Elliptic Curve Cryptography

Chloe Cheong

An Introduction to Public Key Cryptography:

For thousands of years, people have been using cryptography to conceal messages so that only they, and the intended recipient/s are able to understand and view their contents. Whilst cryptography has been used throughout history, the earliest evidence being found in Egypt in 1900 B.C, it was with the Second World War, the dawn of the 21st Century and the rise of the internet and online communication, that there was a renewed push for cryptographic protocols and a particular focus on allowing two parties to share information securely without any previous form of communication.

During World War II, fast and accurate communication of sensitive information was necessary between allies. When put into the hands of the opposing forces, this information could have drastic consequences and as such, complete secrecy was imperative. One of the technologies that emerged during this time was the Enigma Machine, whose trick was to scramble the letters entered into it in accordance with the configuration of its settings. With each keystroke, the settings changed and due to the extremely large number of settings, it was thought that its code was unbreakable. However, methods to decode the ciphertext were discovered in the 1940s and later published in the 1970s. The key thing to note about this cryptosystem, is that in order for someone to decode the text on the other end, they had to have their machine in the exact same configuration as the machine that encoded it. This meant that some information had to be shared beforehand between both parties before the actual transmission was conducted. This brings us to the two features that are always at odds in the field of cryptography: convenience and security. Whilst the Enigma Machine and its protocol were 'secure' in the sense that it was extremely difficult to break the codes as an outsider, it was inconvenient and difficult to share the required information, or what I will call the 'key', beforehand. This kind of cryptographic protocol is called Symmetric, meaning that a small amount of information has to be shared between sender and recipient before more information can be transmitted securely.

This brings into question whether it is possible to have a cryptosystem where no information is shared beforehand. Such systems would be ideal for web-based communication, as people would not have to know or meet each other before the transmission of the information took place. These systems are called Asymmetric and fall under Public Key Cryptography, an idea proposed in the 1970s by Merkle, Diffie and Hellman. It is mathematics that allows for these types of cryptosystems, as they are rooted in mathematical problems that we presume to be computationally difficult to solve. Characteristically, these systems involve both parties creating public and private keys, where the method for using these keys is made publicly known. This seems problematic as anyone may be able to encode a message using the public key, however, the 'trap-door' or one-way functions that these systems utilise make it exceptionally difficult for an outsider to decrypt a message. Often, both Symmetric and Asymmetric cryptosystems are combined to take advantage of each system's strengths, with asymmetric protocols being used to share the key for symmetric encryption.

RSA – An Explanation:

One of the first, most famous and widely used asymmetric cryptosystems is RSA (created by Rivest, Shamir and Adleman). It is important to note that no efficient algorithm has been made to break all cases of RSA, but the choice of certain parameters for encryption is necessary in prevention against particular methods of attack.

It is fairly computationally simple to multiply numbers together; however, it is significantly more difficult to factorise them. For example, if given two very large numbers, say 67643 and 99781, we can multiply both easily using a calculator and get: $6{,}749{,}486{,}183$.

Now, say you are given the number 2765797247. You are told it was made by multiplying two large primes, and that you ate to factorise it. This is a very difficult task, and in order to solve it, you would probably try dividing the number successively by every prime less than its square root: $\sqrt{2765797247} \approx 52591$. At this point you are aware that this will most likely take longer to do than it took the person who gave you the number to multiply its prime factors, 1147 and 1911401, beforehand.

This mathematical problem, known as the discrete factoring problem, becomes the basis for RSA. Given the product, N, of two large prime numbers p and q, it is very hard to factorise N and figure out what p and q are.

To set up this example, I propose that we have three characters: Alice, Bob and Eve (the eavesdropper). Say Bob wishes to be able to receive messages from people. He will pick two large primes, p and q and calculate N, their product, which will become part of his public key (the information he shares with everyone). Next, he will choose an exponent, e, in the range $1 < e < \alpha$, where $\alpha = (p-1)(q-1)$, and $gcd(e, \alpha) = 1$ such that e and α are relatively prime. e will also become part of his public key. Note that if Eve attempts to factor e, she will only be able to gather a small list of factors that α might not have.

At this point, Alice can now encrypt a message, m $(where\ 1 < m < N)$. She will do this by calculating $c = m^e \ (mod\ N)$ and sending the resulting ciphertext to Bob. In order to decode the message, Bob must calculate the inverse, $d = e^{-1}$. Essentially, this means that he must find a number d such that $ed = 1(mod\ (p-1)(q-1))$. This is relatively easy for him as he knows a. This problem can then be solved using the Euclidean Algorithm. Finally, to find the decoded message, Bob must calculate $c^d \ (mod\ N)$.

$$\begin{aligned} c^d &\equiv m^{ed}\ (mod\ N) \\ &\equiv m^{1+r\alpha}\ (mod\ N) \\ &\equiv m \cdot m^{r\alpha}\ (mod\ N) \\ &\equiv m \cdot (m^{\alpha})^r\ (mod\ N) \end{aligned}$$

Now, using the fact that $(m^{\alpha}) \equiv 1\ (mod\ N)$ (which can be proven by a combination of the Chinese Remainder Theorem and Fermat's little theorem), he is left with the original message.

The security of RSA primarily relies on the fact that Eve, the eavesdropper, will not easily be able to factor N. If she could, she would easily be able to calculate a and d, allowing her to decipher the cyphertext sent by Alice. Many people have tried other ways of getting around RSA encryption, specifically by trying to find the value of m through the fact that $c = m^e (mod\ N)$. However, no efficient method has been discovered to solve this problem. As long as N cannot be easily factored, RSA is considered a secure cryptosystem.

Introduction to Elliptic Curve Cryptography:

Elliptic Curve Cryptography (ECC) is a form of public key cryptography that utilises the mathematics behind elliptic curves. It was first introduced in 1985 by mathematicians Victor Miller and Neal Koblitz, who both came up with the idea for a new cryptographic algorithm independently. However, it was not until the early 2000s with the growth of the internet that governments and websites began to use this protocol.

Whilst RSA is still in use today in many modern computer systems, an increase in computational power and the generation of algorithms such as the Quadratic Sieve and the General Number Field Sieve have made factoring easier and pushed RSA towards using larger key sizes. Thus, ECC is becoming increasingly more popular due to its ability to provide the same security as RSA with a much smaller key-size.

The U.S National Institute of Standards and Technology (NIST) has endorsed ECC in its 'Suite B' set of recommended algorithms and it is currently approved by the NSA for protection of top-secret information using a 384-bit key. However, in 2015, the NSA revealed that there are plans to replace Suite B with new protocols due to concern over quantum computing attacks.

The Curves:

What is an elliptic curve? An elliptic curve can be defined over any field, for example the real numbers $\mathbb{R}$, and consists of a set of points that satisfy and equation of the form:

$$y^2 + axy + by = x^3 + cx^2 + dx + e$$

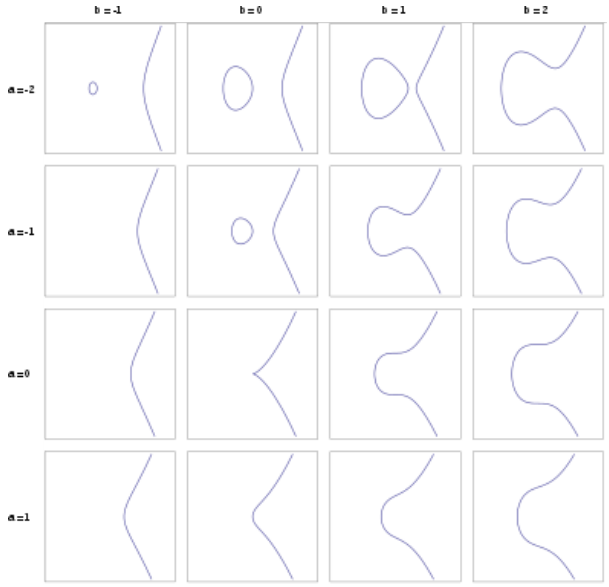
However, the standard representation for the elliptic curves we use for ECC is in Weierstrass form:

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

These curves are denoted by $E(a, b)$ where a and b are the constants in the equation above. Note that in order for a curve in this form to be used, it must be non-singular, i.e. it mustn't intersect with itself. To tell if a curve is singular or not, we compute its discriminant:

$$\Delta = -16 * (4a^3 + 27b^2)$$

If this discriminant is nonzero, the curve is non-singular. The graphs of elliptic curves look something like this:

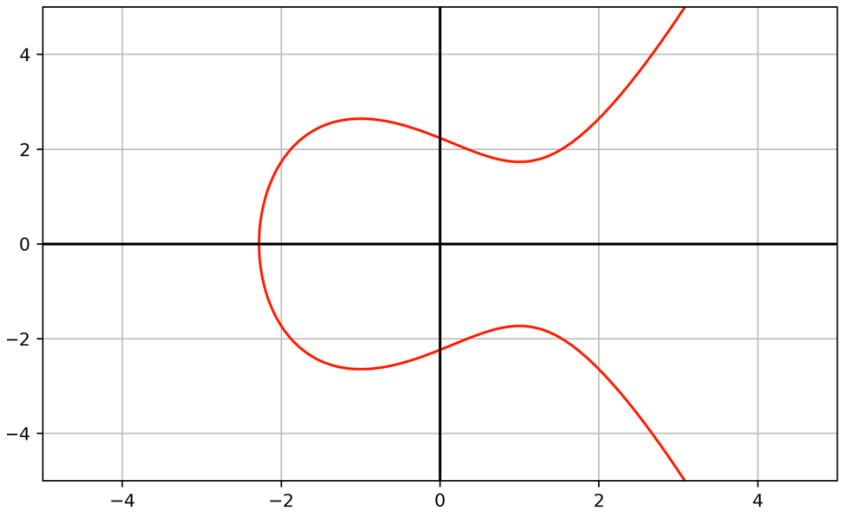


Reference: https://en.wikipedia.org/wiki/Elliptic_curve

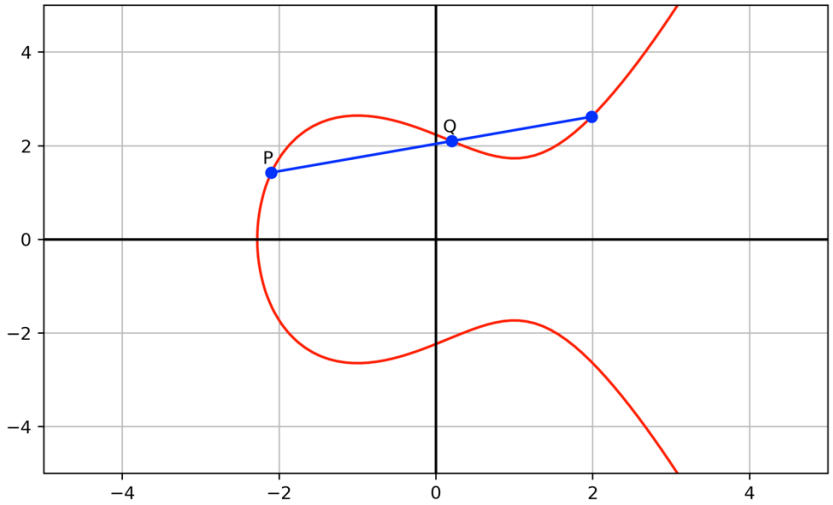
Note that these curves have symmetry about the x-axis, i.e. if I reflect a point on the curve horizontally it will still lie on the curve. Also note that if I draw a line through any two points on the curve, I will intersect the curve at most three times.

Let's play a Game:

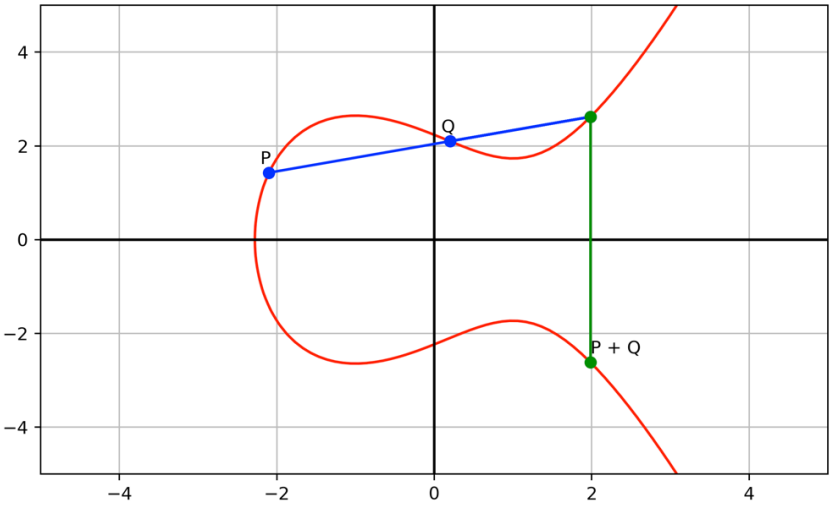
Let's pick a curve, say with $a = -3$ and $b = -5$.



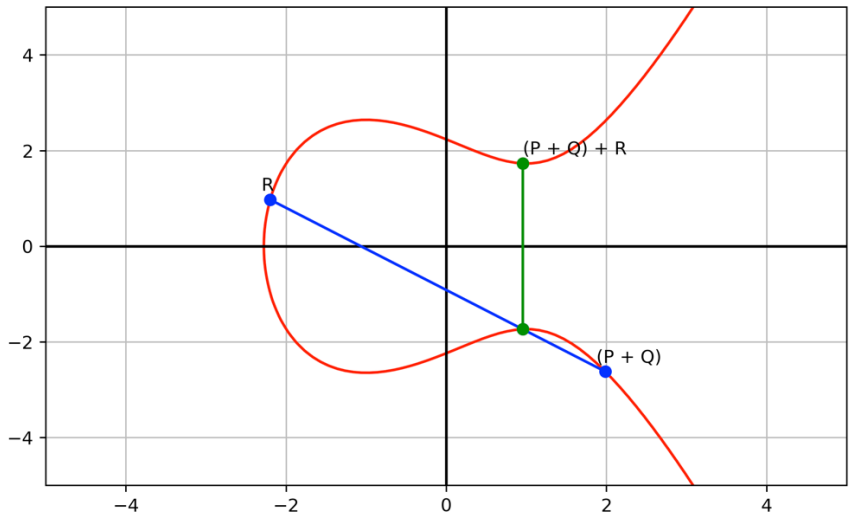
Pick two points on the curve. We will call them P and Q. Draw a line connecting them. This line will intersect the curve at most three times.



From the last point the line intersects the curve, draw a straight line downwards as follows. Let's call the point where this new line intersects the curve $P + Q$.

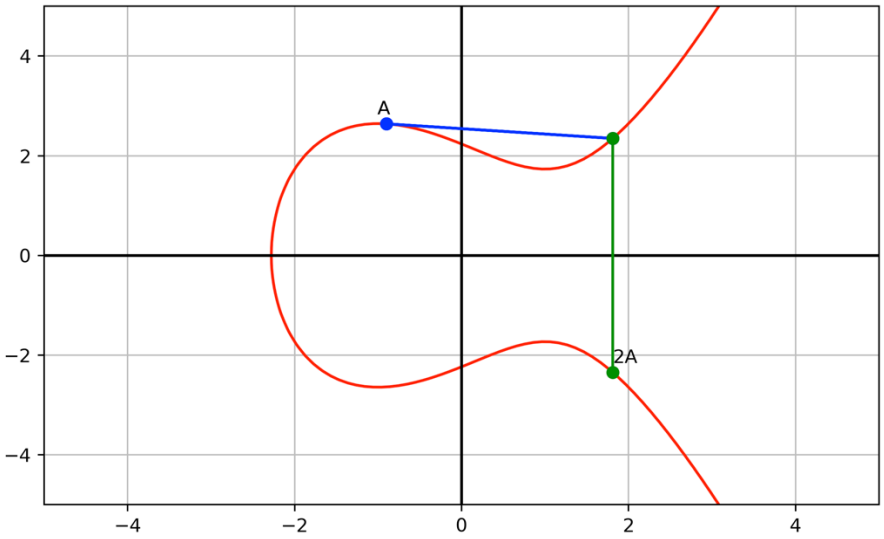


Say we repeat this trick again, but this time we draw our line from the point $P + Q$ to another point on the curve, R. We reflect the third point that this hits the curve horizontally, and voila we have our new point, $(P + Q) + R$.

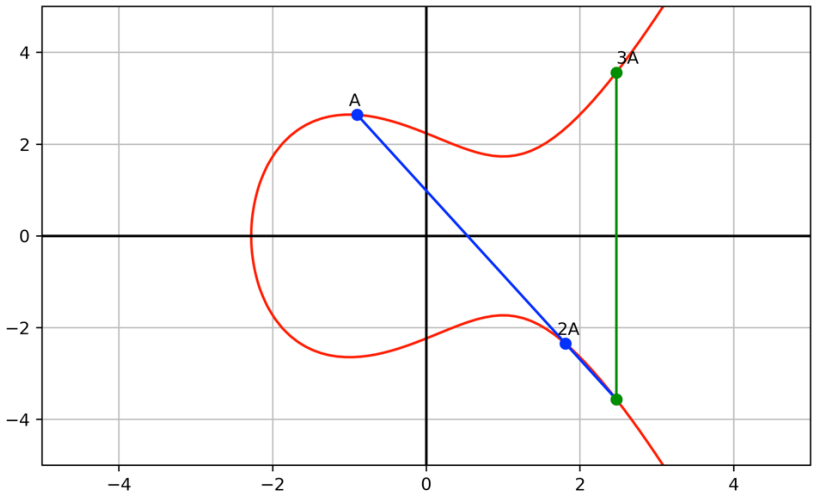


It's clear that we can just keep repeating this by picking a new point on the curve, drawing a line from the previous point through that new point, then drawing a straight line downwards from the new point where this line intersects the curve.

But what if we don't want to have to find a new point to draw our line through every time? Instead, let's start by drawing the tangent line from a point A, then reflecting where it hits the curve horizontally. We will call this $A + A$ or $2A$. This is intuitive because it is essentially the same as the previous example, but where P and Q are infinitely close.

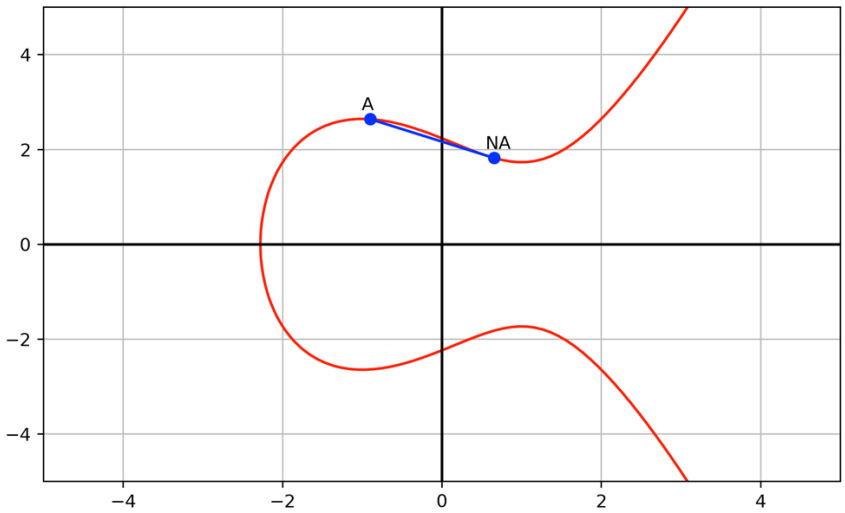


Next, we draw our line between A and $2A$ and perform the same trick again. The new point will now be $2A + A$ or $3A$.



Now, imagine we left someone in the room by themselves and told them to repeatedly do this operation over and over again. We leave for a while and come back. How do we know how many times they have done the operation? All we have is the initial point and the final point, NA.

It's very easy for the person who stayed in the room to tell you how many times they did the operation, but for you, it's difficult to determine how many times without going through the whole process again yourself. For example, If I show you this diagram:



You would struggle to tell me how many times I added A with itself to get NA, but I could easily tell you that I added A to itself 15 times. This becomes our trap-door function: easy to go one way but difficult to go the other.

Definition: A trapdoor function is a function, $f : X \to X$ such that for all $x \in X$, it is easy to calculate $y = f(x)$ but difficult to calculate $x = f^{-1}(y)$ without secret information. Computing $f(x)$ is a forward computation and computing $f^{-1}(y)$ is a backward computation.

Hence, for ECC our public key (what we share with the world) will be the initial point we choose (P), the final point (NP), and the method used to get NP. Our private key will be the value N, the number of operations we performed to get to NP from P.

Let's say we only left the person in the room for a short time. It would be much easier to figure out N by repeating their steps until we reached NP than if we left them in there for a much longer time. So, how do we make our path untraceable without spending a really long time doing the computation ourselves?

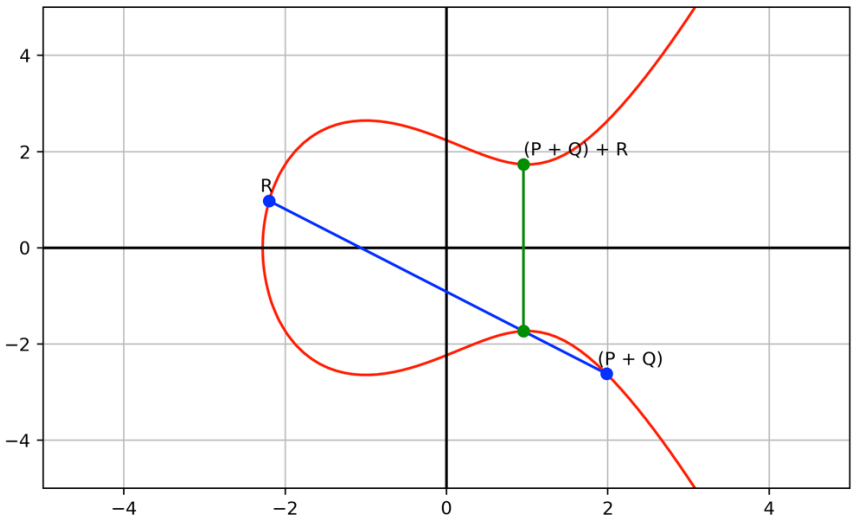
This part is fairly tricky and relies on group law***. We define a group as having:***

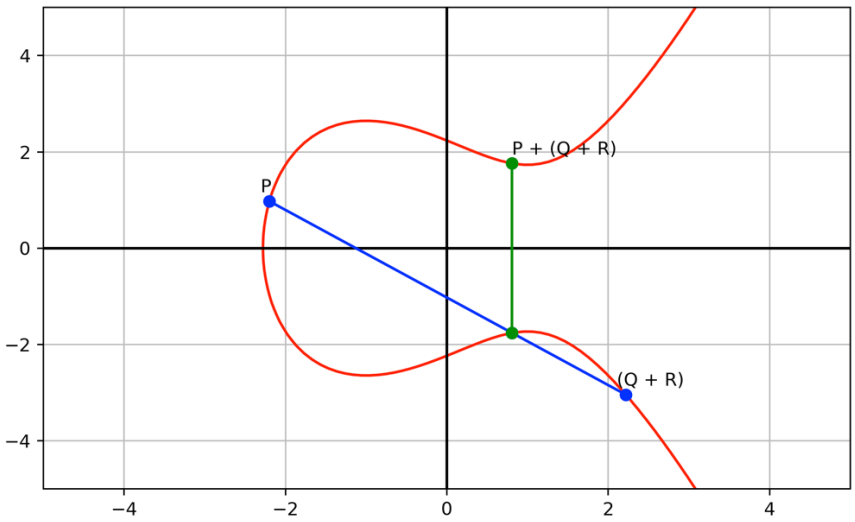
- A set of elements G
- A closed operation on the set that is associative
- An identity element
- Inverses under the set operation.

A group where the operation is also commutative is called an abelian group.

We have already ticked off the first of these criteria when we said that an elliptic curve is a set of points (or elements) that satisfy the equation $y^2 = x^3 + ax + b$. And, we've already come up with an operation on our set: '+'. But, can we show that this operation is associative?

Let's take our points P, Q and R from before. We want to show that $(P + Q) + R = P + (Q + R)$.



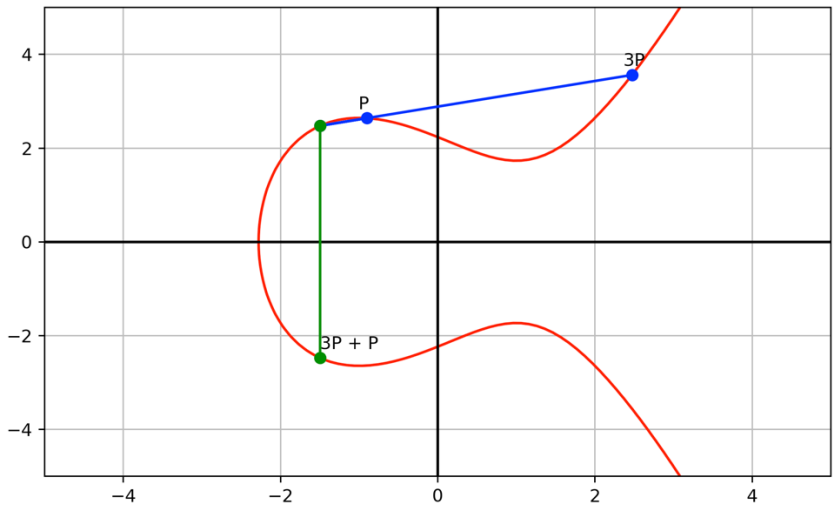


As you can see, they're the same point! If the diagrams didn't convince you, here is the output of the program I wrote to generate the curves:

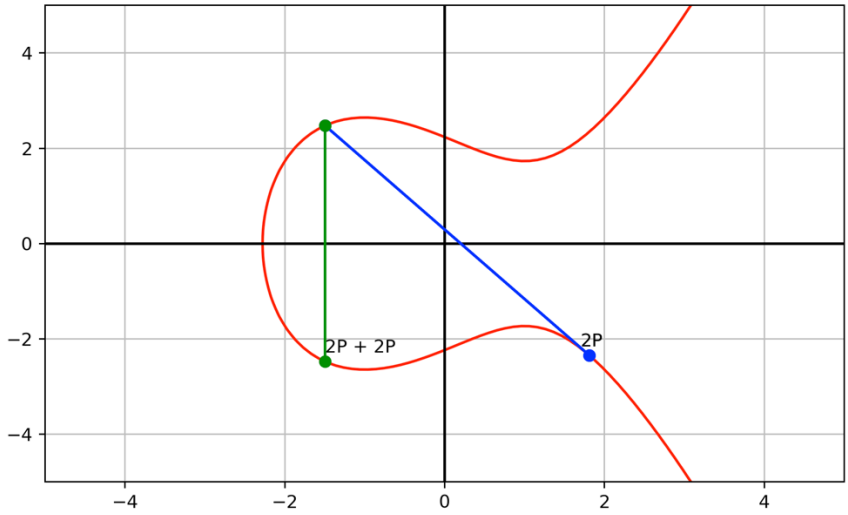
```
P is the point with coordinates:
(-2.1,1.427935572776307)
Q is the point with coordinates:
(0.2,2.099523755521713)
R is the point with coordinates:
(-2.2,0.975704873412036)
Adding P and Q yields:
(1.985260999471319,-2.620810794809187)
Adding Q and R yields:
(2.2192654305184454,-3.0450606800250277)
(P + Q) + R yields:
(0.9531851331698313,1.7339181913574129)
P + (Q + R)yields:
(0.9531851331698316,1.733918191357413)
```

Revisiting our method from before where we took the tangent from our initial starting point, we can also show that for example that for a point P: $P + 3P = 2P + 2P$.

$\boldsymbol{P + 3P}$:



$\boldsymbol{2P + 2P}$:



As above, we see that doing both ways will give us exactly the same point. It's obvious that calculating $4P$ can be done in fewer steps by calculating $2P + 2P$ instead of $P + P + P + P$.

Therefore, if we want to calculate a value NP fast, we just need to decompose NP into powers of 2 that we can add together.

For example, say we want to find $123P$. To decompose $123P$ into powers of 2, we start by representing it in binary: 111011.
This corresponds to:

$$2^0 + 2^1 + 2^3 + 2^4 + 2^5 + 2^6$$

Which is:

$$P + 2P + 8P + 16P + 32P + 64P$$

What we would do is:

- Add P to our sum, then double it
- Add 2P to our sum, then double it
- Double 4P
- Add 8P to our sum and double it
- Add 32P to our sum and double it
- Add 64P to our sum and double it

That's 6 steps as opposed to 123. Impressive! This is the double and add method. The pseudo-code for this is as follows. Let d be the binary representation of n, and m be $log_2(n)$.

```
Def double_add (n, P):
      N ← P
      Q ← 0
      for i from 0 to m do
         if the i^th bit of d is 1:
             Q ← point_add(Q, N)
         N ← point_double(N)
      return Q
```

128 is a fairly 'small' number, and you can imagine that we could calculate NP for extremely large values of N in this way. If this N gets big enough, it also becomes extremely difficult for someone to retrace our steps. This is called the Discrete logarithm problem.

Definition, the Discrete Logarithm Problem: Given two points P and Q, find the integer x such that $Q = xP$.

Elliptic Curve Diffie Hellman Key Exchange:

But where does encryption come into all of this? Recall that for encryption and decryption in an asymmetric cryptosystem, both parties must create a private and public key. Let's say we have two people, Alice and Bob. Both Alice and Bob agree to use P as a starting point (this is public knowledge). Alice begins by deciding on a number N (her private key) and creates her public key NP by using the methods outlined above. Bob also creates his private key M and his public key MP.

Now, Alice takes Bob's public key and starts adding the point to itself N times:

$$MP + MP + MP + \cdots . + MP = M \times NP.$$

Bob does the same with Alice's public key but M times:

$$NP + NP + NP + \cdots . + NP = N \times MP.$$

Note that they can do this very quickly using the double and add method we just outlined. By undergoing this process, Alice and Bob both arrive at the same point:

$$K = (N \times M)P.$$

Given that $N \times M$ is big enough, it will be very difficult to find N or M. Thus, only Bob and Alice will be able to know K, a point on the curve.

At this point, Alice and Bob can choose either the x or y coordinate of K to be their key for encryption. Let's say they choose the x coordinate. Now they can use a symmetric encryption system to encrypt data they wish to share between them by using the key. I have written a program to model ECDH (note that I have accounted for some rounding errors in my program and have made sure that the private keys of both Alice and Bob are relatively small). This can be found attached to this paper.

Introducing restrictions using finite fields:

So far, we have been performing all of our calculations over the real numbers. However, this isn't a very accurate representation of what the curves used for ECC actually look like. Now, rather than restricting our curves to the set of real numbers, let us restrict them further to a set of numbers in a fixed range.

Definition: A finite field is a field that contains a fixed number of elements.

A simple example of a finite field is the set of integers modulo p (some prime number), denoted by $GF(p)$. In a field, both addition and multiplication are associative and commutative. In order for this to be the case, there must be a unique identity element and every element must have a unique inverse. Note that the existence of unique inverses in $GF(p)$ is dependent on the fact that p is prime.

For $GF(p)$, addition and multiplication conform to modular arithmetic. But what does division look like? In $GF(p)$:

$\frac{x}{y} = x \times y^{-1}$, i.e. to divide x by y, multiply x by the multiplicative inverse of y. This can be found by using the extended Euclidean algorithm. Below is code I have written to find the inverse of a number modulo p using the extended Euclidean algorithm.

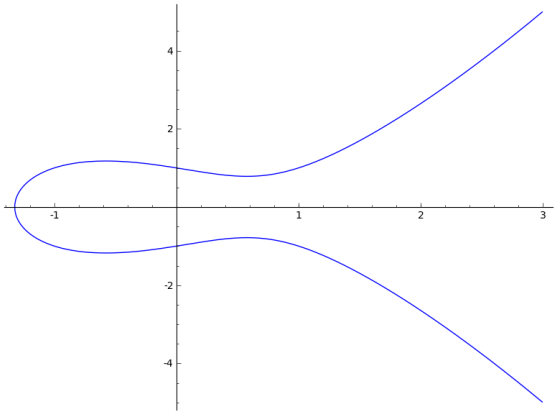
```
#An Implementation of the Ext. Euclidean Algorithm
#Using Bezout's identity gcd(a, b) = xa + yb
def ext_euclidean(a,b):
    prevx, x = 1, 0
    prevy, y = 0, 1
    while b:
        q = a//b
        x, prevx = prevx - q*x, x
        y, prevy = prevy - *y, y
        a, b = b, a % b
    #This returns the gcd, a and b from Bezout's
    return (a, prevx, prevy)

#returns the inverse of a number a, mod p
#i.e. finds b such that (a*b) == 1 (mod p)
def inverse(a,p):
    g, x, y = ext_euclidean(a,p)
    assert (x*a + y*p)%p == g
    if g != 1:
        raise Exception ("inverse does not exist!")
    else:
        return x % p
```

We now will restrict our elliptic curves and define them over $GF(p)$. Our equations for our elliptic curves are now in the form:

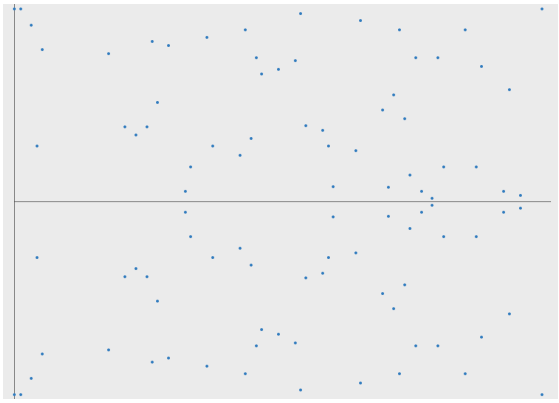
$$y^2 = x^3 + ax + b \text{ (mod p)}$$

Essentially what this translates to is to bringing numbers back to 0 when they reach a maximum (p).
For example, the curve, $y^2 = x^3 - x + 1$, originally looks like this:



From: https://arstechnica.com/information-technology/2013/10/a-relatively-easy-to-understand-primer-on-elliptic-curve-cryptography/2/

But when made mod 97 and we only plot integer values, it becomes this:



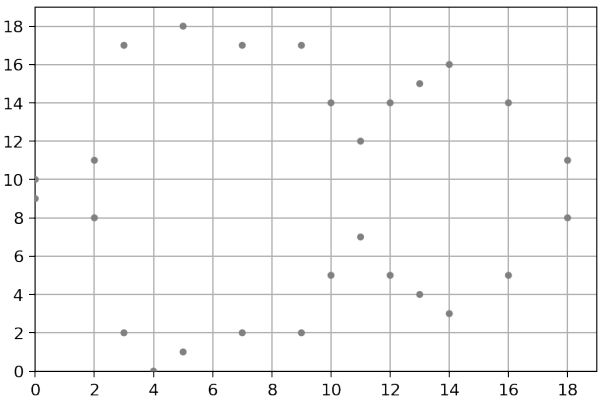
From: https://arstechnica.com/information-technology/2013/10/a-relatively-easy-to-understand-primer-on-elliptic-curve-cryptography/2/

The new plot doesn't really look like a 'curve' as we know it, it's like the original curve was wrapped around the edges. See that the curve is horizontally symmetrical about $p/2$. Even though the curve looks weird, it still forms an abelian group, like the original curve on the real numbers as mentioned above.

Let's try playing the same game we did with the original curve on this new one. Obviously, we now need to modify our 'addition' of points so that it still works in $GF(p)$. Before, we defined 'addition' by drawing a line between two points, taking the new point where this line intersected the curve and drawing a line vertically to the other side of the curve. The point where this hit was the final result of our addition.

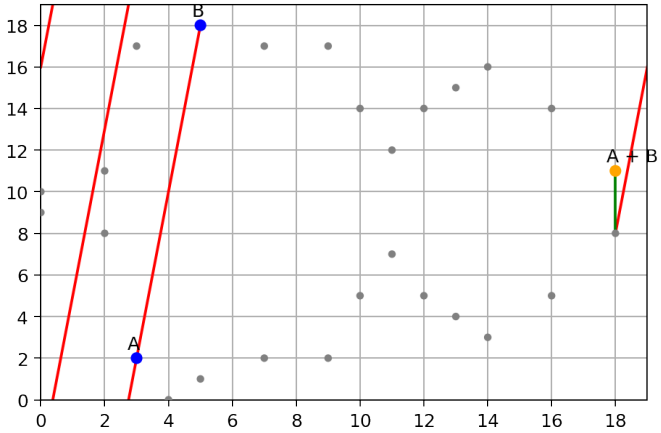
On the new curve, you can visualise the line between two points as a line that wraps around the borders (your curve is contained in a box of size $p \times p$), so that every time it hits the 'edge' of the box it is sent to the other side where it continues. This line will hit another plotted point (one of the dots) somewhere before being reflected about $p/2$ on your curve.

For the curve $y^2 = x^3 - 3x + 5$ (mod 19), the plot looks like this:



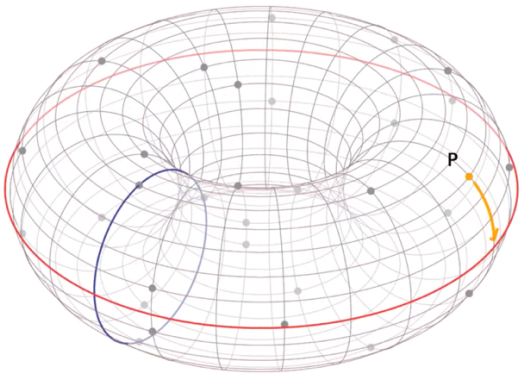
From: https://fangpenlin.com/posts/2019/10/07/elliptic-curve-cryptography-explained/

And when we 'add' the points A = (3, 2) and B = (5, 18) together, the procedure looks something like this:



From: https://fangpenlin.com/posts/2019/10/07/elliptic-curve-cryptography-explained/

This concept can also be represented by a donut shape like in the Trustica video series about ECC.



Digital Signature Schemes:

If Alice and Bob are sending each other messages using ECC, we may wonder how either Alice or Bob know that it is the other person sending them a message. Is there a way for Alice to send a message to Bob so he knows for certain that the message was from her?

Digital Signature Schemes are designed to be the digital equivalent of hand-written signatures. Digital signatures are numbers that are dependent on some secret only known by the sender and the message that they are sending.

In an ideal world, such a scheme should be unforgeable. This means that in our situation, even if Eve (the eavesdropper) is able to find Alice's Signature, she won't be able to successfully forge it herself. These digital signatures thus become useful in the real world, as they can be used to identify if data has been tampered with, authenticate that messages really came from where the sender says, and to prove that someone really did send some message in the past. We can use Elliptic curves to create Digital Signatures.

ECDSA: Elliptic Curve Digital Signature Algorithm

This section will get a bit complicated as it brings together everything we've been doing and will require some more mathematical understanding of previous sections.

Let us return to our example of Alice and Bob. Alice wants to send a message, m to Bob with a signature that verifies that it came from her.

1. *The first step Alice must take is to find the hash of m and truncate the hash's bit length so that it is of length n, the order of the subgroup generated by P in G.*

To understand what this means, we will go back and revisit our procedure of adding a point P to itself repetitively N times to achieve a new point NP. We will now formally define this process of adding a point to itself as scalar multiplication on the elements of the group G, i.e. the set of points on the elliptic curve. Recall our definition of a group earlier on:

We define a group as having:

- A set of elements G
- A closed operation on the set that is associative
- An identity element
- Inverses under the set operation.

A group where the operation is also commutative is called an abelian group.

By using scalar multiplication, we are able to create subgroups of G, where the elements are the multiples of the point P on the elliptic curve. It is true that:

$$xP + yP = (P + P + \cdots .. + P) + (P + \cdots + P) = (x + y)P,$$

or that adding multiples of the point P yields another multiple of P. This implies that the subgroup of G of the multiples of a point P is also closed under group law.

Note the following:

Definition: Let a be an element a group G with the identity element 1 and $\cdot$ as the group operation. The order of a, is the smallest integer n such that:

$$a \cdot a \cdot a \cdot \ldots . a \ (n \ times) = 1$$

The set:

$$\{a, a^2, a^3, a^4, \ldots, a^n\}$$

forms a cyclic subgroup of G of order n, where a is called a generator for that subgroup. All the multiples of P also form a cyclic subgroup with an order, lets also call it n, where P is the generator for the subgroup.

There is an algorithm called Schoof's algorithm that is able to compute the order of the set of points on an elliptic curve E over a finite field in polynomial time. Now, Lagrange's theorem states that the order of the subgroup where P is the generator, is a divisor of the order of the group G. Thus, if the order of G is prime let's say p, the order of the subgroup must be either 1 or p. If the order is p, then all of the elements of the subgroup must be generators, i.e. multiples of one another. Thus, for any curve $E(F)$ where F is a finite field, we are able to see if the order of the curve is prime and thus determine that all subgroups must also be of the same order.

Now we come to the next new point addressed in the first step of ECDSA, hashing. Hashing is a process that reduces any amount of text or data into a fixed output. Cryptographic hashes have the following properties:

- The same text will always result in the same hash
- It's quick to compute
- It should be hard to reverse
- It's hard to find two different texts that has the same hash
- If you make a small change to the text, it will change lots of things in the hash

The hash function used by Alice may be one like SHA-256. We'll call the truncated hash of m she creates z. Continuing on with our procedure of signing:

2. *Next, Alice will choose an integer k, such that* $1 \leq k \leq n - 1$ *(where n is the order of the subgroup* P*).*
3. *She will then calculate* $Q = kP$ *and r, the x coordinate of the point Q modulo n (* x_Q*).*
4. *If r is equal to 0, she will return to step 2.*
5. *Alice will then calculate* $s = k^{-1}(z + r * nA) \ mod \ m$*, where* nA *is her private key.* k^{-1} *is the multiplicative inverse of* k *mod* n*. This can be found using the Euclidean algorithm we outline above.*
6. *If s is equal to 0, we again return to step 2.*
7. *The digital signature is the pair* (r, s)

Now, in order for Bob to verify the signature, he will:

1. *Compute* $u_1 = s^{-1}z \; mod n$ and $u_2 = s^{-1}r \; mod n$
2. *Compute the point* $Q = u_1 * P + u_2 * A$, *where A is Alice's public key.*
3. *Verify that indeed* $r = x_Q \; mod \; n$

It should be noted that in order to calculate s, we had to calculate the inverse of k modulo n. This is only able to be computed if the order of the subgroup, n, is prime. This is why all standardised curves used for ECC have a prime order.

This can easily be coded yourself in python using the **fastecdsa** library. Below is some code I made using this library which creates a private and public key pair, signs a given message and then verifies the signature.

```
from fastecdsa import keys, curve, ecdsa

def gen_keys():
    private, public = keys.gen_keypair(curve.P256)
    return (private, public)

def sign(message, priv):
    (r,s) = ecdsa.sign(message,priv)
    return (r, s)
def verify(sig, message, pub):
    return ecdsa.verify(sig,message,pub)


if __name__ == "__main__":
    name = input("What's you're name? ");
    message = input(f"Hi {name}, What message would you like to send? ")
    priv, pub = gen_keys()
    print(f"Your private key is {priv}")
    print(f"Your public key is {pub}")
    sig = sign(message, priv)
    print("Your signature is {sig}")
    print("Verifying the signature using the public key.....")
    result = verify(sig, message, pub)
    print(result)
```

Known Attacks on ECC:

Elliptic Curve Cryptography is susceptible to attacks both by classical and quantum computers. In this section I will outline a few of the most well-known ones. First, a quick reminder of the Discrete Logarithm Problem.

Definition, Discrete Logarithm Problem: Given two points P and Q, find the integer x such that $Q = xP$.

Weak Curves:

Some elliptic curves can be easier to attack than others as they allow certain algorithms to solve the discrete logarithm problem relatively quickly. For example, for curves that have a group order equal to that of the finite field the discrete logarithm can be computed in polynomial time, as shown by Nigel P. Smart. This is known as Smart's attack.

Other curves that are vulnerable to attacks are super singular curves. These curves allow the elliptic curve discrete logarithm to be reduced to the regular discrete logarithm, such that the scalar can be found in less than exponential time.

Sony's problem:

Note that in the previous section when we were generating digital signatures using ECDSA, Alice had to choose an integer k.

If this k could be found or it was predictable in any way, Eve the eavesdropper would be able to find the private key. This mistake was made by Sony a few years ago when they suffered an attack on their ECDSA protocol due to using a fixed k rather than one that was randomly generated.

The PlayStation 3 console could only run games that were signed by Sony using ECDSA, so that if a new game was created for the console, it had to be approved by Sony and signed first.

Sony's key could easily be found in this situation by buying two separate games for the console before extracting their hashes ($z_1$and z_1), signatures ($(r1, s_1), (r2, s_2)$) and the domain parameters.

Now, because $r = xQ$ mod n in both cases, r1 must be equal to r2.

Next, note that $(s_1 - s_2)\ mod\ n = k^{-1}(z_1 - z_2)\ mod\ n$ (by the equation above for finding s)

Multiplying both sides by k yields: $k(s_1 - s_2)\ mod\ n = (h1 - h2)\ mod\ n$

And dividing both sides by $(s_1 - s_2)$ gives: $k\ mod\ n = (z_1 - z_2) * (s_1 - s_2)^{-1}\ mod\ n$

This allows us to calculate k using the two hashes and signatures of the games.

Now, given k we can get the private key nA by the definition of s: $s = k^{-1}(z + r * nA)\ mod\ m$. Rearranging gives: $nA = r^{-1}(sk - z)$ and we are able to directly calculate the private key.

Baby Step Giant Step:

First, begin by noting that any integer x can be written in the form:

$$x = am + b, \text{ where } a, m \text{ and } b \text{ are integers.}$$

Using this, we can now write the scalar in the discrete logarithm problem as:

$$Q = xP$$

$$Q = (am + b)P$$

And rearrange to yield:

$$Q = amP + bP$$

$$Q - amP = bP$$

Essentially what this algorithm does is it attempts to find collisions between the baby steps, $Q - amP$ and the Giant steps: bP. This is how the algorithm works:

1. *Calculate m = $\sqrt{n}$*
2. *for every b in 0 to m, store bP in a hash table*
3. *for every a in 0 to m:*
    a. *calculate amP*
    b. *calculate $Q - amP$*
    c. *check if there is a bP in the hash table with the equivalent value*
    d. *If this point exists, we're done, and we've found x.*

The reason the algorithm gets its name is that in calculating bP, we take 'little' steps by incrementing b and, when we calculate amP, we take 'big' steps by incrementing a. Just think about how large the increments of am are if m is a really big number.

Now, let's step through the algorithm and see why it works by considering again $Q = amP + bP$.

When $a = 0$, then we are essentially checking if $Q = bP$ for m values of b.

When $a = 1$, then we are checking $if\ Q = mP + bP$. This is equivalent to us checking Q against all values from mP to $2mP$ (as b ranges from 0 to m)

If we continue in this fashion, when $a = m - 1$, we are checking if $Q = (m-1)P + bP$. This is the same as comparing Q to all the points from $(m-1)mP\ to\ m^{2P} = nP$.

This means we've checked all possible points with m additions and multiplications for the baby steps and up to m multiplications and additions for the giant steps. This algorithm has complexity $O(\sqrt{n})$

Pollard's Rho attacks:

The disadvantage of the Baby Step Giant Step method is that it requires a lot of memory, as each value of bP has to be stored in memory.

For this algorithm, we will consider a different problem:
Find integers a, b, A and B such that

$$aP + bQ = AP + BQ.$$

Now, noting that $Q = xP$, we can write this as:

$$aP + b(xP) = AP + B(xP)$$

And rearrange to give:

$$(a + bx)P = (A + Bx)P$$

$$(a - A)P = (B - b)xP$$

This is essentially the same as solving $A - a = B - bx$. Just remember that all coefficients are modulo n. Therefore:

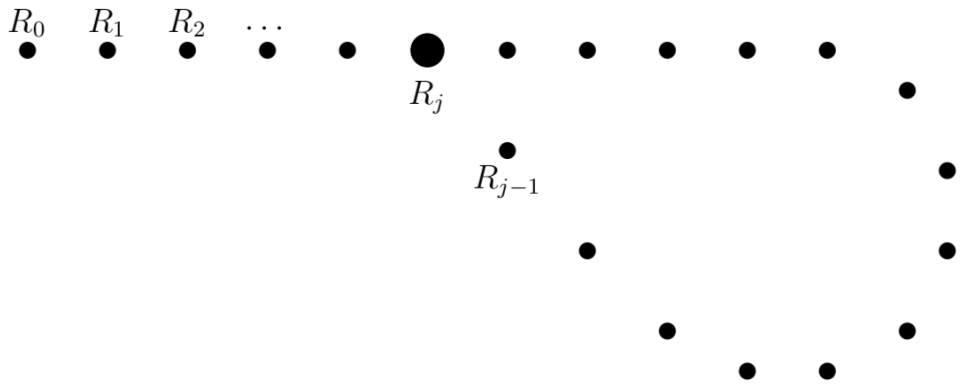
$$a - A = (B - b)\, x\, (mod\ n)$$

$$x = (a - A)(B - b)^{-1}\, (mod\ n)$$

Essentially, from here we will begin generating a pseud-random sequence of points. Let's call these points R, R_2 ….. Every R_i will be in the form $R_i = a_i P + b_i Q$. To generate this sequence, we will use a function, let's call it f, that will take in the previous point and generate the next. i.e.

$$f(R_i) = (a_{i+1}, b_{i+1})$$

This means that the next point is based on the previous. After a while, we will eventually find a 'loop' in our sequence, such that two values in the sequence are the same i.e. $R_i = R_j$. Our sequence ends up looking a little bit like the Greek letter Rho (ρ).



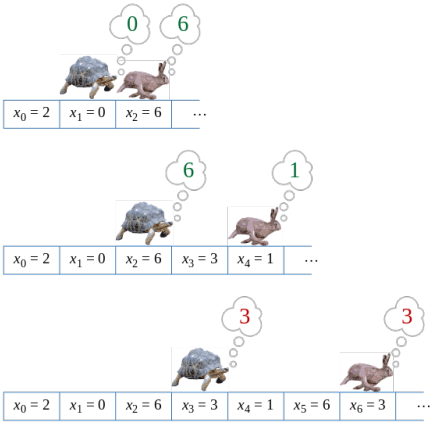
Reference: https://www.maths.unsw.edu.au/sites/default/files/mandyseetthesis.pdf

The reason this must occur is that we only have a finite number of points, which means that there will be repetition at some point down the line.

So, how do we detect the cycle? The cycle can be detected using Floyd's cycle-finding algorithm (also known as the tortoise and the hare algorithm). This entails moving two pointers (the tortoise and the hare) through our sequence of values at different rates until

they both point at equal values. I.e. the tortoise and the hare have found pairs (a, b) and (A, B) such that $aP + bQ = AP + BQ$. This algorithm also has time complexity $O(\sqrt{n})$.

For example, on the sequence: 2, 0, 6, 3, 1, 6, 3 the image below demonstrates how both pointers move at different rates until a match is found.



From: https://en.wikipedia.org/wiki/Cycle_detection#Algorithms

Shor's Algorithm + Quantum Attacks:

In March of 2016, the NSA published an article called: "the beginning of the end for encryption schemes?" following the successful demonstration of Shor's algorithm on scalable hardware. This, coupled with the announcement by the NSA that they intended to avoid ECC due to quantum attacks in 2015, brought into question the security of ECC given advances in Quantum Computing.

Quantum computing, first introduced in 1832 by Richard Feynman, is thought to be the key to destroying all modern asymmetric cryptographical systems. This is because these systems are reliant on mathematical problems that are difficult to compute, such as factorizing large prime numbers (as in RSA) or the Discrete Logarithm problem (as in ECC). In recent years, there has been an increasing demand for cryptographic protocols that can withstand quantum attacks as previous asymmetric systems such as ECC are at risk.

To understand this kind of attack, we will have a very brief overview of how quantum computers work and how quantum computing differs from classical computing.

In a classical computer, information is stored in bits, which are either in the state zero or one. However, quantum computers use fundamental blocks called qubits, particles (Quantum mechanics has its basis in phenomena) that are not zero or one but can be in both states simultaneously. This is known as super-position, and it means that quantum computers can solve extremely difficult and complex problems. This is because doing operations on qubits in superposition means that you can act on both states at the same time.

Another interesting phenomenon in quantum computing is quantum entanglement. When two qubits are entangled, they become a single object that is capable of having 4 different states and, by changing the single qubits the whole entangled qubit will also change. This means that quantum computers allow for true parallel processing power, and by doing the maths, an n-qubit quantum computer will have the capacity to process 2^n operations in parallel. Thus, Quantum Computers can break asymmetric protocols as they can search the entire key-space in feasible time.

Now, we will introduce Shor's Algorithm. In 1994, Peter Shor published a paper: "Algorithms for Quantum Computation: Discrete Logarithms and Factoring", in which he proved that large integer factorisation would change drastically with the introduction of Quantum Computers.

We will use the example of factoring the number 15 into its prime factors. To do so, we will use a 4-qubit register. This can be thought of like a normal 4-bit register. A 4-qubit register is enough to calculate the factorisation of 15, as 15 is 1111 in binary.

Shor's Algorithm is as follows:

- n = the number we are factorising, in this case 15
- x = a number between $1 < x < n - 1$ that is randomly chosen
- This number x is then raised to the number of possible states of the register (the power) and then divided by n. This is stored in another 4-bit register, which will contain the superposition results.

Note that the powers of the register (the possible values that it can take) in this case are the numbers 0 to 15. If we do the above and choose $x = 2$, the remainders of the calculations look like this:

Reg 1:	1	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Reg 2:	1	2	4	8	1	2	4	8	1	2	4	8	1	2	4	8

Now, if we take the length of the repeating sequence $(1, 2, 4, 8)$, let's call it F, then we are able to find a possible factor:

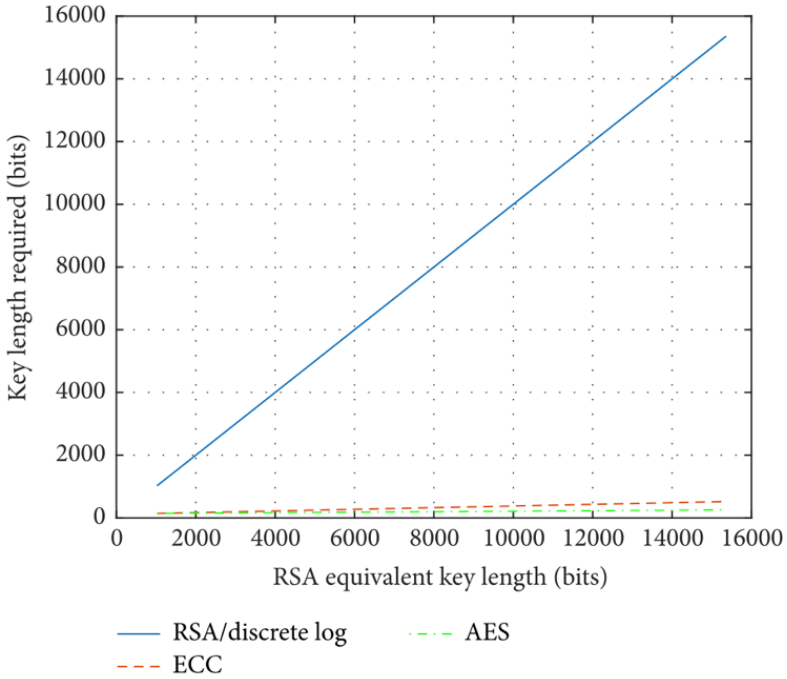
$$P = x^{(F/2)} - 1$$

Now if P isn't prime, we repeat the process with different values (keeping n the same). This algorithm can also be used to solve discrete logarithm problems as in ECC. This is a lot harder to explain, but it utilises the Quantum Fourier Transform, as opposed to the Classical Fast Fourier Transform, which allows it to run in polynomial time.

Analysis of the Security of ECC + Comparison of RSA and ECC security:

Both ECC and RSA are secure in the sense that they are based on mathematical problems that are computationally infeasible to solve using classical computing methods. It should also be noted that RSA in particular has withstood the test of time, as it continues to provide security for countless electronic systems today.

Each system can provide an arbitrary level of security, dependent on their key length. For example, a 256-bit ECC key is equivalent to RSA 3072-bit key (keep in mind that most keys used today for RSA are 2048 bits). A comparison between key length and security for RSA and ECC can be seen in the following diagram:



Reference: https://www.hindawi.com/journals/scn/2017/1467614/

This is the main differentiator between both cryptosystems: ECC is able to provide the same cryptographic strength as RSA with smaller key sizes. This is particularly advantageous for use on devices with lower processing power, something the world is continuing to demand with the use of mobile phones and small devices.

Could I find a way around ECC?

I highly doubt it. However, if the parameters of ECC were not properly managed, now, I would have a better idea of how to attack a system that used ECC using one of the above methods of attack. Maybe in the future, if Quantum Computing progresses to a point where the discrete logarithm problem is tractable, I will be able to crack this problem, but until then, I think that there is a reason why ECC has been so widely adopted and endorsed.

Conclusion:

In conclusion ECC is a relatively secure cryptosystem that provides many advantages given that certain measures are taken (i.e. choosing specific curves that are less vulnerable to the attacks above). However, with the rise of Quantum Computing this cryptosystem will most likely be phased out in favour of new cryptographic protocols that are robust in the face of quantum attacks.