2^2P
- ➔ Add it to our result (so that we get $2^2P + 2^1P + 2^0P$)
- ➔ Double 2^2P to get 2^3P
- ➔ Don’t perform any addition involving 2^3P
- ➔ Double 2^3P to get 2^4P
- ➔ Add it to our result (so that we get $2^4P + 2^2P + 2^1P + 2^0P$)
- ➔ …

*In the end we can compute $151 \cdot P$ performing just seven doublings and four additions!

Python script for Double and Add algorithm

```python
def bits(n):
    """
    Generates the binary digits of n, starting
    from the least significant bit.

    bits(151) -> 1, 1, 1, 0, 1, 0, 0, 1
    """
    while n:
        yield n & 1
        n >>= 1

def double_and_add(n, x):
    """
    Returns the result of n * x, computed using
    the double and add algorithm.
    """
    result = 0
    addend = x

    for bit in bits(n):
        if bit == 1:
            result += addend
        addend *= 2

    return result
```

If doubling and adding are both $O(1)$ operations, then this algorithm is $O(logn)$ (or $O(k)$ if we consider the bit length), which is pretty good - at least much better than the initial one: $O(n)$ algorithm.

Logarithm

Given n and P, we now have at least one polynomial time algorithm for computing $Q = nP$. But what about the other way round - what if we know Q and P and we we need to find out n? This problem is known as the logarithm problem. We call it the “logarithm” instead of “division” for conformity with other cryptosystems (where instead of multiplication we have exponentiation) .

We can’t demonstrate any “easy” algorithm right now, but we could try messing around with the multiplication tool and see some patterns. For example, take the curve $y^2 = x^3 - 3x + 1$ and the point $P = (0, 1)$. We can immediately verify that:

- If n is odd, nP is on the curve on the left semiplane
- If n is even, nP is on the curve on the right semiplane.

If we experimented more, we could probably find more patterns that eventually could lead us to write an algorithm for computing the logarithm on that curve efficiently.

But there's a variant of the logarithm problem: the ***discrete logarithm problem***. As we will see in the next chapter, if we reduce the domain of our elliptic curves, scalar multiplication remains "easy", while the discrete logarithm becomes a "hard" problem. This duality is the key brick of elliptic curve cryptography.

Chapter 2

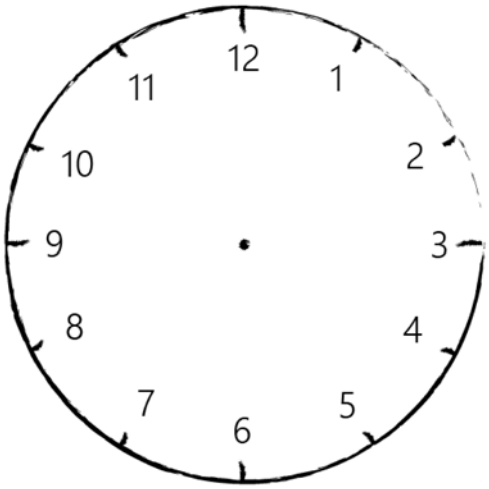
In the previous chapter we've seen how elliptic curves over real numbers can be used in order to define a group. Specifically, we have defined 'rules' about:

- **Point addition**: for 3 aligned points, their sum equals zero - $P + Q + R = 0$
- **Geometric** method: for computing point additions
- **Algebraic** method: for computing point additions too
- **Scalar multiplication**: $nP = P + P + ... + P$ and presented an *"easy"* algorithm (double-and-add) so as to explain its functionality on scalar multiplication.

In this chapter we are going to restrict our curves to ***finite fields*** rather than a set of real numbers, and see what's going to change.

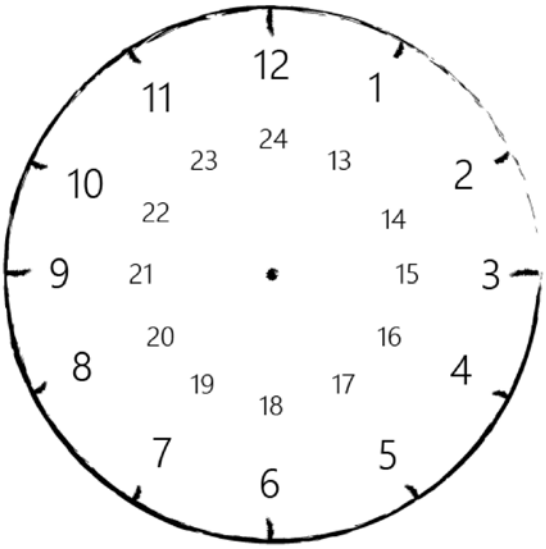
Introduction to modular arithmetic

For starters, let's briefly examine the concept of modular arithmetic; the best way to follow up, is to think of a clock. Modular arithmetic deals with repetitive cycles of numbers and remainders. The time you use everyday is a cycle of 12 hours, divided up into a cycle of 60 minutes.



The numbers go from 1 to 12, but when you get to "13 o'clock", it actually becomes 1 o'clock again (think of how the 24 hour clock numbering works). So 13 becomes 1, 14 becomes 2, and so on.

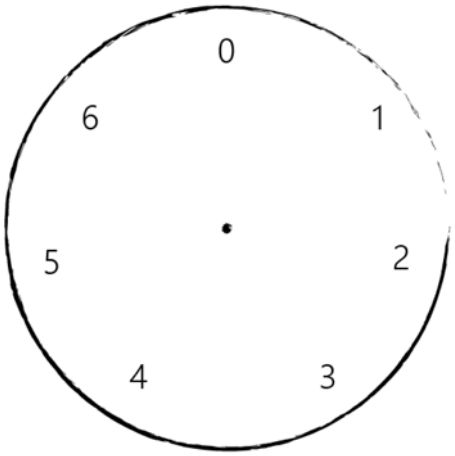
This can keep going, so when you get to "25 o'clock", you are actually back round to where 1 o'clock is on the clock face (and also where 13 o'clock was too).



So in this clock world, you only care where you are in relation to the numbers 1 to 12. In this world, 1,13,25,37,… are all thought of as the same thing, as are 2,14,26,38,… and so on.

What we are saying is "*13 equals 1 plus some multiple of 12*", and "*38 equals 2 plus some multiple of 12*", or, alternatively, "the remainder when you divide 13 by 12 is 1" and "the remainder when you divide 38 by 12 is 2". The way we write this mathematically is $13 \equiv 1 \ mod \ 12$, $38 \equiv 2 \ mod \ 12$, and so on. This is read as "1*3 is congruent to 1 mod* (or modulo) *12*" and "*38 is congruent to 2 mod 12*".

But you don't have to work only in mod 12 (that's the technical term for it). For example, you could work mod 7, or mod 46 instead if you wanted to (just think of clocks numbered from 1 to 7 and 1 to 46 respectively; every time you get past the biggest number, you reset to 1 again).



Let's go back to the normal clock face with the numbers 1 to 12 on it for a moment. Mathematicians usually prefer to put a 0 where the 12 would normally be, so that you would usually write (for example) $24 \equiv 0 \ mod \ 12$ rather than $24 \equiv 12 \ mod \ 12$, although both of these are correct. That is, we think of a normal clock face as being numbered from 0 to 11 instead. This makes sense: we'd normally say that 24 leaves a remainder of 0 when we divide by 12, rather than saying it leaves a remainder of 12 when we divide by 12!

Let's be a bit more formal. In general, if you are working in mod n (where n is any whole number), we write $a \equiv b \ \ mod \ n$ if a and b leave the same remainder when you divide them by n. This is the same as saying that we write $a \equiv b \ \ mod \ n$ if n divides $a - b$. (Look at what we did earlier to see that this definition fits with our examples above.)

So far, we've only talked about notation. Now let's do some maths, and see how congruences (what we've described above) can make things a bit clearer.

Here are some useful properties. We can add congruences. That is, if $a \equiv b \ mod \ n$ and $c \equiv d \ mod \ n$, then $a + b \ \equiv (b + d) \ mod \ n$. Why is this? Well, $a \equiv b \ mod \ n$ means that $a = b + kn$, where k is an integer. Similarly, $c \equiv d \ mod \ n$ means that $c = d + ln$, where l is an integer. So $a + c = (b + kn) + (d + ln) = (b + d) + (k + l)n$, so $a + c \equiv (b + d) \ mod \ n$.
For example, $17 \equiv 4 \ mod \ 13$, and $42 \equiv 3 \ mod \ 13$, so $17 + 42 \equiv 4 + 3 \equiv 7 \ mod \ 13$. Note that both of the congruences that we're adding are $mod \ n$, and so is the answer - we don't add the moduli.

Now you prove that if $a \equiv b \ mod \ n$ and $c \equiv d \ mod \ n$ then $a - c \equiv (b - d) \ mod \ n$. Also, prove that we can do something similar for multiplication: if $a \equiv b \ mod \ n$ and $c \equiv d \ mod \ n$, then $ac \equiv bd \ mod \ n$. You can prove this in the same way that we used

above for addition. Again, both of the congruences that we're multiplying are $mod\ n$, and so is the answer - we don't multiply the moduli.

Division is a bit more tricky: you have to be really careful. Here's an example of why: $10 \equiv 2\ mod\ 8$. But if we "divide both sides by 2", we'd have $5 \equiv 1\ mod\ 8$, which is clearly nonsense! To get a true congruence, we'd have to divide the 8 by 2 as well: $5 \equiv 1\ mod\ 4$ is fine. Why? Well, $a \equiv b\ mod\ n$ means that $a = b + kn$ for some integer n. But now this is a normal equation, and if we're going to divide a by something, then we have to divide all of the right-hand side by 2 as well, including kn. In general, it's best not to divide congruences; instead, think about what they really mean (rather than using the shorthand) and work from there.

Things are quite special if we work $mod\ p$, where p is prime, because then each number that isn't $0\ mod\ p$ has what we call an inverse (or a multiplicative inverse , if we're being fancy). What that means is that for each $a \equiv 0\ mod\ p$, there is a b such that $ab \equiv 1\ mod\ p$.

Let's think about an example. We'll work $mod\ 7$. Then really the only non-zero things are 1,2,3,4,5 and 6 (because every other whole number is equivalent to one of them or 0). So let's find inverses for them. Well, 1 is pretty easy: $1 \times 1 \equiv 1\ mod\ 7$. What about 2? $2 \times 4 \equiv 1\ mod\ 7$. So 4 is the inverse of 2. In fact, we can also see from this that 2 is the inverse of 4 - so that's saved us some work! $3 \times 5 \equiv 1\ mod\ 7$, so 3 and 5 are inverses. And finally, $6 \times 6 \equiv 1\ mod\ 7$, so 6 is the inverse of itself. So yes, each of the non-zero elements mod 7 has an inverse. To prove this, things are going to get a tiny bit more tricky, so I'm going to save the proof for the end and first give an example of using congruences to do useful mathematics.

Suppose we're given the number 11111111 and someone asks us whether it's divisible by 3. We could try to actually divide it. But you probably know a much easier method: we add up the digits and see whether that's divisible by 3. There's a whole article about this sort of divisibility test here . Let's prove this using congruence notation.

Suppose that our number is $a_n 10^n + a_{n-1} 10^{n-1} + ... + 10a_1 + a_0$ so it looks like $a_n a_{n-1} ... a_1 a_0$. Then the sum of its digits is $a_n + a_{n-1} + ... + a_1 + a_0$. We would like to prove that $a_n 10^n + a_{n-1} 10^{n-1} + ... + 10a_1 + a_0$ is divisible by 3 if and only if $a_n + a_{n-1} + ... + a_1 + a_0$ is divisible by 3. Now we notice that $10 \equiv 1\ mod\ 3$ so $10 \times 10 \equiv 1\ mod\ 3$ and more generally $10^k \equiv 1\ mod\ 3$ for all k. Using our results from earlier about adding and multiplying congruences, we discover:

$$a_n 10^n + a_{n-1} 10^{n-1} + ... + 10a_1 + a_0 \equiv a_n + a_{n-1} + ... + a_1 + a_0 \ mod\ 3$$

So, if our number is divisible by 3, then certainly so is the sum of its digits, and vice versa (as we wanted!). The congruence notation hasn't really done any of the maths for us, but it's hopefully made it a bit easier to write out the proof clearly. See whether you can use the notation to prove any of the other divisibility test in that article.

Last but not least, we are going to show that if a and n have no common factors, then a has a multiplicative inverse $mod\ n$ (reminder: that means a number b such that $a \times b \equiv 1\ mod\ n$). In particular, if n is prime, then its only factor apart from 1 is itself, so saying "a and n share no common factors" is just the same as saying "a isn't divisible by n", that is, $a \neq 0\ mod\ p$ this is what we had above.

As a conclusion, if a and n share no common factors, then we can find x and y such that $ax + ny = 1$ (Bezout's theorem). We can rewrite this as $ax = 1 - yn$ and now let's use the congruence notation from earlier; we have $ax \equiv 1\ mod\ n$. So now x is the multiplicative inverse of $a\ mod\ n$ and we'are done!

.

The field of integers modulo p

A finite field is a set with a finite number of elements. An example of finite fields is the set of integers modulo p, where p is a prime number. It is denoted as $Z/p,\ GF(p)$ or F_p. We will use the latter notation.

In fields, we have two binary operations: addition $(+)$ and multiplication $(\cdot)$. Both are closed, associative and commutative. For both operations there exist a unique identity element, and for every element there is a unique inverse element. Finally, multiplication is distributive over the addition:

$$x \cdot (y + z) = x \cdot y + x \cdot z$$

The set of integers modulo p consists of all the integers included in $[0,\ p-1]$. Addition and multiplication work as in modular arithmetic (aka 'clock arithmetic'). Take a look at a few examples of operations in F_{23}:

- Addition: $(18 + 9)\ mod\ 23 = 4$
- Subtraction: $(7 - 14)\ mod\ 23\ = 16$
- Multiplication: $4 \cdot 7\ mod\ 23 = 5$
- Additive inverse: $-5\ mod\ 23 = 18$
  Proof - $(5 + (-5))\ mod\ 23 = (5 + 18)\ mod\ 23 = 0$
- Multiplicative inverse: $9^{-1}\ mod\ 23 = 18$
  Proof - $9 \cdot 9^{-1}\ mod\ 23 = 9 \cdot 18\ mod\ 23 = 1$

As we already said, the integers modulo p are a field, and therefore all the properties listed above hold. Note that the requirement for p to be prime is important! The set of integers $modulo\ 4$ is not a field: 2 has no multiplicative inverse (for example, the equation $2 \cdot x\ mod\ 4 = 12 \cdot x\ mod\ 4 = 1$ has no solutions).

Division modulo p

We will soon define elliptic curves over F_p, but before doing so we need a clear idea of what x/y means in F_P. Simply put $x/y = x \cdot y^{-1}$, or in plain words; x over y is equal to $x\ times$ the multiplicative inverse of y.

The above gives us a basic method to perform division:
Find the multiplicative inverse of a number and then perform a single multiplication.

Computing the multiplicative inverse can be done with use of the "Extended Euclidean Algorithm", which is of $O(logp)\ or\ O(k)\ for\ bit\ length$ in the worst case scenario.

Extended Euclidean Algorithm Implementation

```c
#include <stdio.h>
#include <math.h>

int main(void)
{
  long a = 1180, b = 482, gcd, x, y;

  extendedGCD(a, b, &x, &y, &gcd);
  printf(" gcd = %ld\n x = %ld\n y = %ld\n", gcd, x, y);
  return 0;
}


void extendedGCD(long a, long b, long *x,long *y, long *gcd){

  long q, r, x1, x2, y1, y2;

  if (b == 0) {
    *gcd = a;
     *x = 1;
     *y = 0;
    return;
  }

  x2 = 1;
  x1 = 0;
  y2 = 0;
  y1 = 1;
```

```
  while (b > 0) {
    q = a / b;
      r = a - (q * b);
    *x = x2 - (q * x1);
      *y = y2 - (q * y1);
    a = b;
      b = r;
    x2 = x1;
      x1 = *x;
      y2 = y1;
      y1 = *y;
  }

  *gcd = a;
  *x = x2;
  *y = y2;
}
```

Elliptic curves in F_p.

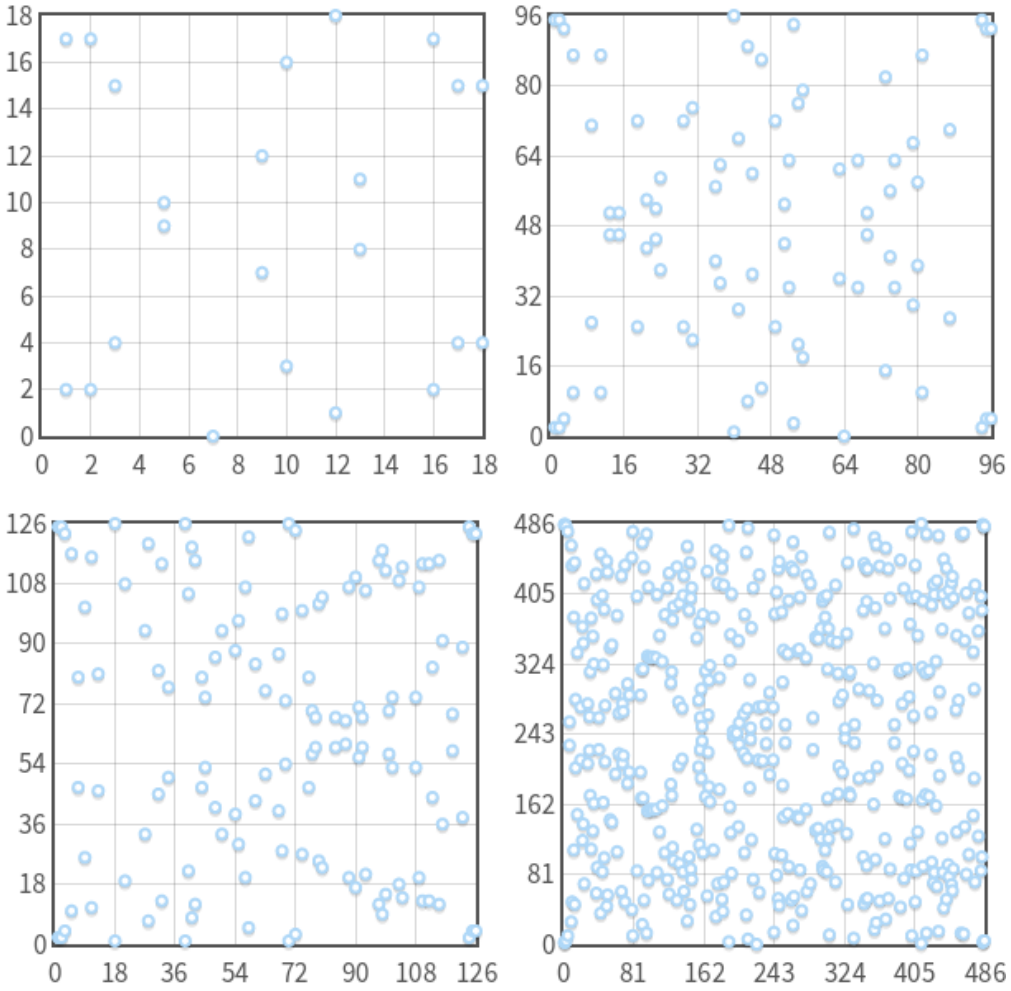
The set of points that till now we described as:

$$\{(x,y)\in R^2 \mid y^2 = x^3 + ax + b,\ 4a^3 + 27b^2 \neq 0\} \cup \{0\}$$

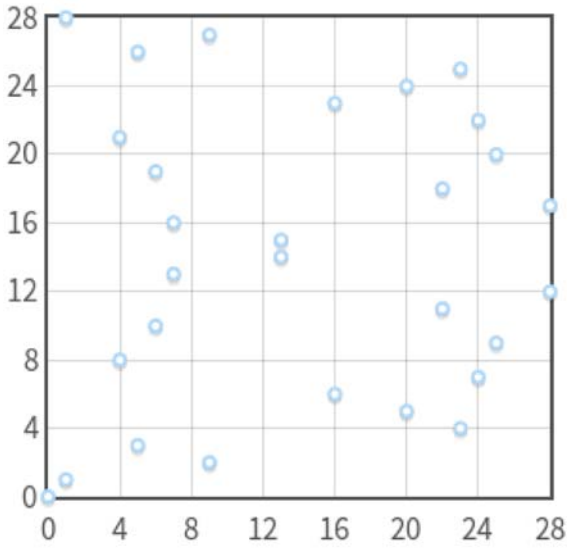
Becomes:

$$\{(x,y)\in (F_p)^2 \mid y^2 \equiv x^3 + ax + b\ (mod p),\ 4a^3 + 27b^2 \neq 0\ (mod p)\} \cup \{0\}$$

Where 0 is the point at infinity and, a and b are two integers in F_p.



The curve $y^2 \equiv x^3 - 7x + 10\ (mod p)$ with $p = 19, 97, 127, 487$.
Note that for every x, there are at most two points.
Also note the symmetry about $y = p/2$.



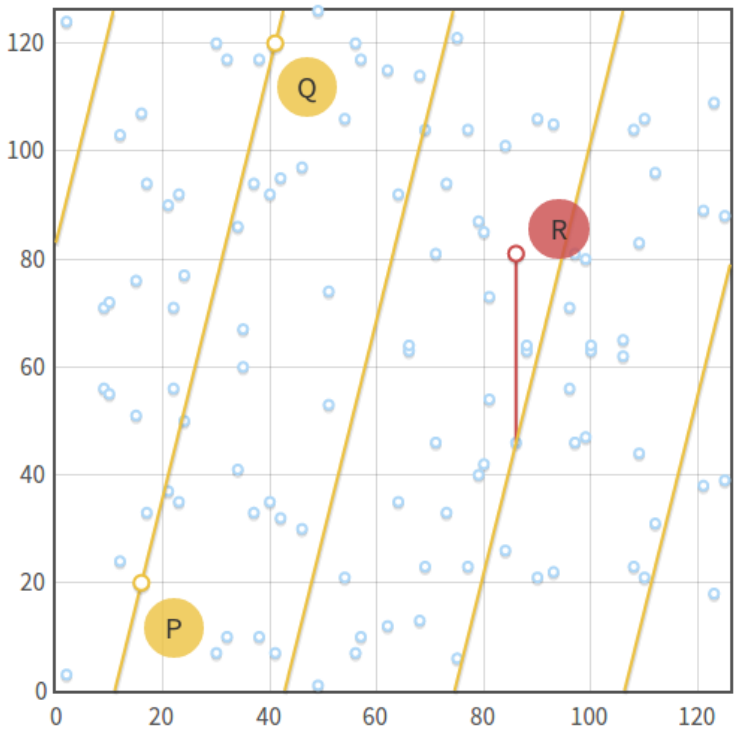
The curve $y^2 \equiv x^3 \ (mod\ 29)$ is singular and has a triple point in $(0,0)$. Thus it is not a valid elliptic curve.

What previously was a continuous curve, is now a set of disjoint points in the xy -plane. We can prove that, even if we have restricted our domain, elliptic curves in F_p still form an Abelian group.

Point addition

We need to change a bit our definition of addition in order to make it work in F_p. With real numbers R, we said that the sum of three aligned points equals zero $(P + Q + R = 0)$. But what does it mean for "*Three points to be aligned in* F_p"?

Three points can be called to be aligned if there is a line that connects all of them. However, lines in F_p are not the same as the lines in R. Informally, a line in F_p is the set of points (x, y) that satisfy the equation $ax + b + c \equiv 0\ (mod p)$.



Point addition over the curve $y^2 \equiv x^3 - x + 3\ (mod\ 127)$, with $P = (16, 20)$ and $Q = (41, 120)$. Note how the line $y = 4x + 83\ (mod\ 127)$ that connects the points "repeats" itself in the plane.

Given that we are in a group, point addition retains the properties we already know:

- $Q + 0 = 0 + Q = Q$
- Given a non-zero point Q, the inverse $-Q$ is the point having the same abscissa but opposite ordinate. Or $-Q = (x_Q, -y_Q\ mod\ p$. For example, if a curve in F_{29} has a point $Q = (2, 5)$, the inverse is $-Q = (2, -5\ mod\ 29) = (2, 24)$.
- $P + (-P) = 0$

Algebraic sum

The equations for calculating point additions are exactly the same as we described in the previous chapter, except for the fact we need to add $modp$ at the end of every expression. Therefore, given $P = (x_P, y_P),\ Q = (x_Q, y_Q)\ and\ R = (x_R, y_R)$, we can calculate $P + Q = -R$ as follows:

$$x_R = (m^2 - x_P - x_Q)\ mod\ p$$
$$y_R = [y_P + m(x_R - x_P)]\ mod\ p\ = [y_Q + m(x_R - x_Q)]\ mod\ p$$

If $P \neq Q$ then the slope m assumes the form:

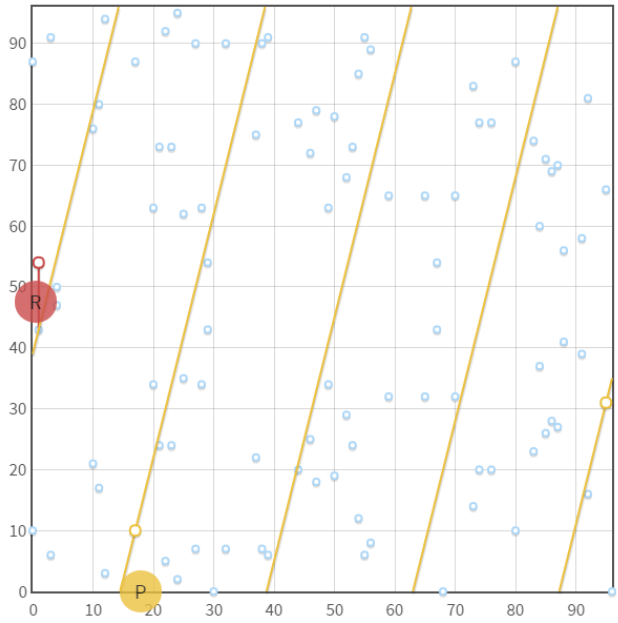
$$m = (y_P - y_Q)(x_P - x_Q)^{-1}\ mod\ p$$

Else, if $P = Q$ then:

$$m = (3x_p{}^2 + a)(2y_p)^{-1}\ mod\ p$$

These equations work in every field, finite or infinite (with the exception of F_2, F_3, which are special cased). We can't prove the group law generally, as it involves complex mathematical concepts. However, we are quoting a proof from Stephan Friedl - who uses only elementary concepts - for whoever might be interested in the last section of the document.

We won't define a geometric method: in fact, there are a few problems with that. For example, in the previous chapter, we said that to compute $P + P$ we needed to take the tangent to the curve in P. But without continuity, the word "tangent" does not make any sense. We can workaround this and other problems, however a pure geometric method would just be too complicated and not practical at all.



Curve: a 2 b 3

Field: p 97

P: x 17 y 10

Q: x 95 y 31

$R = P + Q$: x 1 y 54

Point addition over the elliptic curve $y^2 = x^3 + 2x + 3$ in $\mathbb{F}_{97}$.

The curve has 100 points (including the point at infinity).

The order of an elliptic curve group

Try to recall how we said that an elliptic curve defined over a finite field has a finite number of points - but how many points are we talking about exactly?

Firstly, let's call this number of points in a group the "order of the group". Trying all the possible values for x from 0 to $p-1$ is not a feasible way to count the points, as it would require $O(p)$ steps, which is hard especially when p is a large prime.

There is a faster algorithm for computing the order, Schoof's algorithm - which runs in polynomial time.

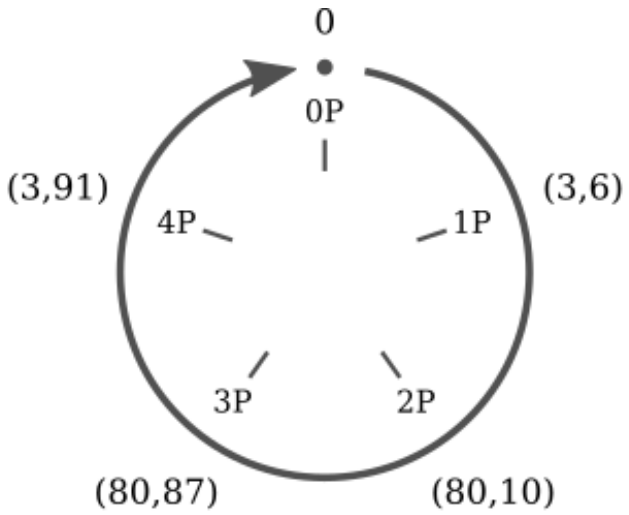
Scalar multiplication and cyclic subgroups

As with reals, multiplication can be defined as:

$$nP = P + P + ... + P \ (n \ times \ P)$$

Here we can use the **double and add** algorithm to perform multiplication in $O(logn)$ steps (or $O(k)$, where k is the number of bits of n). Multiplication over points for elliptic curves in F_p has an interesting point:

Let's take the curve $y^2 \equiv x^3 + 2x + 3 \ (mod \ 97)$ and the point $P = (3, 6)$. Once we calculate all the multiples of P we will notice that there are just five distincts points $(0, P, 2P, 3P.4P)$ and they are repeating cyclically.



It's easy to spot the similarity between Scalar Multiplication on elliptic curves and Addition in modular arithmetic.

$$0P = 0$$
$$1P = (3, 6)$$
$$2P = (80, 10)$$
$$3P = (80, 87)$$
$$4P = (3, 91)$$

$$5P = 0$$
$$6P = (3, 6)$$
$$7P = (80, 10)$$
$$8P = (80, 87)$$
$$9P = (3, 91)$$

Notice that the multiples of P are just five (the rest points of the elliptic curve never appear) and that they are cyclically repeating themselves. We can sum up the previous multiples by:

$$5kP = 0$$
$$(5k+1)P = P$$
$$(5k+2)P = 2P$$
$$(5k+3)P = 3P$$
$$(5k+4)P = 4P$$

For every integer k. Finally, these five equations can be aggregated into a single one thanks to the modulo operator:

$$kP = (k mod 5)P$$

Alongside we can verify that these five points are closed under addition - which means; No matter how we add $0, P, 2P, 3P \ or \ 4P$, the result will always be one of these five points. Same goes with any other point, not just the $P = (3, 6)$ one.

If we take a generic P then:

$$\underbrace{nP + mP = P + \ldots + P}_{N \text{ times}} + \underbrace{P + \ldots + P}_{M \text{ times}} = (n + m)\,P$$

Which means, if we add two multiples of P, we obtain a multiple of P as well. This is enough to prove that **the set of the multiples of P is a cyclic subgroup** of the group formed by the elliptic curve.

A Subgroup is a group which is a *subset* of another group. A Cyclic Subgroup is a subgroup which elements are repeating *cyclically*, like we have shown in the above example. The point P is called **generator** or **base point** of the cyclic subgroup.

Cyclic subgroups are the foundations of ECC and other Cryptosystems

Subgroup order

Have you wondered so far what the order of a subgroup generated by a point P is (or in other words, what the order of P is)? To answer this question we cannot use the Schoof's algorithm as that only works on *whole elliptic curves* and not on subgroups.

So far we have defined the **order** as the ***number of points of a group***. When within a cyclic subgroup, we can accordingly say that the order of P is the ***smallest positive integer*** n**, such that** $nP = 0$**.**

The order of P is linked to the order of the elliptic curve by *Lagrange's theorem*, which states that *the* ***order*** *of a* ***subgroup*** *is a* ***divisor of the order of the parent*** *group*. In other words, if an elliptic curve contains N points and one of its subgroups contains n points, then n is a *divisor* of N.

The last two points, give us a way to find out the order of a subgroup with a base point of P

I. Calculate the elliptic curve's order N using Schoof's algorithm.
II. Find out all the divisors of N.
III. For every divisor n of N, compute nP.
IV. The smallest n such that $nP = 0$ is the order of the subgroup.

1st example

The curve $y^2 = x^3 - x + 3$ over the field F_{37} has order $n = 1, 2, 3, 6, 7, 14, 21 \; or \; 42.$ If we try out $P = (2, 3)$, we can see that $P \neq 0, \; 2P \neq 0, ..., \; 7P = 0$ hence the order of P is $n = 7.$

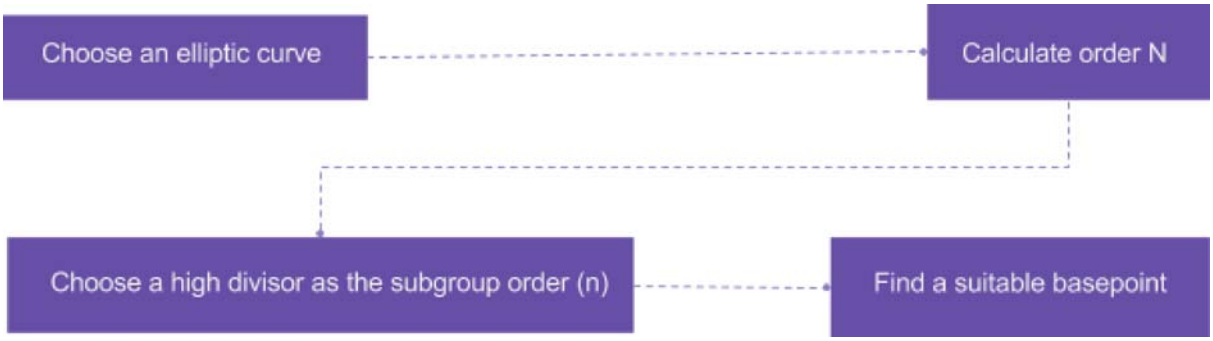
> Note that it is important to take the smallest divisor possible, not just a random one - because if we proceeded randomly, we could have taken n=14, which is not the order of the subgroup, but one of its multiples.

2nd example

The elliptic curve $y^2 = x^3 - x + 1$ over the field F_{29} has order $N = 37$, which is a **prime.** Its subgroups may only have order $n = 1 \; or \; 37$**.** When $n = 1$, the subgroup contains only the point at ***infinity***; when $n = N$, the subgroup contains **all the points of the elliptic curve.**

Finding a base point

For ECC algorithms we need subgroups with a high order. Our methodology is going to be like:



First, we need to introduce one more term. Lagrange's theorem implies that the number $h = N/n$ is always an integer (because n is a divisor of N). The number h has a name: it's the ***cofactor*** of the subgroup.

Consider that for every point of an elliptic curve we have $NP = 0$. This happens because N is a multiple of any candidate n. Using the definition of cofactor we can state:

$$n(hP) = 0$$

Now suppose that n is a prime number. This equation, written in this form, is telling us that the point $G = hP$ generates a subgroup of order n (except when $G = hP = 0$, in which case the subgroup has order 1).

In the light of this, we can not outline the following algorithm:

1. Calculate the order N of the elliptic curve
2. Choose the order n of the subgroup. For the algorithm to work, this number must be prime and must be a divisor of N
3. Compute the cofactor of $h = N/n$
4. Choose a random point P on the curve
5. Compute $G = hP$
6. If $G = 0$ then go back to step 4, otherwise we have found a generator of a subgroup with order n and cofactor h.

**The algorithm works only if n is a prime number. If n isn't a prime, then the order of G could be one of the divisors of n.*

Discrete logarithm

As we did when working with continuous elliptic curves, we are now going to discuss the question: if we know P and Q, what is k such that $Q = kP$?

This problem, which is known as the discrete logarithm problem for elliptic curves, is believed to be a "hard" problem, in that there is no known polynomial time algorithm that can run on a classical computer. There are, however, no mathematical proofs for this belief.

This problem is also analogous to the discrete logarithm problem used with other cryptosystems such as the **Digital Signature Algorithm** (DSA), the **Diffie-Hellman** key exchange (D-H) and the **ElGamal algorithm**. The difference is that, with those algorithms, we use *modulo exponentiation* instead of *scalar multiplication*. Their discrete logarithm problem can be stated as follows:

If we know $a,\ b$, what's k such that $b = a^k \ mod p$?

Both these problems are "**discrete**" because they involve finite sets (more precisely, ***cyclic subgroups***). And they are "logarithms" because they are analogous to ordinary logarithms.

What makes ECC interesting is that, as of today, the discrete logarithm problem for elliptic curves seems to be "harder" if compared to other similar problems used in cryptography. This implies that we need fewer bits for the integer k in order to achieve the same level of security as with other cryptosystems, as we will see in detail in the pages to come.

Chapter 3

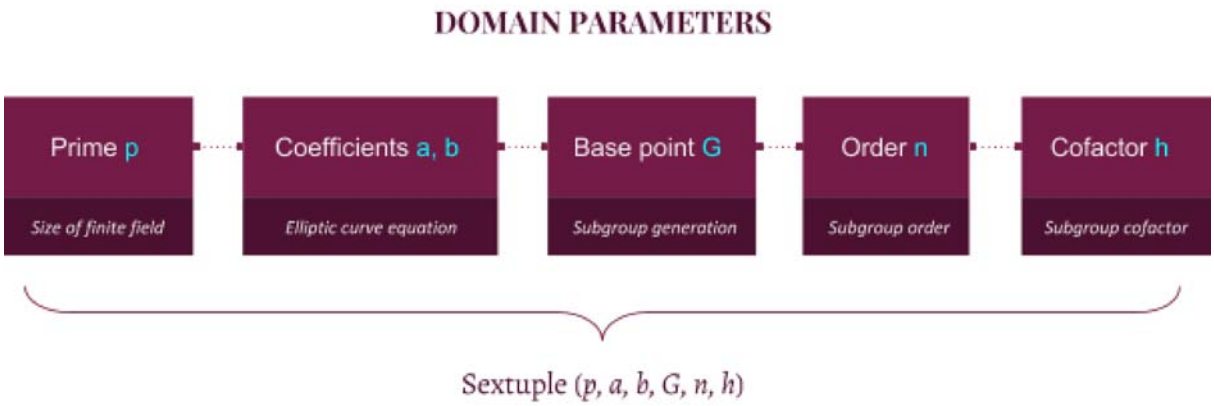
In the previous chapter, we have discussed what an elliptic curve is and we defined a group law in order to do some math with the points of elliptic curves. Once we restricted elliptic curves to finite fields of integers modulo a prime, we saw that the points of elliptic curves generate cyclic subgroups - at that point we introduced the terms; base point, cofactor and order.

We also saw that Scalar Multiplication in finite fields is an *easy* problem, while the Discrete Logarithm problem seems to be *hard*. In this chapter we are eventually going to see how all this we've talked about applies to cryptography.

Domain parameters

As long as our elliptic curve algorithms will work in a cyclic subgroup of an elliptic curve over a finite field, they will need the following parameters:

- The prime p that specifies the size of the finite field.
- The coefficients a, b of the elliptic curve equation.
- The base point G that generates our subgroup.
- The order n of the subgroup
- The cofactor h of the subgroup

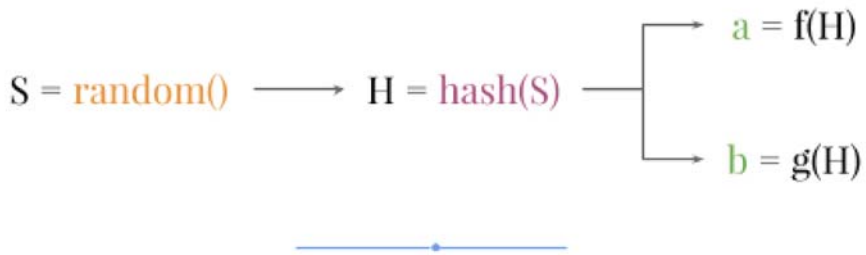


Random curves

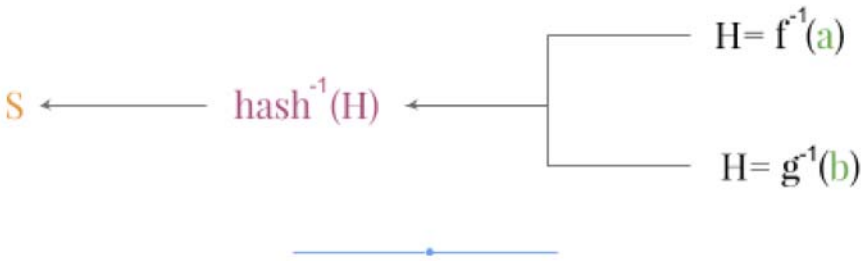
Don't take it for granted when we say that the Discrete Logarithm problem is "hard". There are some classes of elliptic curves that are particularly weak and allow the use of special purpose algorithms to solve the Discrete Logarithm problem efficiently. For instance, all the curves that have $p = hn$ (means that the order of the finite field equals the order of the elliptic curve) is vulnerable to Smart's attack, which can be used to solve discrete logarithms in polynomial time on a classical computer.

Now suppose we give you the domain parameters of a curve. There is a possibility that we've discovered a new class of weak curves that nobody knows, and we probably have built a "fast" algorithm for computing discrete logarithms on the curve we gave to you. How could you be convinced that we are not aware of any vulnerability? How can we assure you that the curve is "safe" - in the sense that it can't be used for special purpose attacks by us?

To solve this kind of problem, we sometimes use an additional domain parameter: the seed S. This is a random number used to generate the coefficients a, b, or the base point G, or even both. These parameters are generated by computing the hash of the seed S. Hashes, as you may already know, are easy to compute but hard to reverse.



This is a sketch of how a random curve is generated from a seed (**S**): the hash of a random number is used to calculate different parameters of the curve.



If we wanted to cheat and try to construct a seed from the domain parameters, we would have to solve a hard problem: called as *hash inversion*.

A curve generated through a seed S is said to be verifiably random. The **principle** of using hashes to generate parameters is known as "**nothing up my sleeve**" and is commonly used in Cryptography.

This trick should give some sort of assurance that the curve has not been specially crafted to expose vulnerabilities known to the author. In fact, if someone gives you a curve along with a seed, it means they were not free to *arbitrarily* choose the parameters a and b, and you should be *relatively* sure that the curve cannot be used for special purpose attacks by them. The reason why we say *"relatively"* will be explained in the next chapter.

A standardized algorithm for generating and checking random curves is described in **ANSI X9.62** and is based on **SHA-1**. You can check the algorithms for generating verifiable random curves on a specification by SECG (URL: http://www.secg.org/sec1-v2.pdf - look for "Verifiably Random Curves and Base Point Generators").

Elliptic Curve Cryptography

We had a lot to talk about, but we after all we managed to reach the summit of our project. So, to put it simple:

- The private key is a random integer d chosen from $\{1, ..., n-1\}$ (where n is the order of the subgroup)
- The public key is the point $H = dG$ (where G is the base point of the subgroup)

You can see that if we know G and d along with the other domain parameters, finding H is quite easy. However, if H and G are known, finding the private key d is pretty hard, because it requires us to solve the discrete logarithm problem.

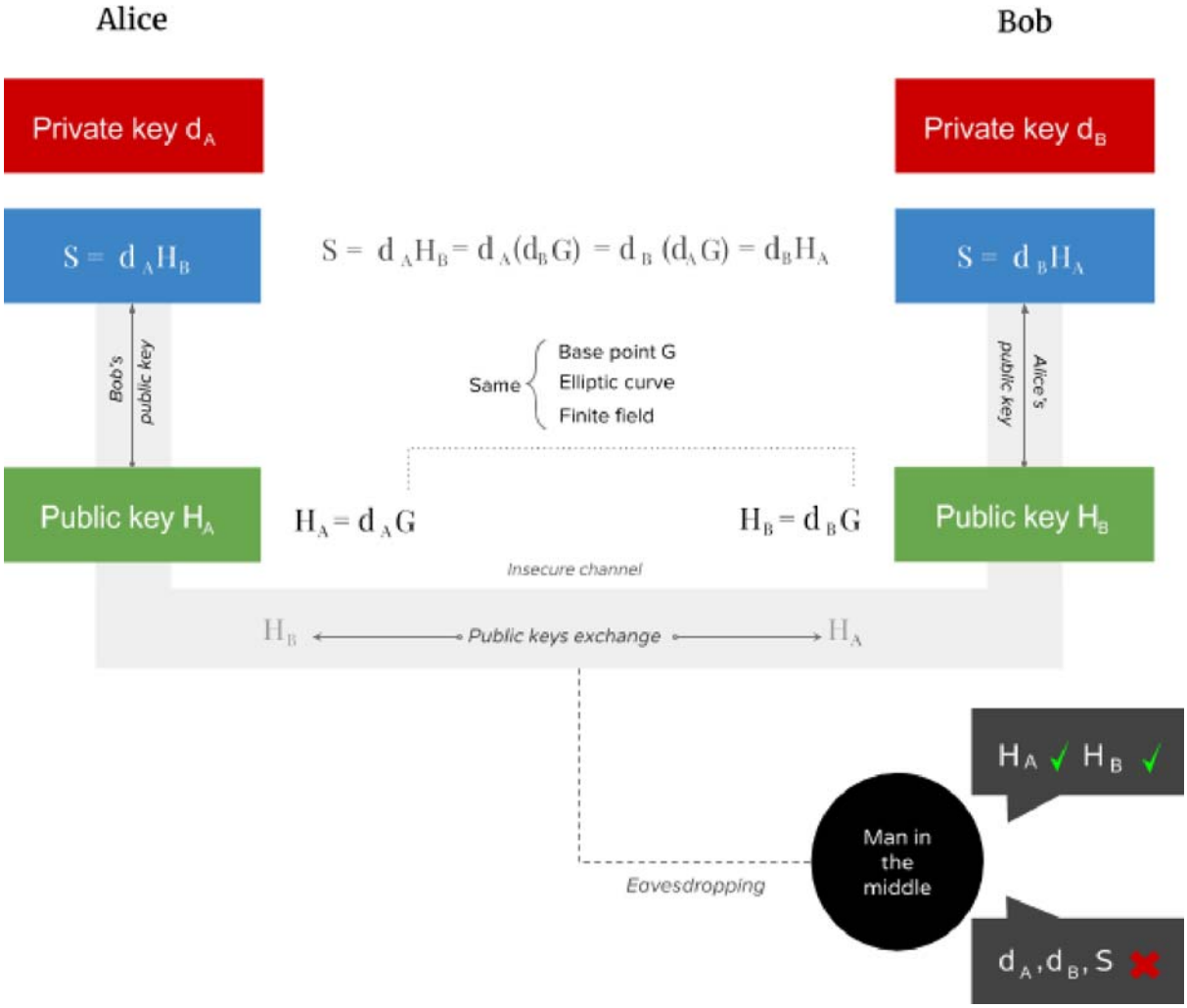
ECDH (Elliptic Curve Diffie-Hellman) and ECDSA (Elliptic Curve Digital Signature Algorithm) are two public-key algorithms based on the above strategy. They are used for encryption and digital signing respectively.

Encryption with ECDH

ECDH is a variant of the Diffie-Hellman algorithm for elliptic curves. It is actually a key-agreement protocol more than an encryption algorithm. This means that ECDH defines how keys should be generated and exchanged between parties. The way to encrypt data using such keys, is up to us.

Let's see an example with our usual pair Alice and Bob, so as to understand the use of the ECDH protocol.

Here is our schema:



1. First, Alice and Bob generate their own private and public keys. We have the private key d_A and the public key $H_A = d_A G$ for Alice and the keys d_B and $H_B = d_B G$ for Bob. Remember that both Alice and Bob use the same domain

parameters: the same base point G on the same elliptic curve on the same finite field.

2. Then Alice and Bob exchange their public keys H_A and H_B over an insecure channel. The man-in-the-middle intercepts H_A and H_B, but cannot find out neither d_A nor d_B without solving the discrete logarithm problem.
3. Alice calculates $S = d_A H_B$ (using her own private key and Bob's public key), and Bob calculates $S = d_B H_A$ (using his own private key and Alice's public key). The seed S is the same for both Alice and Bob.

$$S = d_A H_B = d_A(d_B G) = d_B(d_A G) = d_B H_A$$

The man-in-the-middle however can only know H_A and H_B along with the domain parameters and he still would not be able to find out the **shared secret** S. This is known as the Diffie-Hellman problem, which can be stated as follows:

"Given three points P, aP and bP, what is the result of abP?"

Or equivalently:

"Given three integers k, k^x and k^y, what is the result of k^{xy}?"

The latter form is originally used in the Diffie-Hellman algorithm, based on modular arithmetic.

The Diffie-Hellman problem for elliptic curves is assumed to be a hard problem. It is believed to be **as hard as the Discrete Logarithm problem**, although no mathematical proofs are available. What we can tell for sure is that it can't be *'harder'*, because solving the logarithm problem is a way of solving the Diffie-Hellman problem after all.

Now that Alice and Bob have obtained the **shared secret**, they can exchange data with symmetric encryption. For instance, they can use the x coordinate of S as the key to encrypt messages using secure ciphers like AES or 3-DES. This is more or less what TLS does, only the difference is that TLS concatenates the x coordinate with other numbers *relative* to the connection and then computes a **hash** of the *resulting byte string*.

You can see how ECDH computes private/public keys and shared secrets over an elliptic curve in this script we have developed in python for our project.

```
import collections
import random

EllipticCurve = collections.namedtuple('EllipticCurve', 'name p a
b g n h')
# here we define our elliptic curve

curve = EllipticCurve(
    'secp256k1', # curve type
    # Field characteristic.

p=0xfffffffffffffffffffffffffffffffffffffffffffffffffffffffefffffc
2f,
    # Curve coefficients.
    a=0,
    b=7,
    # Define the base point.

g=(0x79be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81
798,
0x483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8
),
    # Subgroup order.

n=0xfffffffffffffffffffffffffffffffebaaedce6af48a03bbfd25e8cd03641
41,
    # Subgroup cofactor.
    h=1,
)


# Modular arithmetics application

def inverse_mod(k, p):
    # Returns the inverse of k mod p
    # returns x such that (x * k) % p == 1
    # k <> 0 and p = prime
    if k == 0:
```

```
        raise ZeroDivisionError('division by zero') # zero
division error!

    if k < 0:
        # k ** -1 = p - (-k) ** -1  (mod p)
        return p - inverse_mod(-k, p) # return the inverse

    # Extended Euclidean algorithm
    s, old_s = 0, 1
    t, old_t = 1, 0
    r, old_r = p, k

    while r != 0:
        quotient = old_r // r
        old_r, r = r, old_r - quotient * r
        old_s, s = s, old_s - quotient * s
        old_t, t = t, old_t - quotient * t

    gcd, x, y = old_r, old_s, old_t

    assert gcd == 1
    assert (k * x) % p == 1

    return x % p


# Functions applied on curve points

def is_on_curve(point):
    # Returns True if the given point lies on the elliptic curve
    if point is None:
        # None represents the point at infinity.
        return True

    x, y = point

    return (y * y - x * x * x - curve.a * x - curve.b) % curve.p
== 0


def point_neg(point):
```

```
    # Returns -point
    assert is_on_curve(point)

    if point is None:
        # -0 = 0
        return None

    x, y = point
    result = (x, -y % curve.p)

    assert is_on_curve(result)

    return result


def point_add(point1, point2):
    # Returns the result of (P1 + P2) according to the group law
    assert is_on_curve(point1)
    assert is_on_curve(point2)

    if point1 is None:
        # 0 + P2 = P2
        return point2
    if point2 is None:
        # P1 + 0 = P1
        return point1

    x1, y1 = point1
    x2, y2 = point2

    if x1 == x2 and y1 != y2:
        # P1 + (-P2) = 0
        return None

    if x1 == x2:
        # if (P1 == P2)
        m = (3 * x1 * x1 + curve.a) * inverse_mod(2 * y1, curve.p)
    else:
        # if (P1 != P2).
        m = (y1 - y2) * inverse_mod(x1 - x2, curve.p)
```

```
    x3 = m * m - x1 - x2
    y3 = y1 + m * (x3 - x1)
    result = (x3 % curve.p,
              -y3 % curve.p)

    assert is_on_curve(result)

    return result


def scalar_mult(k, point):
    # Returns (k * P) computed using the double-and-add algorithm
    assert is_on_curve(point)

    if k % curve.n == 0 or point is None:
        return None

    if k < 0:
        # k * P = -k * (-P)
        return scalar_mult(-k, point_neg(point))

    result = None
    addend = point

    while k:
        if k & 1:
            # Add
            result = point_add(result, addend)

        # Double
        addend = point_add(addend, addend)

        k >>= 1

    assert is_on_curve(result)

    return result


# Keypair generation and ECDHE
```

```
def make_keypair():
    # Generate random private-public key pair
    private_key = random.randrange(1, curve.n)
    public_key = scalar_mult(private_key, curve.g)

    return private_key, public_key


print('Curve:', curve.name)

# Alice generates her own key-pair.
alice_private_key, alice_public_key = make_keypair()
print("Alice's private key:", hex(alice_private_key))
print("Alice's public key: (0x{:x},
0x{:x})".format(*alice_public_key))

# Bob generates his own key-pair.
bob_private_key, bob_public_key = make_keypair()
print("Bob's private key:", hex(bob_private_key))
print("Bob's public key: (0x{:x},
0x{:x})".format(*bob_public_key))

# Alice and Bob now exchange their public keys and verify the
shared secret
s1 = scalar_mult(alice_private_key, bob_public_key)
s2 = scalar_mult(bob_private_key, alice_public_key)
assert s1 == s2

print('Shared secret: (0x{:x}, 0x{:x})'.format(*s1))
```

The curve used in this script is $secp256k1$ from SECG - which stands for “Standards for Efficient Cryptography Groups”, founded by Certicom. The same curve is also used by **Bitcoin** for ***digital signatures***.

The domain parameters of the curve are:

p = 0xffffffff ffffffff ffffffff ffffffff ffffffff ffffffff fffffffe fffffc2f
a =0
b =7
x_G = 0x79be667e f9dcbbac 55a06295 ce870b07 029bfcdb 2dce28d9 59f2815b 16f81798
y_G = 0x483ada77 26a3c465 5da4fbfc 0e1108a8 fd17b448 a6855419 9c47d08f fb10d4b8
n = 0xffffffff ffffffff ffffffff fffffffe baaedce6 af48a03b bfd25e8c d0364141
h = 1

**numbers are taken from the OpenSSL Source Code*

The script is simple and includes some of the algorithms we have described so far:

- Point addition
- Double and add
- ECDH

Once you run the script, the output will look like:

```
Curve: secp256k1
Alice's private key:
0xe32868331fa8ef0138de0de85478346aec5e3912b6029ae71691c384237a3eeb
Alice's public key:
(0x86b1aa5120f079594348c67647679e7ac4c365b2c01330db782b0ba611c1d677,
0x5f4376a23eed633657a90f385ba21068ed7e29859a7fab09e953cc5b3e89beba)
Bob's private key:
0xcef147652aa90162e1fff9cf07f2605ea05529ca215a04350a98ecc24aa34342
Bob's public key:
(0x4034127647bb7fdab7f1526c7d10be8b28174e2bba35b06ffd8a26fc2c20134a,
0x9e773199edc1ea792b150270ea3317689286c9fe239dd5b9c5cfd9e81b4b632)
Shared secret:
(0x3e2ffbc3aa8a2836c1689e55cd169ba638b58a3a18803fcf7de153525b28c3cd,
0x43ca148c92af58ebdb525542488a4fe6397809200fe8c61b41a105449507083)
```

Ephemeral ECDH

The “E” in ECDHE stands for “Ephemeral” which refers to the fact that the keys exchanged are **temporary** rather than static.

ECDHE is used for example in TLS, where both parties (the client and the server) generate their public-private key pair on the fly, once the connection has been established. They keys are then signed with the TLS certificate - for authentication - and finally exchanged between the parties.

Signing with ECDSA

What if Alice wants to sign a message with her private key d_A and Bob wants to **validate** the signature using Alice’s public key H_A - nobody but Alice should be able to produce valid signatures. Everyone should be able to check signatures.

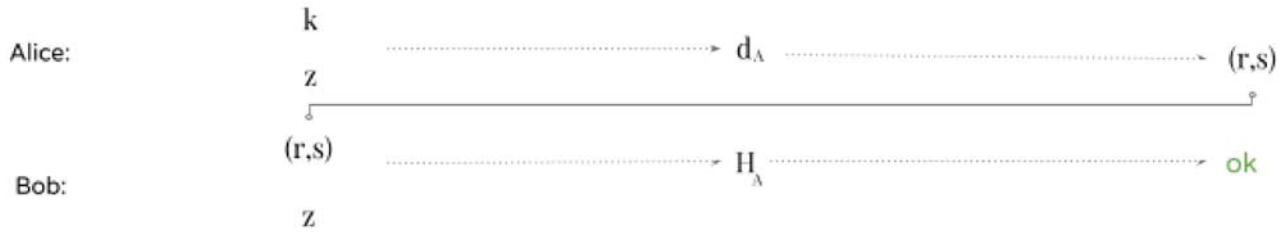
As we noted before, Alice and Bob are using the same domain parameters. The algorithm we are going to examine is ECDSA, a variant of the Digital Signature Algorithm applies to Elliptic Curves.

ECDSA works on the has of the message rather than on the message itself. The choice of the hash function is up to us, but it should be obvious that a cryptographically secure hash function should be chosen. The has of the message should be truncated so that the bit length of the hash is the same as the bit length of n (subgroup order). The truncated hash is an integer and will be denoted as z.

The algorithm when performed by Alice (to sign her message) goes as:

1. Take a random integer k chosen from $\{1, ..., n-1\}$ (where n is still the subgroup order).
2. Calculate the point $P = kG$ (where G is the base point of the subgroup).
3. Calculate the number $r = x_P \, mod n$ (where x_P is the x coordinate of P).
4. If $r = 0$ then choose another k and try again.
5. Calculate $s = k^{-1}(z + rd_A) \, mod n$ (where d_A is Alice’s private key and k^{-1} is the multiplicative inverse of $k \, modulo \, n$).
6. If $s = 0$ then choose another k and try again.

The pair (r, s) is the signature.



Alice signs the hash z using her private key d_A and a random k. Bob verifies that the message has been correctly signed using Alice's public key H_A.

More simply, this algorithm first generates a secret k. This secret is hidden in r thanks to **point multiplication -** r is then bound to the message hash by the equation:

$$s = k^{-1}(z + rd_A) \; mod n \; .$$

In order to calculate s, we have computed the inverse of $k \; modulo \; n$. As you might remember, this is guaranteed to work only if n is a **prime** number. If a subgroup has a non-prime order, ECDSA can't be used. It's not by chance that almost all standardized curves have a prime order, and those that have a non-prime order are **unsuitable** for ECDSA.

Verifying signatures

In order to verify the signature we need Alice's public key H_A, the truncated hash z and the signature (r, s).

1. Calculate the integer $u_1 = s^{-1} z \; mod n$
2. Calculate the integer $u_2 = s^{-1} r \; mod n$
3. Calculate the point $P = u_1 G + u_2 H_A$

The signature is valid only is $r = x_P \, mod n$

Correctness of the algorithm

The logic behind this algorithm may not seem obvious at first sight, however if we put together all the equation we have written so far, it's going to make more of a sense.

Let's start by the point $P = u_1 G + u_2 H_A$. From the definition of the public key we have that $H_A = d_A G$ (where d_A is the private key). We can write:

$$\begin{aligned} P &= u_1 G + u_2 H_A \\ &= u_1 G + u_2 d_A G \\ &= (u_1 + u_2 d_A) G \end{aligned}$$

Using the definitions of u_1 and u_2, we can write:

$$\begin{aligned} P &= (u_1 + u_2 d_A) G \\ &= (s^{-1} z + s^{-1} r d_A) G \\ &= s^{-1} (z + r d_A) G \end{aligned}$$

**Here we have omitted $"modn"$ for brevity and because the cyclic subgroup generated by G has order n, hence $"modn"$ is superfluous.*

Previously, we defined:

$$s = k^{-1} (z + r d_A) \ modn$$

If we multiply each side of the equation by k and divide by s, we get:

$$k = s^{-1} (z + r d_A) \ modn$$

Substituting this result in our equation for P, we get:

$$P = s^{-1} (z + r d_A) G = kG$$

Note that this is the same equation for P we had at step 2 of the signature generation algorithm. When generating signatures and verifying them, we're calculating the same point P (just with a different set of equations). This is why the algorithm works.

Playing with ECDSA

You can see this python script for **signature generation** and **verification.** The code share parts with the ECDH script (the domain parameters and the public/private key pair generation algorithm).

The output looks as follows:

```
Curve: secp256k1
Private key: 0x9f4c9eb899bd86e0e83ecca659602a15b2edb648e2ae4ee4a256b17bb29a1a1e
Public key: (0xabd9791437093d377ca25ea974ddc099eafa3d97c7250d2ea32af6a1556f92a,
0x3fe60f6150b6d87ae8d64b78199b13f26977407c801f233288c97ddc4acca326)

Message: b'Hello!'
Signature: (0xddcb8b5abfe46902f2ac54ab9cd5cf205e359c03fdf66ead1130826f79d45478,
0x551a5b2cd8465db43254df998ba577cb28e1ee73c5530430395e4fba96610151)
Verification: signature matches

Message: b'Hi there!'
Verification: invalid signature

Message: b'Hello!'
Public key: (0xc40572bb38dec72b82b3efb1efc8552588b8774149a32e546fb703021cf3b78a,
0x8c6e5c5a9c1ea4cad778072fe955ed1c6a2a92f516f02cab57e0ba7d0765f8bb)
Verification: invalid signature
```

As you can see, the script first signs a message (the byte string "Hello!"), then verifies the signature. Afterwards, it tries to verify the same signature against another message ("Hi there!") and verification fails. Lastly, it tries to verify the signature against the correct message, but using another random public key and verification fails again.

The importance of *k*

When generating ECDSA signatures, it is important to keep the secret k really ***secret***. If we used the same k for all our signatures, or if our random number generator was predictable, an attacker would be able to find out the private key itself.

In particular, Sony has made a similar mistake a few years ago. The PlayStation 3 game console can only run games signed by Sony with ECDSA. This way, if one wanted to create a new game for PlayStation 3, they couldn't distribute it to the

public without a signature from Sony. The problem was that all the signatures made by Sony were generated using a static k.

We could easily recover Sony's private key d_S by buying just two signed games, extracting their hashes z_1, z_2 and their signatures (r_1, s_1) and (r_2, s_2) along with the domain parameters.

1. First off, note that $r_1 = r_2$ - because $r = x_P \, mod n$ and $P = kG$ is the same for both signatures.
2. Consider that $(s_1 - s_2) \, mod \, n = k^{-1}(z_1 - z_2) \, mod \, n$
   *this result comes from the equation for s.
3. Now multiply each side of the equation by k: $k(s_1 - s_2) \, mod \, n = (z_1 - z_2) \, mod \, n$.
4. Divide by $(s_1 - s_2)$ to get $k = (z_1 - z_2)(s_1 - s_2)^{-1} \, mod \, n$.

The last equation lets us calculate k using only two hashes and their corresponding signatures. Now we can extract the private key using the equation for s:

$$s = k^{-1}(z + rd_S) \, mod \, n \;\Rightarrow d_S = r^{-1}(sk - z) \, mod \, n$$

Voila! Similar techniques may be employed if k is *predictable* in some way as well.

Chapter 4

So far we have seen the ECDH and ECDSA algorithms and how the Discrete Logarithm problem for elliptic curves plays an important role for their security. In this section we will try to get a brief idea of how "hard" is the complexity of the logarithm problem with today's techniques. Finally, we will try to answer the question: Why do we need Elliptic Curve Cryptography if RSA and other cryptosystems based on modular arithmetic work well?

Breaking the Discrete Logarithm problem

The two most efficient algorithms for computing Discrete Logarithms on elliptic curves, are the:

- ❏ Baby-step, giant-step algorithm
- ❏ Pollard's rho method

Before we continue, let us remind you what the Discrete Logarithm problem is about: *"Given two points* P *and* Q*, find out the integer* x *that satisfies the equation* $Q = xP$*"* The points are part of the subgroup of an elliptic curve, which has a base point G and which order is n.

Baby-step, giant-step

Keep in mind that we can always write any integer x as $x = am + b$, where $a,\ m$ and b are three arbitrary integers. For example the number ten can be written as:

$$10 = 2 \cdot 3 + 4$$

So now we can rewrite the Discrete Logarithm problem equation as follows:

$$Q = xP$$
$$Q = (am + b)P$$
$$Q = amP + bP$$
$$Q - amP = bP$$

The "baby-step, giant-step" is a "meet in the middle" algorithm. Unlike to the Brute-force attack -which forces us to calculate all the points xP for every x until we

find Q, it calculates a few values for bP and few values for $Q - amP$ until it finds a correspondence.

The algorithm works as shown below:

1. Calculate $m = [\sqrt{n}]$
2. For every b in $0, ..., m$, calculate bP and store the results in a hash table.
3. For every a in $0, ..., m$:
    a. Calculate amP
    b. Calculate $Q - amP$
    c. Check the hash table and look if there is any point bP such that $Q - amP = bP$
    d. If such point exists, then we have found $x = am + b$

We initially calculate the points bP with little **'baby'** increments for the coefficient b $(1P, 2P, 3P, ...)$. Then in the second part of the algorithm, we calculate the points amP with huge **'giant'** increments for am $(1mP, 2mP, 3mP, ...$ where m is a huge number$)$.

$$Q = (3 + 6m)P$$
logarithm

To understand why this algorithm works, forget for a moment that the points bP are cached and take the equation $Q = amP + bP$. Consider what follows:

- When $a = 0$ we are checking whether Q is equal to bP, where b is one of the integers from 0 to m. This way, we are comparing Q against all points from $0P$ to mP.
- When $a = 1$ we are checking whether Q is equal to $mP + bP$. We are comparing Q against all points from mP to $2mP$.
- When $a = 2$ we are comparing Q against all the points from $2mP$ to $3mP$.
- …

- When $a = m - 1$, we are comparing Q against all points from $(m-1)mP$ to $m^2P = nP$

In conclusion, we are checking all points from $0P$ to nP (all the possible points) performing at most $2m$ additions and multiplications (exactly m for the baby-steps, at most m for the giant-steps).

If you consider that a lookup on a hash table takes $O(1)$ time, it's easy to see that this algorithm has both time and space complexity $O(\sqrt{n})$ (or $O(2^{k/2})$ if you consider the bit length). It's still exponential time, but much better than a brute-force attack.

Baby-step giant-step in practice

It may help you understand if you see what the complexity $O(\sqrt{n})$ means in practice. Let's take a standardized curve $prime192v1$ $(aka\ secp192r1,\ ansiX9p192r1)$. This curve has an order $n = 0xffffffff\ ffffffff\ ffffffff\ 99def836\ 146bc9b1\ b4d22831$. The square root of n is approximately $7.922816251426434 \cdot 10^{28}$ (almost eighty octillions).

Now imagine storing $\sqrt{n}$ points in a hash table. Suppose that each point requires exactly 32 bytes: our hash table would need approximately $2.5 \cdot 10^{30}\ bytes$ of memory. The total world storage capacity is in the order of the zettabyte $(10^2\ bytes)$. This is almost ten orders of magnitude lower than the memory required by our hash table. Even if our points took $1\ byte$ each, we would still be very far from being able to store all of them.

This is impressive, and becomes even more impressive if you consider that $prime192v1$ is one of the curves with lowest order. The order of $secp521r1$ (another standard curve from NIST) is approximately $6.9 \cdot 10^{156}$.

Pollard's ρ

Pollard's rho is another algorithm for computing discrete logarithms. It has the same asymptotic time complexity $O(\sqrt{n})$ of the baby-step giant-step algorithm, but its space complexity is just $O(1)$.

First of all, another reminder of the Discrete Logarithm problem: given P and Q find x such that $Q = xP$. With Pollard's rho, we will solve a slightly different problem: **given P and Q, find the integers $a,\ b,\ A\ and\ B$ such that $aP + bQ = AP + BQ$.**

Once the four integers are found, we can use the equation $Q = xP$ to find out x:

$$
\begin{aligned}
aP + bQ &= AP + BQ \\
aP + bxP &= AP + BxP \\
(a + bx)P &= (A + Bx)P \\
(a - A)P &= (B - b)xP
\end{aligned}
$$

Now we can get rid of P. Before we do so though, remember that our subgroup is cyclic with order n, therefore the coefficients used in point multiplication are modulo n.

$$
\begin{aligned}
a - A &\equiv (B - b)x\ (mod\ n) \\
x &= (a - A)(B - b)^{-1}\ mod\ n
\end{aligned}
$$

The principle of operation of Pollard's rho is simple: we define a **pseudo-random sequence of $(a,\ b)$ pairs**. This sequence of pairs can be used to generate the sequence of points $aP + bQ$. Because both P and Q are elements of the same cyclic subgroup, the **sequence of points $aP + bQ$ is cyclic too**.

This means that sooner or later we will detect a cycle in our pseudo-random sequence $(a,\ b)$ pairs. That is: **we will find a pair $(a,\ b)$ and another distinct pair $(A,\ B)$ such that $aP + bQ = AP + BQ$.** Same points, distinct pairs: we can apply the equation above to find the logarithm.

The problem is: *How do we detect the cycle in an efficient way?*

Tortoise and Hare

In order to detect our cycle, we could try all the possible values for a and b using a pairing function, but given that there are n^2 such pairs, our algorithm would be $O(n^2)$, much worse than a brute-force attack.

There is a faster method though: the **tortoise and hare algorithm** (aka Floyd's cycle-finding algorithm). The picture below shows the principle of operation of the tortoise and hare method, which is at the core of Pollard's rho.

We have the curve $y^2 \equiv x^3 + 2x + 3 \ (mod\ 97)$ and the points $P = (3,\ 6)$ and $Q = (80,\ 87)$. The points belong to a cyclic subgroup of order 5. We walk a sequence of pairs at different speeds until we find two different pairs $(a,\ b)$ and $(A,\ B)$ that produce the same point. In this case, we have found the pairs $(3,\ 3)$ and $(2,\ 0)$ that allow us to calculate the logarithm as $x = (3-2)(0-3)^{-1}\ mod\ 5 = 3$. And in fact we correctly have $Q = 3P$.

Basically, we take our pseudo-random sequence of (a, b) pairs, together with the corresponding sequence of $aP + bQ$ points. The sequence of (a, b) pairs may or may not be cyclic, but the sequence of point is, because both P, Q were generated from the same base point, and from the properties of subgroups we know that we can't "escape" from the subgroup using just scalar multiplication and addition.

Now we take our two pets, the tortoise and the hare, and make them walk our sequence from left to right. The **tortoise** (the green spot in the picture) is **slow** and **reads each point one by one**; the **hare** (represented in red) is **fast** and **skips a point at every step**.

After some time both the tortoise and the hare will have found the same point, but with different coefficient pairs. Or, to express that with equations, the tortoise will have found a pair (a, b) and the hare will have found a pair $(A,\ B)$ such that $aP + bQ = AP + BQ$.

If our random sequence is defined through an algorithm (as opposed to being stored statically), it's easy to see how this principle of operation **requires just** $O(logn)$ **space**. Calculating the asymptotic time complexity is not that easy, but we can build

a probabilistic proof that shows how the **time complexity** is $O(\sqrt{n})$, as we have already said.

Pollard's ρ in practice

We said that baby-step giant-step can't be used in practice, because of the huge memory requirements. Pollard's rho, on the other hand, requires very few memory. So, how practical is it?

Certicom launched a challenge in 1998 to compute discrete logarithms on elliptic curves with bit lengths ranging from 109 to 359. As of today, only **109-bit long curves have been successfully broken**. The latest successful attempt was made in 2004. Quoting Wikipedia:

> *"The prize was awarded on 8 April 2004 to a group of about 2600 people represented by Chris Monico. They also used a version of a parallelized Pollard rho method, taking 17 months of calendar time."*

As we have already said, $prime192v1$ is one of the "smallest" elliptic curves. We also said that Pollard's rho has $O(\sqrt{n})$ time complexity. If we used the same technique as Chris Monico (the same algorithm, on the same hardware, with the same number of machines), how much would it take to compute a logarithm on $prime192v1$?

$$17\ months \times \frac{\sqrt{2^{192}}}{\sqrt{2^{109}}} \simeq 5 \cdot 10^{13}\ months$$

This number is pretty self-explanatory and gives a clear idea of how hard it can be to break a discrete logarithm using such techniques.

Pollard's ρ vs Baby-step giant-step

Here is a script that puts together the baby-step giant-step script, Pollard's script with a brute-force script into a final one to compare their performances.

This fourth script computes all the logarithms for all the points on the "tiny" curve using different algorithms and reports how much time it did take:

```
Curve order: 10331
Using bruteforce
Computing all logarithms: 100.00% done
Took 2m 31s (5193 steps on average)
Using babygiantstep
Computing all logarithms: 100.00% done
Took 0m 6s (152 steps on average)
Using pollardsrho
Computing all logarithms: 100.00% done
Took 0m 21s (138 steps on average)
```

As we could expect, the brute-force method is tremendously slow if compared to the others two. Baby-step giant-step is the faster, while Pollard's rho is more than three times slower than baby-step giant-step (although it uses far less memory and fewer number of steps on average).

Also look at the number of steps: brute force used 5193 steps on average for computing each logarithm. 5193 is very near to $10331 / 2$ (half the curve order). Baby-step giant-steps and Pollard's rho used 152 steps and 138 steps respectively, two numbers very close to the square root of 10331 (101.64).

Final consideration

While discussing these algorithms, we have presented many numbers. It's important to be cautious when reading them: algorithms can be greatly optimized in many ways. Hardware can improve. Specialized hardware can be built.

The fact that an approach today seems impractical, does not imply that the approach can't be improved. It also does not imply that other, better approaches exist (remember, once again, that we have no proofs for the complexity of the discrete logarithm problem).

Shor's algorithm

If today's techniques are unsuitable, what about tomorrow's techniques? Well, things are a bit more worrisome: there exist a quantum algorithm capable of computing discrete logarithms in polynomial time: Shor's algorithm, which has time complexity $O((log\ n)^3)$ and space complexity $O(log\ n)$.

This peculiarity implies that with a limited number of qubits, we can deal with lots of possible inputs at the same time. So, for example, we can tell a quantum computer that there's a number x uniformly distributed between 0 and $n-1$. This requires just $log\ n$ qubits instead of $nlog\ n$ bits. Then, we can tell the quantum computer to perform scalar multiplication xP. This will result in the superposition of states given by all the points from $0P$ to $(n-1)P$ - that is, if we measured our qubits now, we would obtain one of the points from $0P$ to $(n-1)P$ with probability $1/n$.

This was to give you an idea of how powerful state superposition is. Shor's algorithm does not work exactly this way, it is actually more complicated. What makes it complicated is that, while we can "simulate" n states at the same time, at some point we have to reduce these many states to just a few ones, because we want as output a single answer, not many (i.e. we want to know one single logarithm, not many probable wrong logarithms).

ECC and RSA

Now let's forget about quantum computing, which is still far from being a serious problem. The question I'll answer now is: ***why bother with elliptic curves if RSA works well***? A quick answer is given by NIST, which provides with a table that compares RSA and ECC key sizes required to achieve the same level of security:

RSA key size (bits)	ECC key size (bits)
1024	160
2048	224
3072	256
7680	384
15360	521

Note that there is no linear relationship between the **RSA** key sizes and the **ECC** key sizes (in other words: if we double the **RSA** key size, we don't have to double the **ECC** key size). This table tells us not only that **ECC** uses less memory, but also that key generation and signing are considerably faster.

But why is it so? The answer is that the faster algorithms for computing discrete logarithms over elliptic curves are **Pollard's rho** and **baby-step giant-step**, while in the case of **RSA** we have faster algorithms. One in particular is the general number field sieve: an algorithm for integer factorization that can be used to compute discrete logarithms. The general number field sieve is the fastest algorithm for integer factorization to date.

All of this applies to other cryptosystems based on modular arithmetic as well, including **DSA**, **D-H** and **ElGamal**.

Hidden threats of NSA

So far we have discussed algorithms and mathematics. Now it's time to discuss people, and things get more complicated.

If you remember, in the last chapter we said that certain classes of elliptic curves are weak, and to solve the problem of trusting curves from dubious sources we added a random **seed** to our **domain parameters**. And if we look at standard curves from NIST we can see that they are all verifiably random.

If we read the Wikipedia page for "nothing up my sleeve", we can see that:

- The random numbers for **MD5** come from the **sine** of integers.
- The random numbers for **Blowfish** come from the first digits of π.
- The random numbers for **RC5** come from both e and the golden ratio.

These numbers are random because their digits are uniformly distributed. And the are also unsuspicious, because they have a justification.

Now the question is: where do the random seeds for NIST curves come from? The answer is: we don't know. Those seeds have no justification at all.

Is it possible that NIST has discovered a "sufficiently large" class of weak elliptic curves and has tried many possible seeds until they found a vulnerable curve? We can't answer this question, but this is a legit and important question. We know that NIST has succeeded in standardizing at least a vulnerable **random number generator**

(a generator which, oddly enough, is based on elliptic curves). Perhaps they also succeeded in standardizing a set of weak elliptic curves. How do we know? We can't.

What's important to understand is that "**verifiably random**" and "**secure**" are not **synonyms**. And it doesn't matter how hard the logarithm problem is, or how long our keys are, if our algorithms are broken, there's nothing we can do.

With respect to this, RSA wins, as it does not require special domain parameters that can be tampered. RSA (as well as other modular arithmetic systems) may be a good alternative if we can't trust authorities and if we can't construct our own domain parameters. And in case you are asking: yes, TLS may use NIST curves.

Curves and fields

Koblitz curves over binary fields.

Those are elliptic curves in the form $y^2 + xy = x^3 + ax^2 + 1$ (where a is either 0 or 1) over finite fields containing 2^m elements (where m is a prime). They allow particularly efficient point additions and scalar multiplications. Examples of standardized Koblitz curves are $nistk163$, $nistk283$ and $nistk571$ (three curves defined over a field of 163, 283 and 571 bits).

Binary curves.

They are very similar to Koblitz curves and are in the form $x^2 + xy = x^3 + x^2 + b$ (where b is an integer often generated from a random seed). As the name suggests, binary curves are restricted to binary fields too. Examples of standardized curves are $nistb163,\ nistb283\ and\ nistb571.$ It must be said that there are growing concerns that both Koblitz and Binary curves may not be as safe as prime curves.

Edwards curves

in the form $x^2 + y^2 = 1 + dx^2y^2$ (where d is either 0 or 1). These are particularly interesting not only because point addition and scalar multiplication are fast, but also because the formula for point addition is always the same, in any case ($P \neq Q,\ P = Q,\ P = -Q,\ \ldots$). This feature leverages the possibility of side-channel attacks, where you measure the time used for scalar multiplication and try to guess the scalar coefficient based on the time it took to compute. Edwards curves are relatively new (they were presented in 2007) and no authority such as Certicom or NIST have yet standardized any of them.

Curve25519 and **Ed25519**

are two particular elliptic curves designed for ECDH and a variant of ECDSA respectively. Like Edwards curves, these two curves are fast and help preventing side-channel attacks. And like Edwards curves, these two curves have not been standardized yet and we can't find them in any popular software (except OpenSSH, that supports Ed25519 key pairs since 2014).

Thank you