Elliptic curve cryptography

How Bitcoin works?

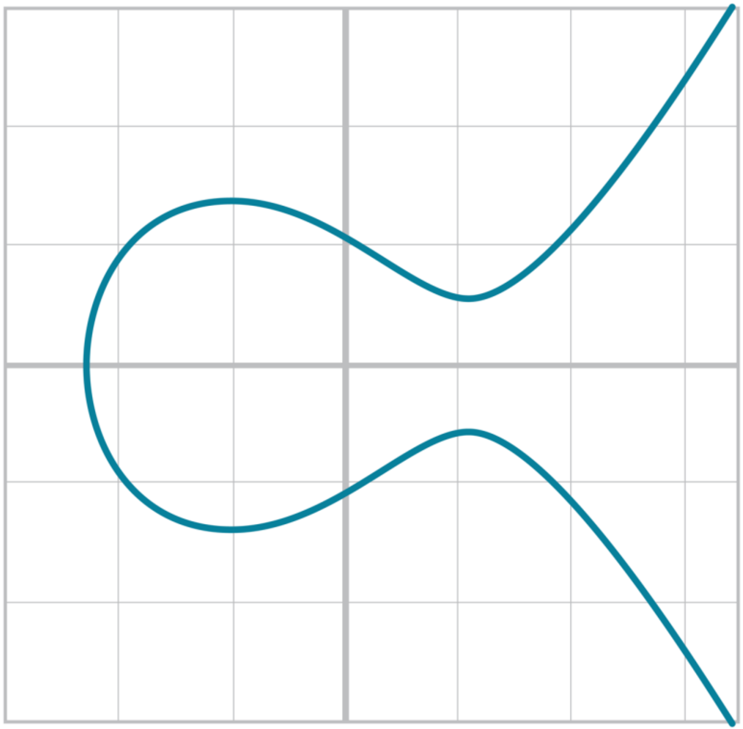
Elliptic curves

Definition of an elliptic curve:

$$\{ (x,y) : y^2 = x^3 + ax + b \} + \{ I \}$$

Elliptic curves

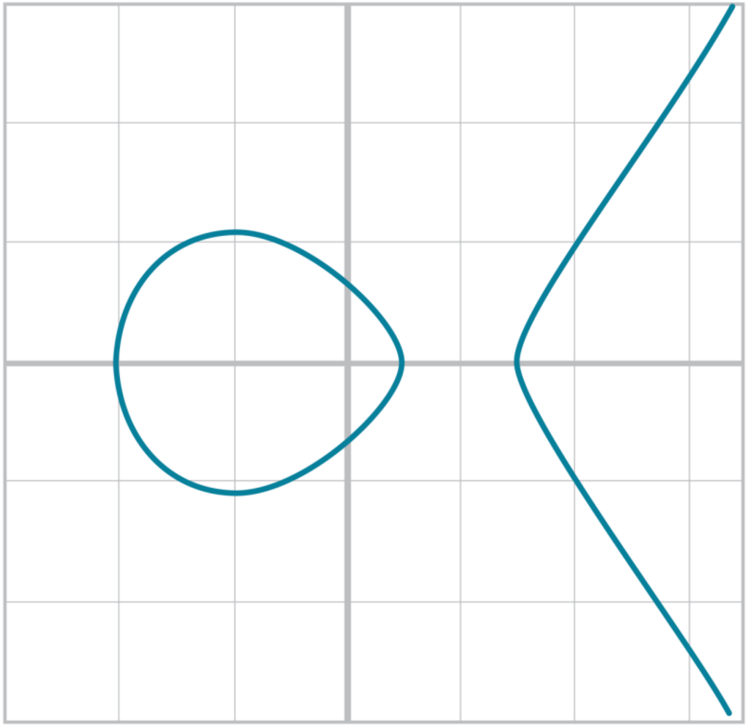
Examples



https://github.com/jimmysong/programmingbitcoin/blob/master/ch02.asciidoc

Elliptic curves

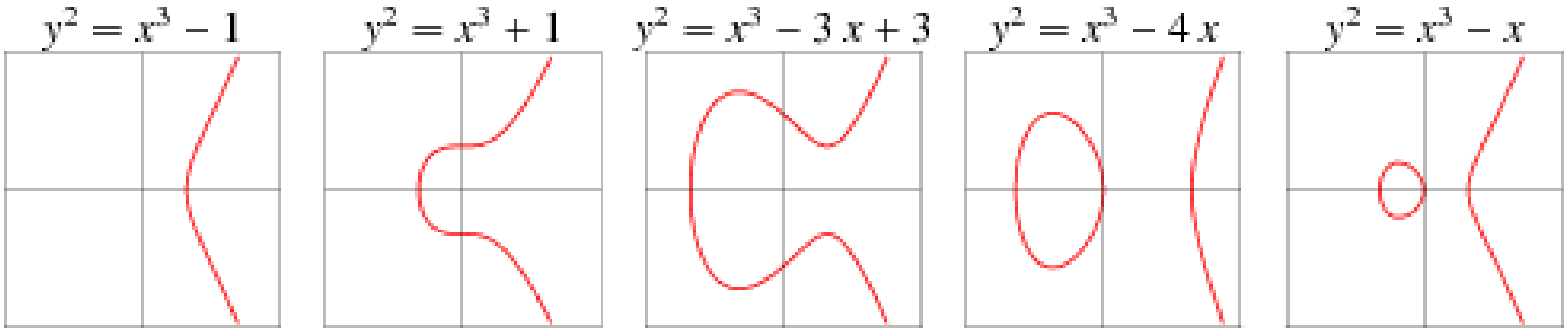
Examples



https://github.com/jimmysong/programmingbitcoin/blob/master/ch02.asciidoc

Elliptic curves

Examples

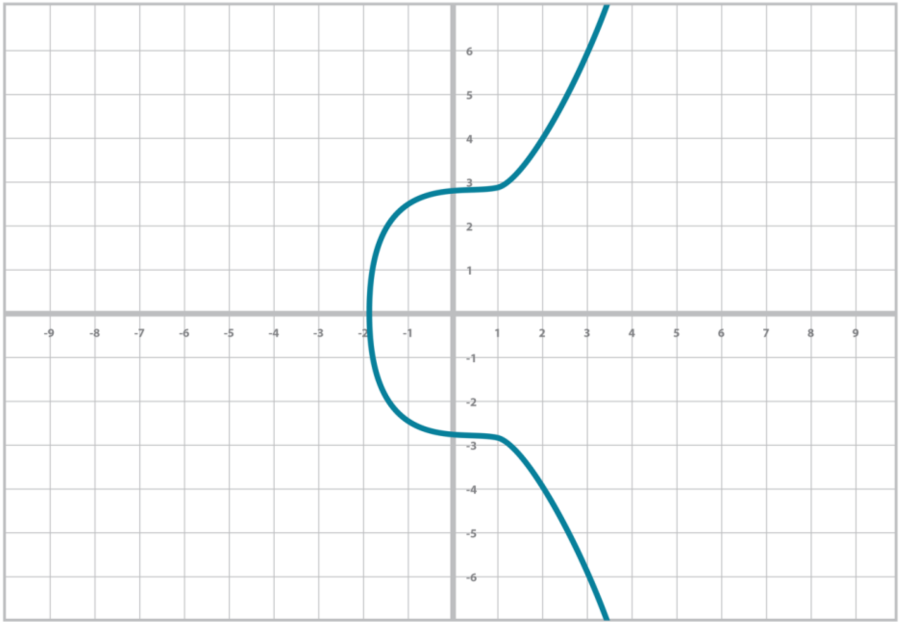


https://mathworld.wolfram.com/EllipticCurve.html

Curva in Bitcoin

secp256k1

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



https://github.com/jimmysong/programmingbitcoin/blob/master/ch02.asciidoc

Elliptic curves

In reality, we only care about the points on a curve

Elliptic curves

```python
class Point:

    def __init__(self, x, y, a, b):
        self.a = a
        self.b = b
        self.x = x
        self.y = y
        # x being None and y being None represents the point at infinity
        # Check for that here since the equation below won't make sense
        # with None values for both.
        if self.x is None and self.y is None:
            return
        # make sure that the elliptic curve equation is satisfied
        # y**2 == x**3 + a*x + b
        if self.y**2 != self.x**3 + a * x + b:
            # if not, throw a ValueError
            raise ValueError('({}, {}) is not on the curve'.format(x, y))

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

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

Operations with points

Adding two points on an elliptic curve

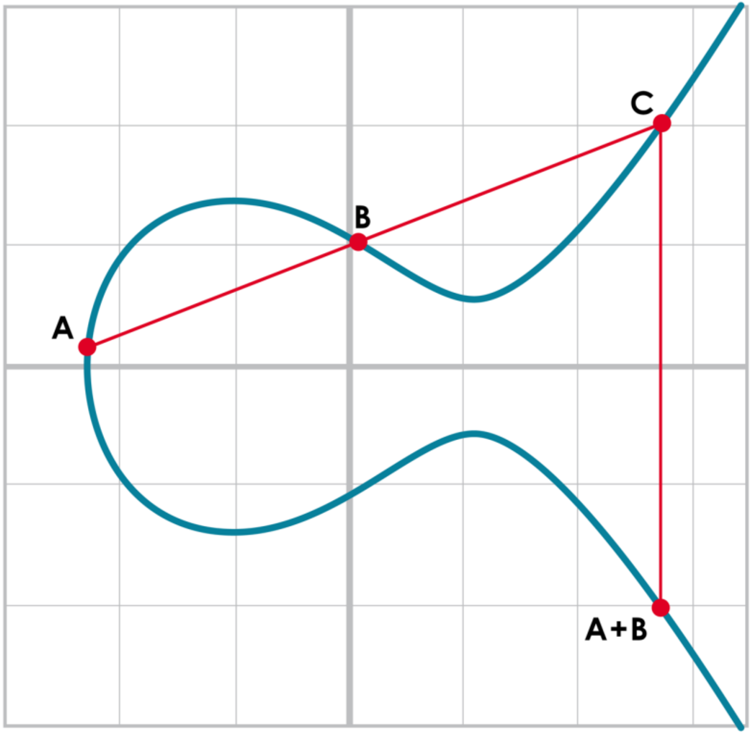
The most important operation:

- Adding two points
- We’ll this allows us to have a group

Adding two points

Intuition:

1. Two points define a line
2. The line intersects the curve in a third point
3. Mirror image around the x axis is the sum

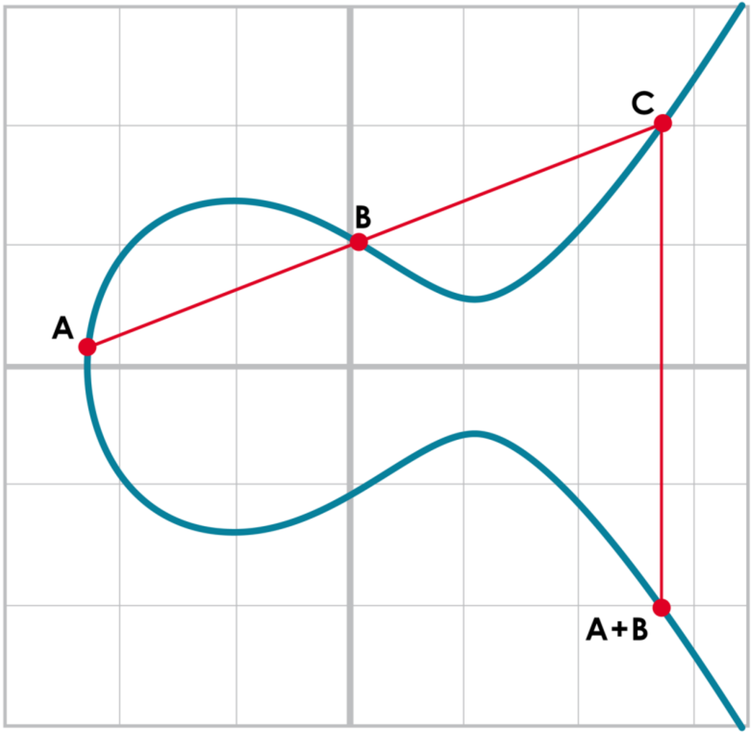


Adding two points

Intuition:

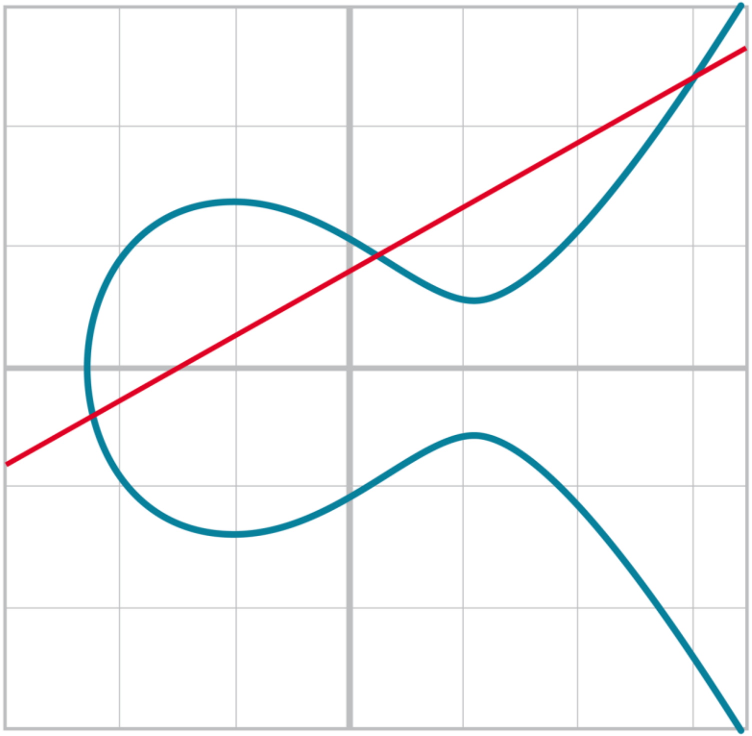
1. Two points define a line
2. The line intersects the curve in a third point
3. Mirror image around the x axis is the sum

But this does not always happen!



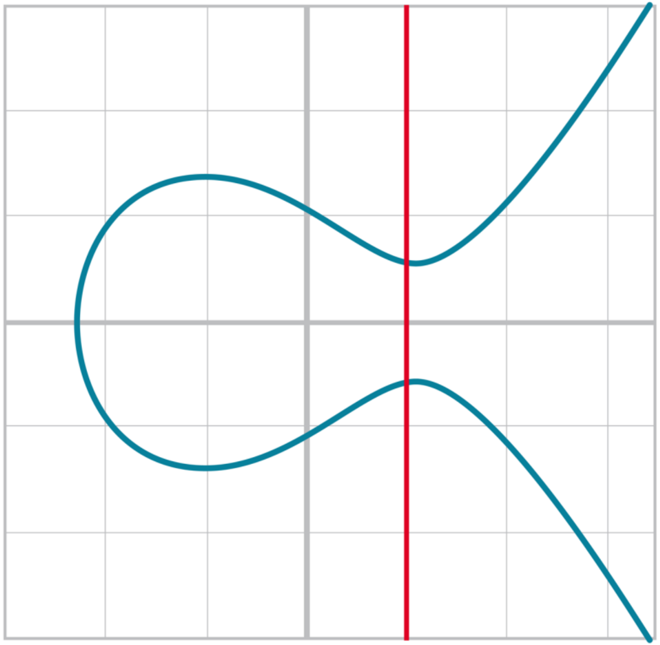
The curve and the line

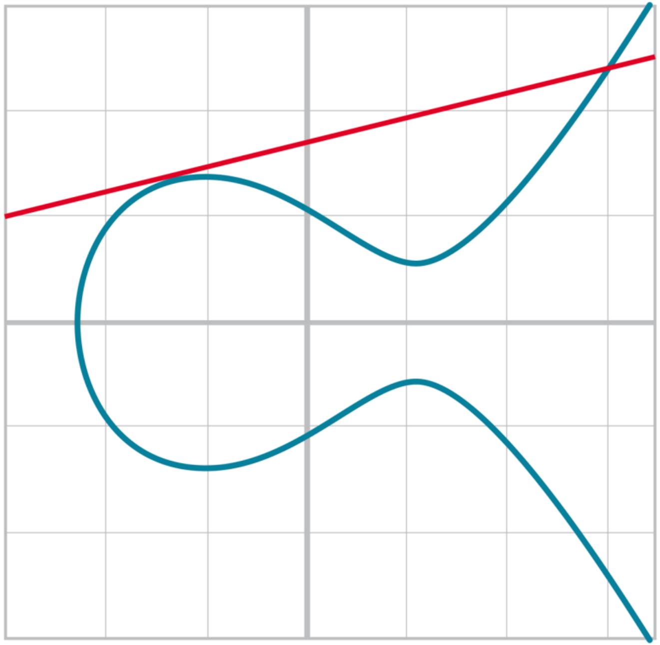
Case 1: three points of contact



The curve and the line

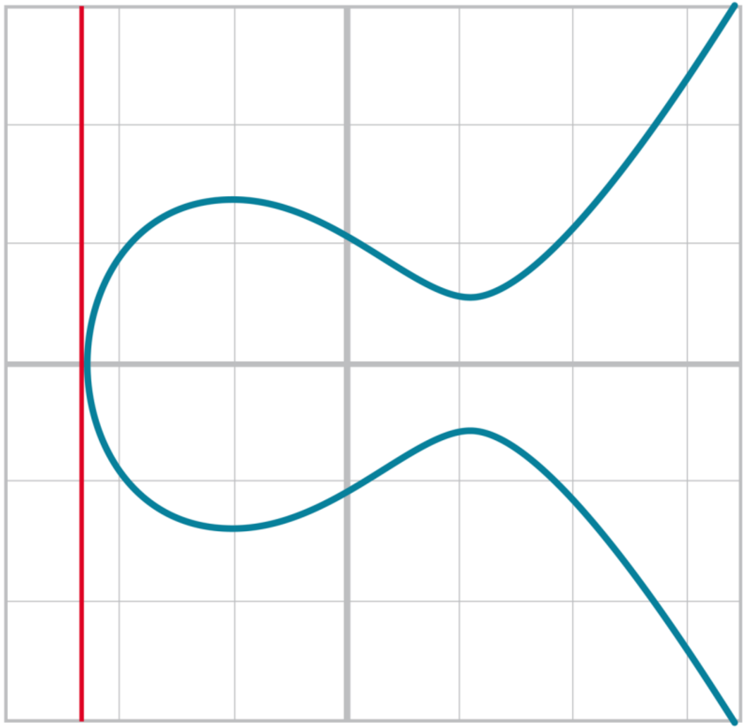
Case 2: two points of contact





The curve and the line

Case 3: one point of contact



Adding two points

Property of the sum of two numbers:

- *Neutral element:* there is an element 0 s.t. 0 + A = A + 0 = A, for all A
- *Commutativity:* A + B = B + A
- *Associativity:* (A + B) + C = A + (B + C)
- *Inverse element*: there is –A s.t. A + (-A) = (-A) + A = 0, for all A

Adding two points

Neutral element

$\{ (x,y) : y^2 = x^3 + ax + b \} + \{ I \}$

Adding two points

Neutral element

{ (x,y) : $y^2 = x^3 + ax + b$ } + { I }

Adding two points

Neutral element

$$\{ (x,y) : y^2 = x^3 + ax + b \} + \{ I \}$$

I + A = A, for all A on the curve

Adding two points

Inverse element

$$\{ (x,y) : y^2 = x^3 + ax + b \} + \{ I \}$$

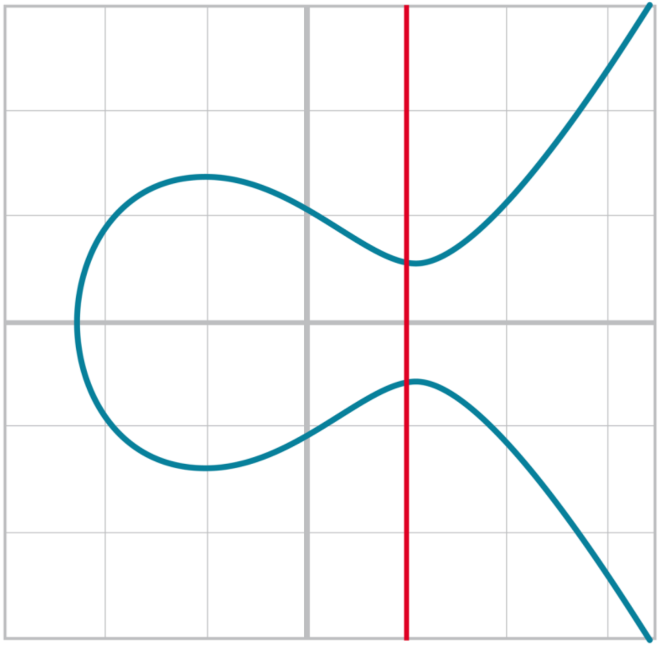
$A = (x,y) \Rightarrow -A = (x,-y)$

$A + (-A) = I$

Adding two points

Inverse element

Intuition: line through two points, the third point of intersection (reflected) is the sum

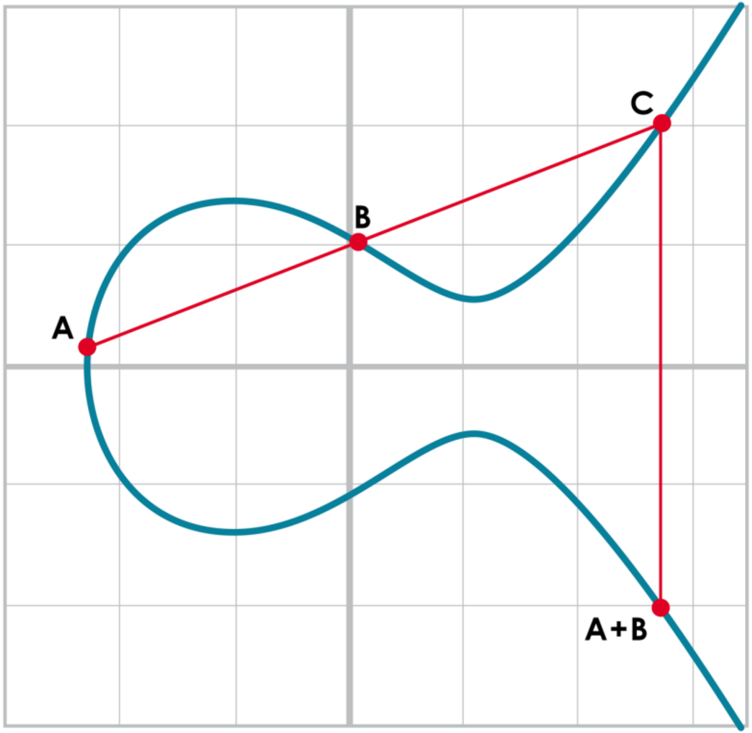


Adding two points

The most important case

Intuition:

1. Two points define a line
2. The line intersects the curve in a third point
3. Mirror image around the y axis is the sum



Adding two points

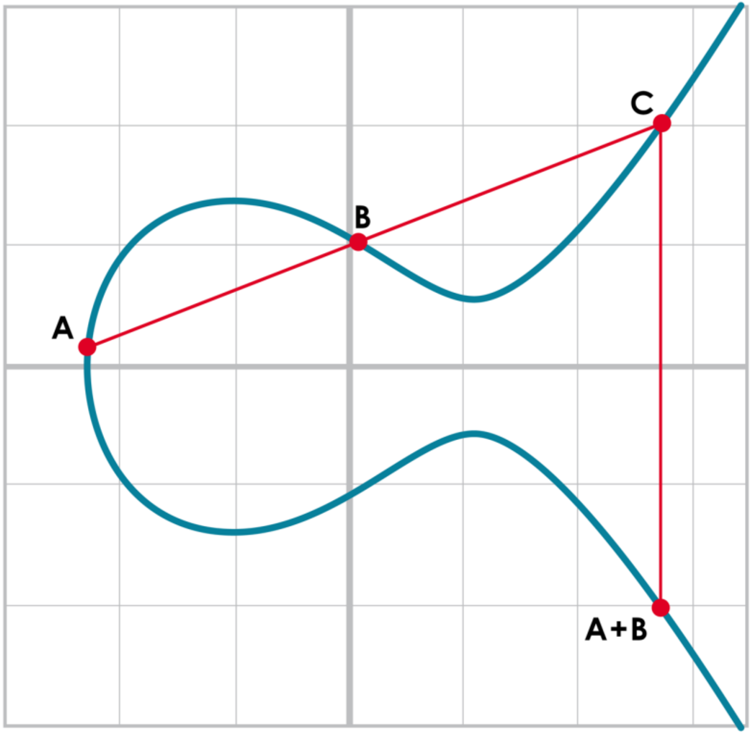
The most important case

What are the coordinates of A + B?

P1 = A = (x1,y1)
P2 = B = (x2,y2)
P3 = A + B = (x3,y3)

P1 + P2 = P3

If I know the coordinates of P1 and P2, how can I compute the coordinates of P3?



Adding two points

The most important case

$P1 + P2 = P3$

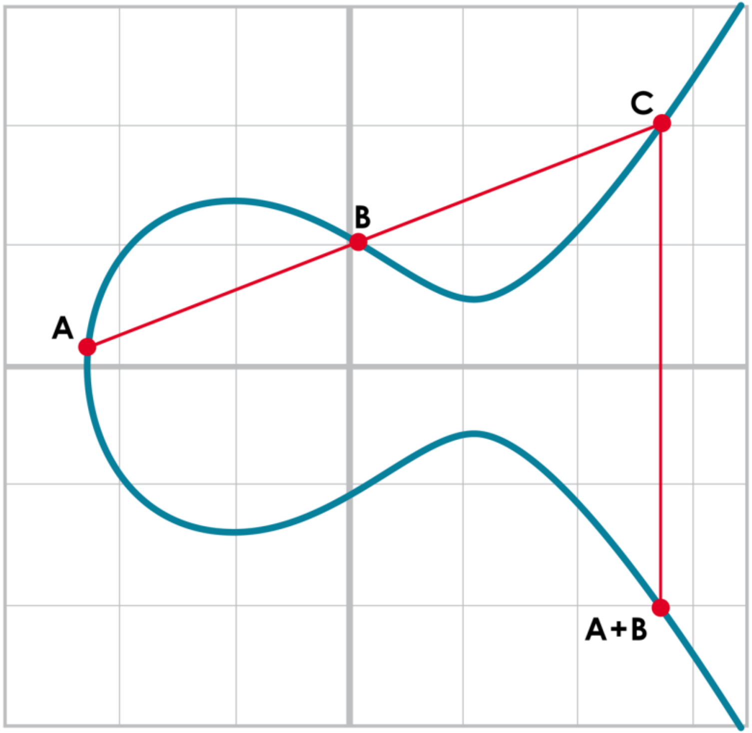
$P1 = (x1,y1)$, $P2 = (x2,y2)$, $P3 = (x3,y3)$

$x1 \neq x2$

$y = s(x - x1) + y1$

$s = \frac{y2-y1}{x2-x1}$

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



Adding two points

The most important case

P1 + P2 = P3

P1 = (x1,y1), P2 = (x2,y2), P3 = (x3,y3)

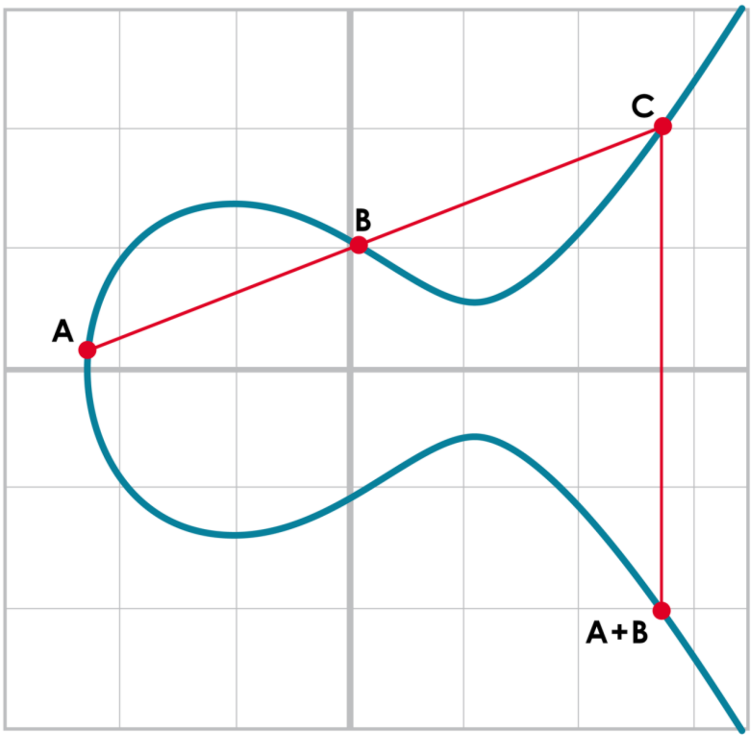
x1 ≠ x2

y = s(x- x1) + y1

$s = \frac{y2-y1}{x2-x1}$

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

$y^2 = (s(x - x1) + y1)^2 = x^3 + ax + b$



Adding two points

The most important case

P1 + P2 = P3

P1 = (x1,y1), P2 = (x2,y2), P3 = (x3,y3)

s = $\frac{y2-y1}{x2-x1}$ $\qquad$ $y^2 = x^3 + ax + b$

$y^2 = \boxed{(\, s(x - x1) + y1\,)^2 = \; x^3 + \mathrm{ax} + \mathrm{b}}$

$$x^3 \; - \; s^2x^2 + (a + 2s^2x1 \; - 2sy1)x + \mathrm{b} - s^2x1^2 + 2sx1y1 \; - y1^2 = 0$$

Adding two points

The most important case

P1 + P2 = P3

P1 = (x1,y1), P2 = (x2,y2), P3 = (x3,y3)

s = $\frac{y2-y1}{x2-x1}$ $\qquad y^2 = x^3 + ax + b$

$y^2 = (s(x - x1) + y1)^2 = x^3 + ax + b$

$x^3 - s^2x^2 + (a + 2s^2x1 - 2sy1)x + b - s^2x1^2 + 2sx1y1 - y1^2 = 0$

Equals to 0 for
x = x1,x2,x3

Adding two points

The most important case

P1 + P2 = P3

P1 = (x1,y1), P2 = (x2,y2), P3 = (x3,y3)

s = $\frac{y2-y1}{x2-x1}$ $y^2 = x^3 + ax + b$

$y^2 = (\ s(x - x1) + y1\ )^2 = \ x^3 + ax + b$

$x^3 \ - \ s^2x^2 + (a + 2s^2x1 \ - 2sy1)x + b - s^2x1^2 + 2sx1y1 \ - y1^2 = 0$

(x-x1)(x-x2)(x-x3) = 0

Equals to 0 for
x = x1,x2,x3

Adding two points

The most important case

P1 + P2 = P3

P1 = (x1,y1), P2 = (x2,y2), P3 = (x3,y3)

s = $\frac{y2-y1}{x2-x1}$ $y^2 = x^3 + ax + b$

$y^2 = (s(x-x1) + y1)^2 = x^3 + ax + b$

Vieta!!! https://en.wikipedia.org/wiki/Vieta%27s_formulas

$x^3 - s^2x^2 + (a + 2s^2x1 - 2sy1)x + b - s^2x1^2 + 2sx1y1 - y1^2 = 0$

(x-x1)(x-x2)(x-x3) = 0

Adding two points

The most important case

P1 + P2 = P3

P1 = (x1,y1), P2 = (x2,y2), P3 = (x3,y3)

s = $\frac{y2-y1}{x2-x1}$ $\quad$ $y^2 = x^3 + ax + b$

$y^2 = (s(x - x1) + y1)^2 = x^3 + ax + b$

$x^3 - s^2x^2 + (a + 2s^2x1 - 2sy1)x + b - s^2x1^2 + 2sx1y1 - y1^2 = 0$

$x^3 - (x1+x2+x3)x^2 + (x1x2 + x1x3 + x2x3)x - x1x2x3 = 0$

Vieta!!! https://en.wikipedia.org/wiki/Vieta%27s_formulas

$s^2 = x1 + x2 + x3$

Adding two points

The most important case

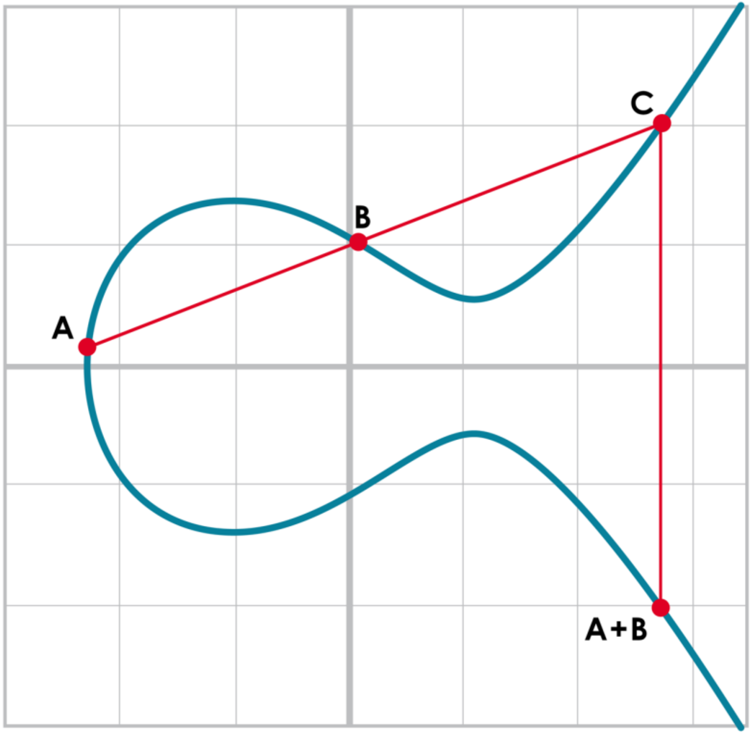
P1 + P2 = P3

P1 = (x1,y1), P2 = (x2,y2), P3 = (x3,y3)

x1 ≠ x2

x3 = $s^2 - x1 - x2$

y3 = s(x1 - x3) – y1



Adding two points

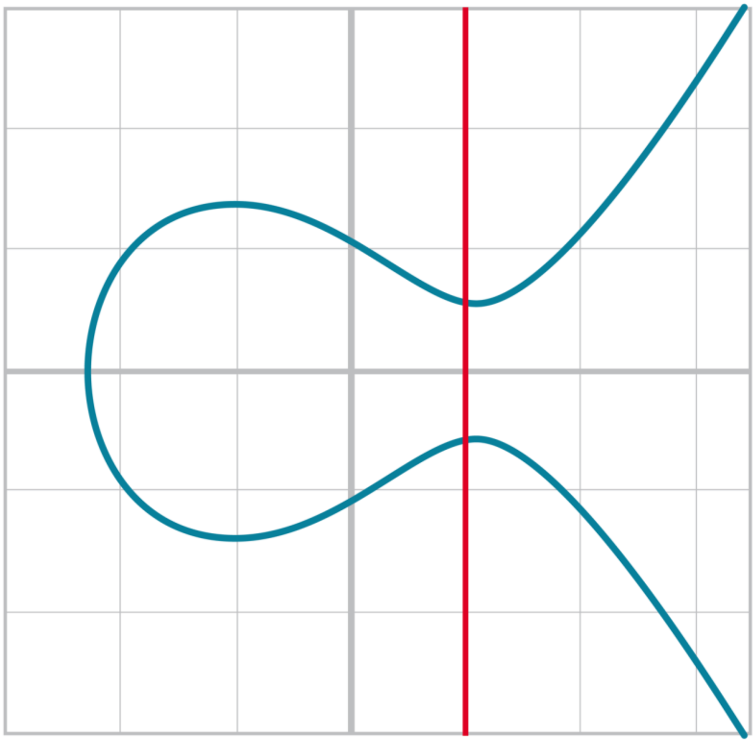
The other cases

$P1 + P2 = P3$

$P1 = (x1,y1) = P2 = (x2,y2)$, $P3 = (x3,y3)$

$x1 = x2$, $y1 \neq y2$

$P3 = I$



Adding two points

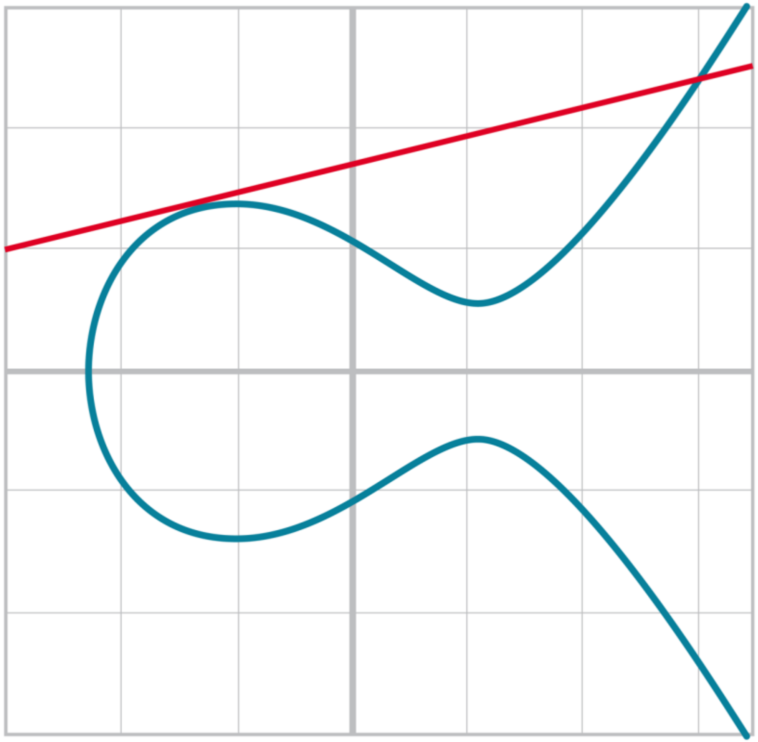
The other cases

P1 + P2 = P3
P1 = P2 = (x1,y1)

Assume that two different points tend to each other

s of this tanget curve: the derivate

s = $(3x1^2$ + a)/(2y1)
x3 = $s^2 - 2x1$
y3 = s(x1 – x3) – y1



Adding two points

The last case

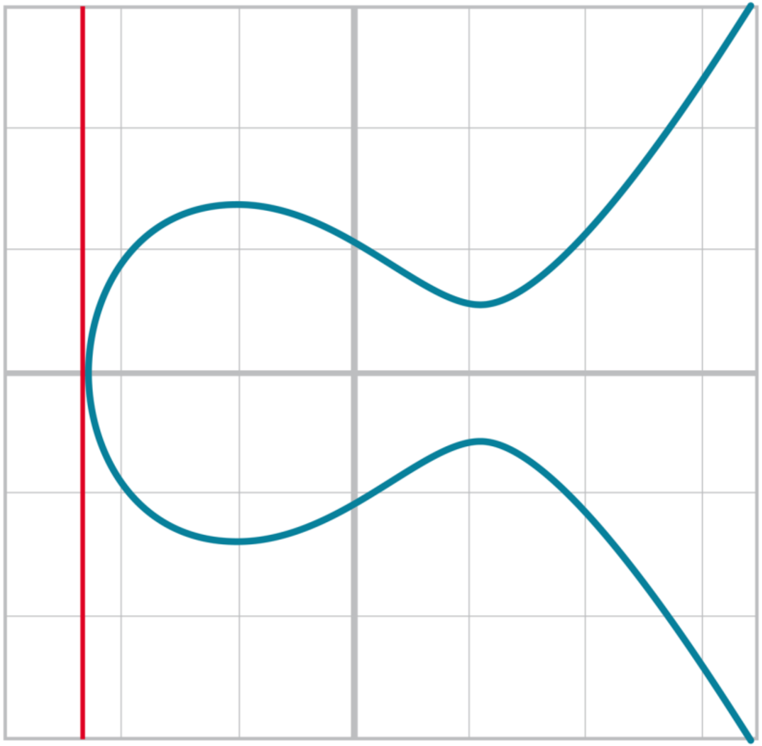
P1 + P2 = P3
P1 = P2 = (x1,y1)

Two points tend to each other

s of this tanget curve: the derivate

s = $(3x1^2 + a)/(2y1)$
x3 = $s^2 - 2x1$
y3 = s(x1 – x3) – y1

y1 = 0 $\Rightarrow$ P1 + P1 = I



Adding two points

Property of the sum of two numbers:

- *Neutral element:* there is an element 0 s.t. 0 + A = A + 0 = A, for all A
- *Commutativity:* A + B = B + A
- *Associativity:* (A + B) + C = A + (B + C)
- *Inverse element*: there is –A s.t. A + (-A) = (-A) + A = 0, for all A

Easy to show with our formulas!

Adding two points

Implementation

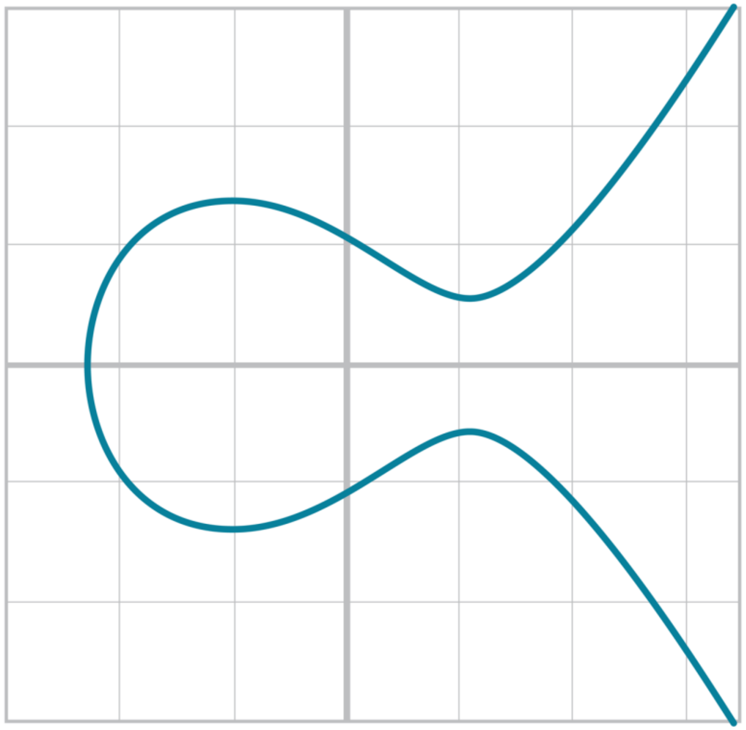
TODO.md × Programa.md × curves.py

```python

class Point:

    ...

    def __add__(self, other):
        if self.a != other.a or self.b != other.b:
            raise TypeError('Points {}, {} are not on the same curve'.format(self, other))
        # Case 0.0: self is the point at infinity, return other
        if self.x is None:
            return other
        # Case 0.1: other is the point at infinity, return self
        if other.x is None:
            return self

        # Case 1: self.x == other.x, self.y != other.y
        # Result is point at infinity
        if self.x == other.x and self.y != other.y:
            return self.__class__(None, None, self.a, self.b)

        # Case 2: self.x ≠ other.x
```

Elliptic curves

Our graphs are over real numbers

In practice: curves are over **finte fields**



Finite fields

A **Finite field** is a tuple F = (G,+,·,0,1) with these properties:

1) *Closure* : $\mathrm{a} + \mathrm{b} \in$ G, y a · b ∈ G, for all a,b ∈ G
2) *Additive neutral*: 0 ∈ G, y 0 + a = a, for all a ∈ G
3) *Multiplicative neutral*: 1 ∈ G, y 1 · a = a, for all a ∈ G
4) *Additive inverse*: for all a ∈ G, exists –a ∈ G t.q. a + (-a) = 0
5) *Multiplicative inverse*: for all a ∈ G, a ≠ 0, exists a^{-1} ∈ G, s.t. a · a^{-1} = 1

Finite fields

Prototype of a Finite field:

$F_p = \{0,1,2,\ldots,p-1\}, p\ a\ \boldsymbol{prime}\ number$

$a\ +_p\ b = \ (a+b)\ \%\ p\quad (modulo\ p)$

$a\ \cdot_p\ b = \ (a \cdot b)\ \%\ p\quad (modulo\ p)$

Finite fields

Inverse

How do we define the additive inverse?

-a % p $(a \in F_p \rightarrow -a = (p - a) \in F_p)$

E.g. $F_p = \{0,1,\ldots,18\}$, p = 19

-14 = 5 $\in F_p$

14 $+_p$ 5 = (14 + 5 = 19) % 19 = 0

Finite fields

Inverse

How do we define the multiplicative inverse?

$$a \cdot_p b = (a \cdot b)\ \%\ p \quad (modulo\ p)$$

Little Fermat's Theorem: If p is prime, and a > 0 is a natural number, then:

$$\boldsymbol{a^{p-1} \equiv 1\ (mod\ p)}$$

$a^{-1} = a^{p-2}$ % p

$a \cdot_p a^{-1} = (a \cdot a^{p-2})$ % p = 1

Finite fields

Exponentiation

Little Fermat's Theorem : If p is prime, and a > 0 is a natural number, then:

$$\boldsymbol{a^{p-1} \equiv 1 \ (mod\ p)}$$

$a^k \in G = \ a^k$ % p

k >> p -> k = m · (p-1) + n

a^k % p = $a^{\mathrm{m} \cdot (\mathrm{p}-1)}$ % p · $a^{\mathrm{n}} \% \, p = a^{\mathrm{n}}$ % p = $a^{\mathrm{k\%p}}$ % p

Finite fields

Implementation

```python
class FieldElement:

    def __init__(self, num, prime):
        if num >= prime or num < 0:
            error = 'Num {} not in field range 0 to {}'.format(
                num, prime - 1)
            raise ValueError(error)
        self.num = num
        self.prime = prime

    def __repr__(self):
        return 'FieldElement_{}({})'.format(self.prime, self.num)

    def __eq__(self, other):
        if other is None:
            return False
        return self.num == other.num and self.prime == other.prime

    def __ne__(self, other):
        # this should be the inverse of the == operator
        return not (self == other)
```

And much more!

Elliptic curves

En el salvaje

Elliptic curves are considered over Finite fields, and not over the reals

y^2 = x^3 + *7 over* F_{103}

(17,64) is on the curve:

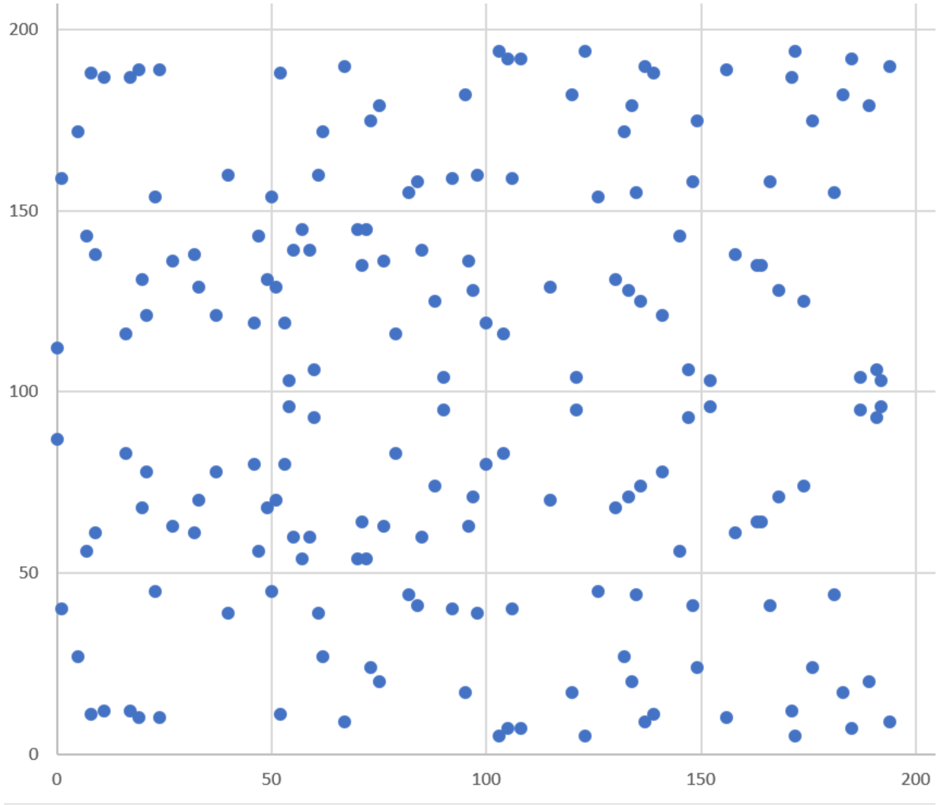
$y^2 = \ 64^2 \ \% \ 103 = 79$
x^3 + 7 = 17^3 + 7 % 103 = 79

Elliptic curves

Curva $y^2 = x^3 + 7$ over F_{223}

Simmetric around the middle

How many points can a curve have over a finite field?



Elliptic curves

Adding two points

Does anything change over finite fields?

P1 + P2 = P3
P1 = (x1,y1), P2 = (x2,y2), P3 = (x3,y3)

1) x1 ≠ x2

x3 = $s^2 - x1 - x2$
y3 = s(x1 - x3) – y1
s = $\frac{y2-y1}{x2-x1}$

Elliptic curves

Adding two points

Does anything change over finite fields?

P1 + P2 = P3

P1 = (x1,y1), P2 = (x2,y2), P3 = (x3,y3)

1) x1 ≠ x2

x3 = $s^2 - x1 - x2$ % p

y3 = s(x1 - x3) – y1 % p

s = $\frac{y2-y1}{x2-x1} \% p$

2) P1 = P2

x3 = $s^2 - 2x1 \% p$

y3 = s(x1 – x3) – y1 % p

s = $(3x1^2$ + a)/(2y1) % p

Scalar multiplication

The most important operation over ecc

P

P + P

P + P + P

P + P + P + P

.

.

.

Scalar multiplication

The most important operation over ecc

$P = 1 \cdot P$

$P + P = 2 \cdot P$

$P + P + P = 3 \cdot P$

$P + P + P + P = 4 \cdot P$

.

.

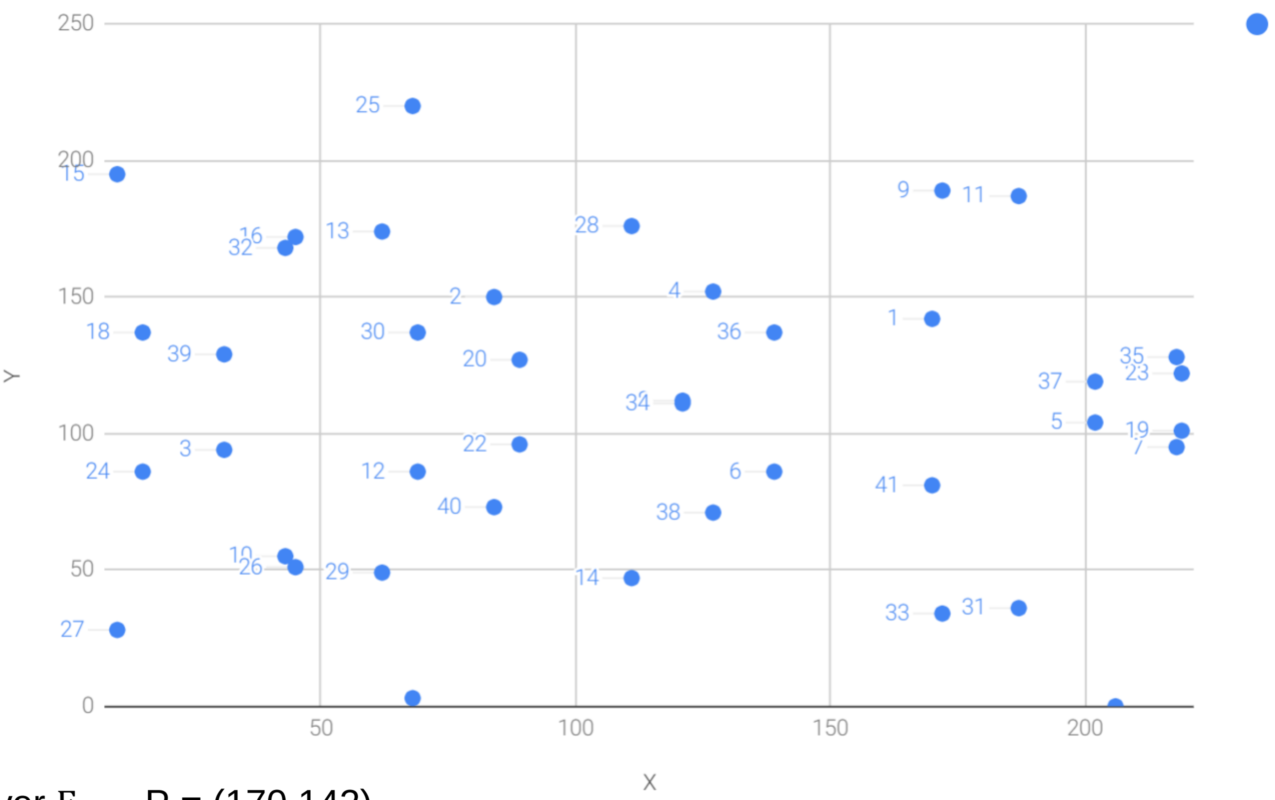
.

$a \cdot P$, for any natural number a

Scalar multiplication!!!

Scalar multiplication

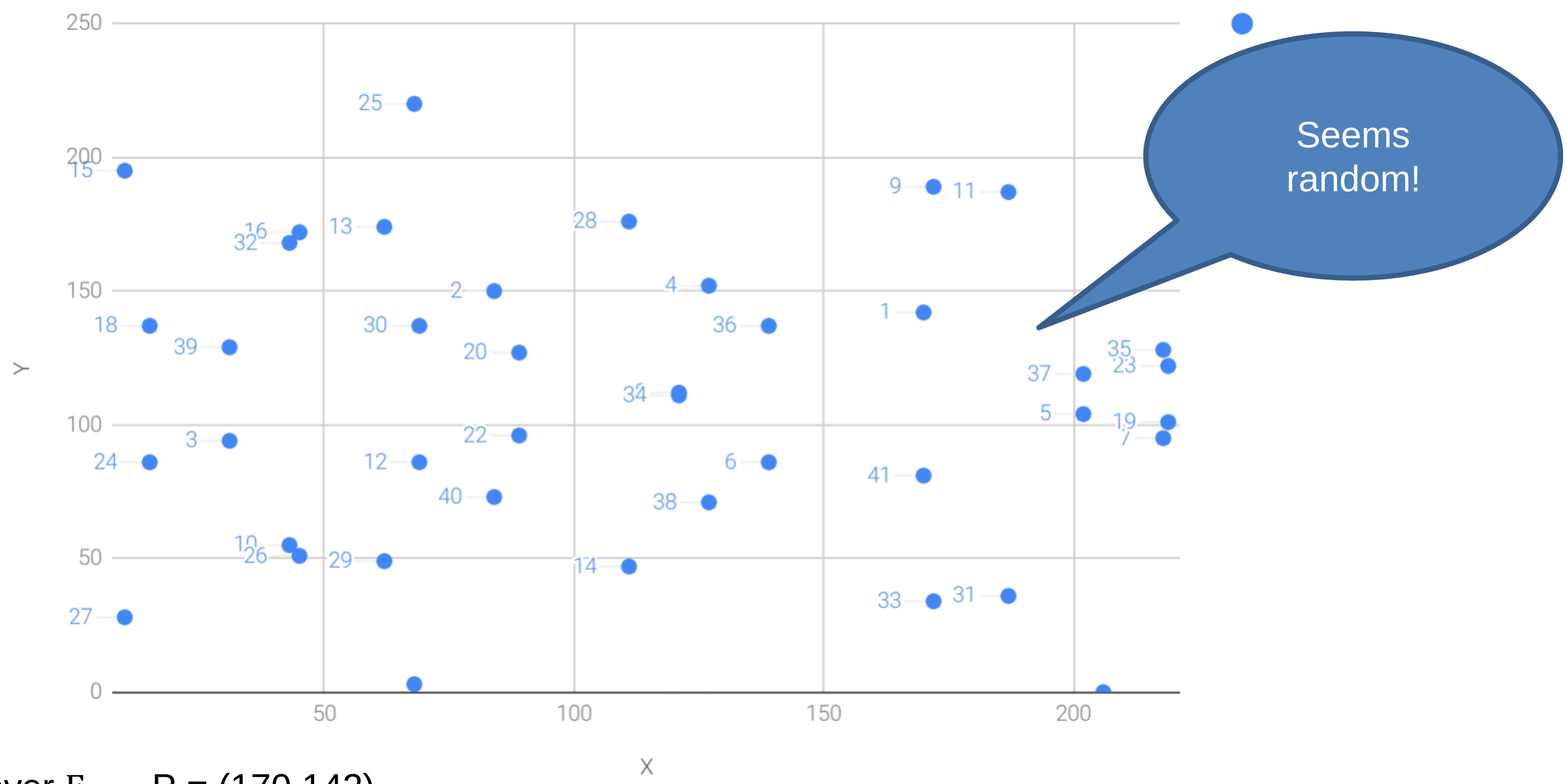
What does this look like graphically?



$y^2 = x^3 + 7$ over F_{223}, P = (170,142)

Scalar multiplication

What does this look like graphically?



$y^2 = x^3 + 7$ over F_{223}, P = (170,142)

Scalar multiplication

The discrete logarithm problem

$R = e \cdot P$

If I have P and R, the best algorithm to find e needs to try out all the possible e

I.e. There is no analythic formula for scalar division

The discrete log problem: invert the scalar product in an arbitrary group

One of the most important problems in Computer Science: ¿is there a polynomial algorithm for solving the discrete log problem?

Scalar multiplication

A fundamental property

I have an elliptic curve over a finite field $\mathrm{F_p}$:

- Number of points on my curve is finite, say m
- A take some G on the curve:
- The set $\{1 \cdot \mathrm{G}, 2 \cdot \mathrm{G}, 3 \cdot \mathrm{G}, \ldots\}$ is finitoe(P+Q is on the curve, for P,Q on the curva)
- So there must exist an n s.t. $\mathrm{n} \cdot \mathrm{G} = \mathrm{I}$ (point at infinity)
- Why?

Scalar multiplication

A fundamental property

I have an elliptic curve over a finite field $\mathrm{F_p}$:

- Number of points on my curve is finite, say m
- A take some G on the curve:
- The set {1 · G, 2 · G, 3 · G,…} is finitoe(P+Q is on the curve, for P,Q on the curva)
- So there must exist an n s.t. n · G = I (point at infinity)
- Why?

The curve with the point addition is a **finite group**

{1 · G, 2 · G, 3 · G,…, n · G} is a cyclic subgroup of the points on the curve

- How can we show this?

Scalar multiplication

The discrete logarithm problem

R = e $\cdot$ P

If I have P and R, the best algorithm for finding e has to try ***all posible e***

Maximum e is the number of points on the curve

The brute-force algorithm is considered to be exponential!!!

- Has to deal with the number representation (length of binary vs size)

Scalar multiplication

The discrete logarithm problem

R = e $\cdot$ P

Why is this called *the discrete logaritm*? (And not the division?)

Notation for groups: G = (A, $\cdot_{\mathrm{G}}$), and not (A, $+_{\mathrm{G}}$)

Therefore: P $\cdot_{\mathrm{G}}$ P $\cdot_{\mathrm{G}}$ P $\cdot_{\mathrm{G}}$ P $\cdot_{\mathrm{G}}$ P $\cdot_{\mathrm{G}}$ P $\cdot_{\mathrm{G}}$ P = P^7

We are solving P^7 = R, knowing P y R

So we are looking for 7 = $\log_{\mathrm{P}} R$

Scalar multiplication

A smarter algorithm

Double and add

$151 \cdot P$

$151 = 10010111_2$

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

Scalar multiplication

A smarter algorithm

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

$151 = 10010111_2$

P

1) SUM = I
2) SUM += P
3) P += P (= 2P)
4) SUM += P
5) P += P (= 4P)
6) SUM += P
7) P += P (= 8P)
8) P += P (= 16P)
9) SUM += P
10) P += P (= 32P)
11) P += P (= 64P)
12) P += P (= 128P)
13) SUM += P

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

Scalar multiplication

A smarter algorithm

$e \cdot P = (d_1 d_2 \ldots d_k)_2 \cdot P$

```
current = P
sum = I

for i = k downto 1
        if d_k == 1
                sum += current
        current += current
```

Elliptic curve in Bitcoin

secp256k1

What do we need to define an elliptic curve in the real world?

1) $y^2 = x^3 + ax + b$ → Parameters a and b

2) Over a finite field F_p → order p of the field

3) A generator $G = (G_x, G_y)$ which is a point on the curve

4) The subgroup $\{G, 2G, 3G, \ldots, nG = I\}$ generated by G → order n of the subgroup

Elliptic curve in Bitcoin

secp256k1

What do we need to define an elliptic curve in the real world?

1) $y^2 = x^3 + ax + b$ → Parameters **a** and **b**

2) Over a finite field F_p → **p** is the order of the field

3) A generator **$G = (G_x, G_y)$** which is a point on the curve

4) The subgroup {G,2G,3G,…,nG = I} generated by G → order **n** of the subgroup

Elliptic curve in Bitcoin

secp256k1

What do we need to define an elliptic curve in the real world?

1) $y^2 = x^3 + ax + b$ → Parameters **a** and **b**

2) Over a finite field F_p → **p** is the order of the field

3) A generator **$G = (G_x,G_y)$** which is a point on the curve

4) The subgroup $\boxed{\{G,2G,3G,\ldots,nG = I\}}$ generated by G → order **n** of the subgroup

Everything happens here!

Elliptic curve in Bitcoin

secp256k1

What is going on here?

- $y^2 = x^3 + ax + b$ over F_p defines a finite *group* (points with the addition of points)
- The *subgroup* {G,2G,3G,…,nG = I} generated by G is a subgroup of this group

We will be solving the *discrete logarithm problem* in {G,2G,3G,…,nG = I}

If n is prime we can define the inverse of any scalar in the finite *field* F_n **!!!**

Elliptic curve in Bitcoin

secp256k1

1) $y^2 = x^3 + 7$ → Parameters **a = 0** and **b = 7**

2) Over a finite field F_p → order of the field **$p = 2^{256} - 2^{32} - 977$**

3) A generator **$G = (G_x, G_y)$** which is a point on the curve

G_x = 0x79be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798
G_y = 0x483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8

4) The subgroup {G,2G,3G,…,nG = I} generated by G → the order **n** of the subgroup

n = 0xfffffffffffffffffffffffffffffffebaaedce6af48a03bbfd25e8cd0364141

Elliptic curve in Bitcoin

secp256k1

1) $y^2 = x^3 + 7$ → Parameters **a = 0** and **b = 7**

2) Over a finite field F_p → order of the field **$p = 2^{256} - 2^{32} - 977$**

n is a prime number

3) A generator **$G = (G_x, G_y)$** which is a point on the curve

G_x = 0x79be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d[illegible]9f2815b16f81798
G_y = 0x483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554[illegible]9c47d08ffb10d4b8

4) The subgroup {G,2G,3G,…,nG = I} generated by G → the order **n** of the subgroup

n = 0xfffffffffffffffffffffffffffffffebaaedce6af48a03bbfd25e8cd0364141

Elliptic curve in Bitcoin

secp256k1

$p = 2^{256} - 2^{32} - 977$

n = 0xfffffffffffffffffffffffffffffffebaaedce6af48a03bbfd25e8cd0364141

The two are almost 2^{256} → both coordintes and scalars can be expressed in 256 bits

Recall: sha256 = 256 bits!!!

Elliptic curve in Bitcoin

Keys

$y^2 = x^3 + 7$

n = 0xfffffffffffffffffffffffffffffffebaaedce6af48a03bbfd25e8cd0364141

G = (G_x,G_y)

Private key: a number *e* of 256 bits (in reality from [0,n])

Public key: the point $P = e \cdot G$ on our curve

e is my secret

P is public and serves as my identity!

Elliptic curve in Bitcoin

Keys

$y^2 = x^3 + 7$

n = 0xfffffffffffffffffffffffffffffffebaaedce6af48a03bbfd25e8cd0364141

$G = (G_x, G_y)$

Private key: a number *e* of 256 bits (in reality from [0,n])

Public key: the point $P = e \cdot G$ on our curve

If I know both P and G, why can I not reconstruct *e*?

Elliptic curve in Bitcoin

Keys

$y^2 = x^3 + 7$

n = 0xfffffffffffffffffffffffffffffffebaaedce6af48a03bbfd25e8cd0364141

$G = (G_x, G_y)$

Discrete log of size **n!!!**

Private key: a number *e* of 256 bits (in reality from [0,n])

Public key: the point $P = e \cdot G$ on our curve

If I know both P and G, why can I not reconstruct *e*?

Elliptic curve in Bitcoin

Keys

$y^2 = x^3 + 7$

n = 0xfffffffffffffffffffffffffffffffebaaedce6af48a03bbfd25e8cd0364141

G = (G_x,G_y)

Private key: a number *e* of 256 bits (in reality from [0,n])
Public key: the point P = e · G on our curve

How do we gerate *e*?

1) e = randint(0,n) ← but with a real source of randomness, not the one given by Python!
2) e = sha256(b'my secret # 7893') ← "brain wallet"
3) e = sha256(sha256(b'my secret # 7893') + b'salt') ← "brain wallet with salt"

Elliptic curve in Bitcoin

How do I sign my document?

Private key: a number e of 256 bits
Public key: the point $P = e \cdot G$ on our curve
Document: d → I will sign $z = hash256(d)$

1. Select a random k (but really random) in [0,n]
2. Compute the point $F = k \cdot G = (F_x, F_y)$
3. Let $r = F_x \pmod{n}$
4. If r == 0 goto 1.
5. Let $s = k^{-1}(z + r \cdot e)$
6. If s == 0 goto 1.

My signature is (r,s)

Elliptic curve in Bitcoin

How do I sign my document?

Private key: a number e of 256 bits
Public key: the point $P = e \cdot G$ on our curve
Document: d → I will sign $z = hash256(d)$

Everything is calculated in F_n

1. Select a random k (but really random) in [0,n]
2. Compute the point $F = k \cdot G = (F_x, F_y)$
3. Let $r = F_x \ (mod\ n)$
4. If r == 0 goto 1.
5. Let $s = k^{-1}(z + r \cdot e)$
6. If s == 0 goto 1.

My signature is (r,s)

Elliptic curve in Bitcoin

How do I sign my document?

Private key: a number e of 256 bits
Public key: the point $P = e \cdot G$ on our curve
Document: d → I will sign $z = hash256(d)$

1. Select a random k (but really random) in [0,n]
2. Compute the point $F = k \cdot G = (F_x, F_y)$
3. Let $r = F_x \ (mod\ n)$
4. If r == 0 goto 1.
5. Let $s = k^{-1}(z + r \cdot e)$
6. If s == 0 goto 1.

My signature is (r,s)

Everything in F_n
(n is prime)

$k^{-1} = k^{(n-2)}$ % n (Fermat)
$s = k^{-1}(z+re)$ % n

Elliptic curve in Bitcoin

How do I verify a signature?

I receive: The signature (r,s)
Public key: the point $P = e \cdot G$ on our curve
Document: d → I sign $z = hash256(d)$

1. Calculate $u = s^{-1}z$
2. Calculate $v = s^{-1}r$
3. Calculate $C = u \cdot G + v \cdot P = (C_x, C_y)$

Return $C_x == r$

Elliptic curve in Bitcoin

How do I verify a signature?

I receive: The signature (r,s)
Public key: the point $P = e \cdot G$ on our curve
Document: d → I sign $z = hash256(d)$

1. Calculate $u = s^{-1}z$
2. Calculate $v = s^{-1}r$
3. Calculate $C = u \cdot G + v \cdot P = (C_x, C_y)$

Return $C_x == r$

Everything in F_n
(n is prime)

Elliptic curve in Bitcoin

How do I verify a signature?

I receive: The signature (r,s)
Public key: the point $P = e \cdot G$ on our curve
Document: d → I sign $z = hash256(d)$

1. Calculate $u = s^{-1}z$
2. Calculate $v = s^{-1}r$
3. Calculate $C = u \cdot G + v \cdot P = (C_x, C_y)$

Return $C_x == r$

??????????????

Elliptic curve in Bitcoin

Why does this work?

Public key: the point $P = e \cdot G$ on our curve
Document: d → I sign $z = hash256(d)$
Signature: (r,s); $r = F_x$; $F = k \cdot G$, $s = k^{-1}(z + r \cdot e)$
Verification: $u = s^{-1}z$, $v = s^{-1}r$, $C = u \cdot G + v \cdot P = (C_x, C_y)$

$$
\begin{aligned}
C &= u \cdot G + v \cdot P \\
&= u \cdot G + ve \cdot G \\
&= (u + ve) \cdot G \\
&= (s^{-1}z + s^{-1}re) \cdot G \\
&= s^{-1}(z + re) \cdot G
\end{aligned}
$$

Elliptic curve in Bitcoin

Why does this work?

Public key: the point $P = e \cdot G$ on our curve
Document: d → I sign $z = hash256(d)$
Signature: (r,s); $r = F_x$; $F = k \cdot G$, $\boxed{s = k^{-1}(z + r \cdot e)}$ $k = s^{-1}(z + re)$
Verification: $u = s^{-1}z$, $v = s^{-1}r$, $C = u \cdot G + v \cdot P = (C_x, C_y)$

$$\begin{aligned} C &= u \cdot G + v \cdot P \\ &= u \cdot G + ve \cdot G \\ &= (u + ve) \cdot G \\ &= (s^{-1}z + s^{-1}re) \cdot G \\ &= s^{-1}(z + re) \cdot G \end{aligned}$$

Elliptic curve in Bitcoin

Why does this work?

Public key: the point $P = e \cdot G$ on our curve
Document: d → I sign $z = hash256(d)$
Signature: (r,s); $r = F_x$; $F = k \cdot G$, $\boxed{s = k^{-1}(z + r \cdot e)}$ → $k = s^{-1}(z + re)$
Verification: $u = s^{-1}z$, $v = s^{-1}r$, $C = u \cdot G + v \cdot P = (C_x, C_y)$

$$
\begin{aligned}
C &= u \cdot G + v \cdot P \\
&= u \cdot G + ve \cdot G \\
&= (u + ve) \cdot G \\
&= (s^{-1}z + s^{-1}re) \cdot G \\
&= s^{-1}(z + re) \cdot G \\
&= k \cdot G \\
&= F
\end{aligned}
$$

Elliptic curve in Bitcoin

How do I sign my document?

Private key: a number *e* of 256 bits
Public key: the point $P = e \cdot G$ on our curve
Document: d → I will sign $z = hash256(d)$

1. Select a **random** *k* (but really random) in [0,n]
2. Compute the point $F = k \cdot G = (F_x, F_y)$
3. Let $r = F_x \pmod n$
4. If r == 0 goto 1.
5. Let $s = k^{-1}(z + r \cdot e)$
6. If s == 0 goto 1.

My signature is (r,s)

Elliptic curve in Bitcoin

Why does this work?

Public key: the point $P = e \cdot G$ on our curve
Document: d_1, d_2 → I sign $z_1 = hash256(d_1)$, $z_2 = hash256(d_2)$
Signature: (r,s); $r = F_x$; $F = k \cdot G$, $s = k^{-1}(z + r \cdot e)$

$kG = (r,y)$

$s_1 = k^{-1}(z_1 + r \cdot e)$, $s_2 = k^{-1}(z_2 + r \cdot e)$

$s_1/s_2 = (z_1 + r \cdot e)/(z_2 + r \cdot e)$

$s_1(z_2 + r \cdot e) = s_2(z_1 + r \cdot e)$

$s_1 z_2 + s_1 re = s_2 z_1 + s_2 re$

$s_1 re - s_2 re = s_2 z_1 - s_1 z_2$

$e = (s_2 z_1 - s_1 z_2)/(rs_1 - rs_2)$

Elliptic curve in Bitcoin

Why does this work?

Public key: the point $P = e \cdot G$ on our curve
Document: d_1, d_2 → I sign $z_1 = hash256(d_1)$, $z_2 = hash256(d_2)$
Signature: (r,s); $r = F_x$; $F = k \cdot G$, $s = k^{-1}(z + r \cdot e)$

$kG = (r,y)$

$s_1 = k^{-1}(z_1 + r \cdot e)$, $s_2 = k^{-1}(z_2 + r \cdot e)$

$s_1/s_2 = (z_1 + r \cdot e)/(z_2 + r \cdot e)$

$s_1(z_2 + r \cdot e) = s_2(z_1 + r \cdot e)$

$s_1z_2 + s_1re = s_2z_1 + s_2re$

$s_1re - s_2re = s_2z_1 - s_1z_2$

$e = (s_2z_1 - s_1z_2)/(rs_1 - rs_2)$

BAM!

I have your Private key
☹

Elliptic curve in Bitcoin

Why does this work?

Public key: the point $P = e \cdot G$ on our curve
Document: d_1, d_2 → I sign $z_1 = hash256(d_1), z_2 = hash256(d_2)$
Signature: (r,s); $r = F_x$; $F = k \cdot G$, $s = k^{-1}(z + r \cdot e)$

$kG = (r,y)$
$s_1 = k^{-1}(z_1 + r \cdot e)$, $s_2 = k^{-1}(z_2 + r \cdot e)$
$s_1/s_2 = (z_1 + r \cdot e)/(z_2 + r \cdot e)$
$s_1(z_2 + r \cdot e) = s_2(z_1 + r \cdot e)$
$s_1z_2 + s_1re = s_2z_1 + s_2re$
$s_1re - s_2re = s_2z_1 - s_1z_2$
$e = (s_2z_1 - s_1z_2)/(rs_1 - rs_2)$

PS3 hack

BAM!

I have your Private key
☹

Serialization

Object that are transferred over the network:

1. Public key
2. The signature
3. A Bitcoin address (wait for it)

All of our objects need to be transferred in a certain format:

- Our implementation of ECC, Keys, and signatures, are going to be Python objects
- This is not something we can pass onto another person not using Python

That is: we have to serialize **Public key** and the **signature**

The SEC format

Standards for Efficient Cryptography

There is a standard for key serialization in ECDSA: SEC

SEC allows two types of serialization:
1. Uncompressed SEC format
2. Compressed SEC format

The SEC format

Uncompressed SEC format

P = (x,y) is a Public key

x, y are numbers of 256 bits = 32 bytes (1 byte = 8 bits = 2 hex numbers)

Uncompressed SEC serialization of P =(x,y):

1. prefix = 0x04
2. x_{SEC} = x in 32 bytes big-endian int
3. y_{SEC} = y in 32 bytes big-endian int

Return prefix + x_{SEC} + y_{SEC} (+ is a concatenation of bytes)

The SEC format

Uncompressed SEC format

P = (x,y) is a Public key

x, y are numbers of 256 bits = 32 bytes (1 byte = 8 bits = 2 hex numbers)

Uncompressed SEC serialization of P =(x,y):

1. prefix = 0x04
2. x_{SEC} = x in 32 bytes big-endian int
3. y_{SEC} = y in 32 bytes big-endian int

```
047211a824f55b505228e4c3d5194c1fcfaa15a456abdf37f9b9d97a4040afc073dee6c8906498
4f03385237d92167c13e236446b417ab79a0fcae412ae3316b77

        - 04 - Marker
        - x coordinate - 32 bytes
        - y coordinate - 32 bytes
```

The SEC format

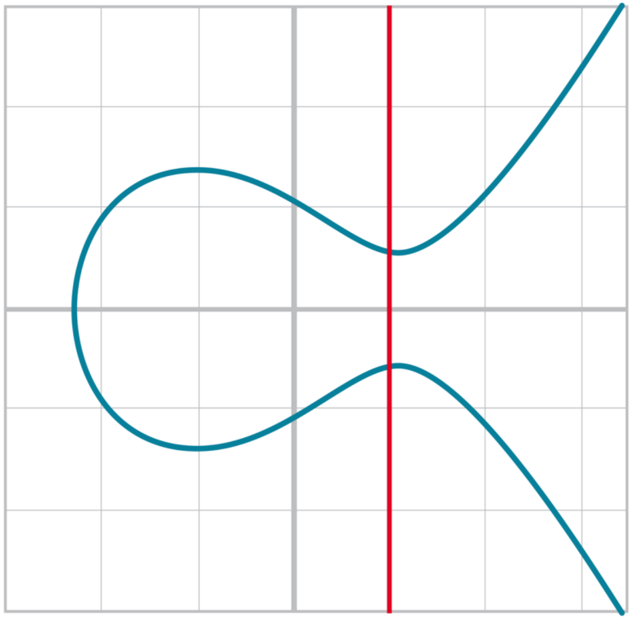
Compressed SEC format

P = (x,y) is a Public key, and lives on the curve $y^2 = x^3 + ax + b$

If I know x, there are only two candidates for y!!!

(x,y) , (x,-y)

To encode this I need: x + 1 bit to signal if we will use y or -y



The SEC format

Compressed SEC format

P = (x,y) is a Public key, and lives on the curve $y^2 = x^3 + ax + b$

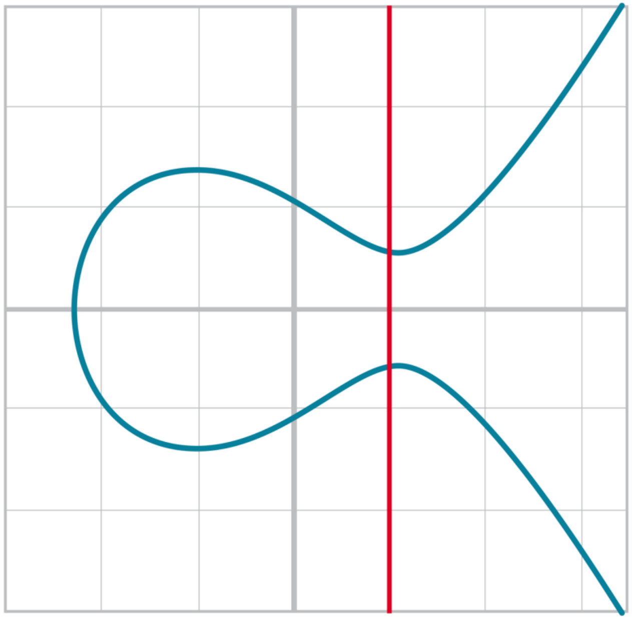
If I have x → (x,y), (x,-y) are valid candidates

-y = (p-y) % p, since we are working over F_p

If I have x → (x,y), (x,p-y) are valid candidates

p is a prime number greater than 2

y even → p-y odd
y odd → p-y even



The SEC format

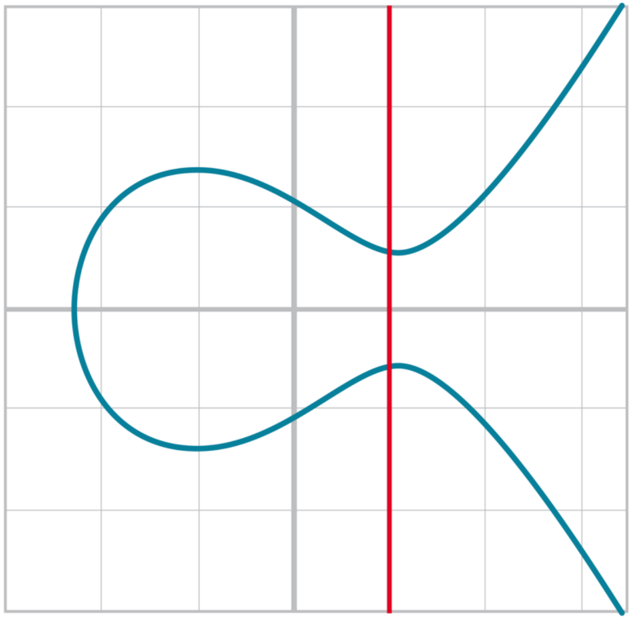
Compressed SEC format

P = (x,y) is a Public key, and lives on the curve $y^2 = x^3 + ax + b$

If I have x → (x,y), (x,-y) are valid candidates

y even → p-y odd
y odd → p-y even

Compressed SEC format:
- x
- Parity of y (even or odd)



The SEC format

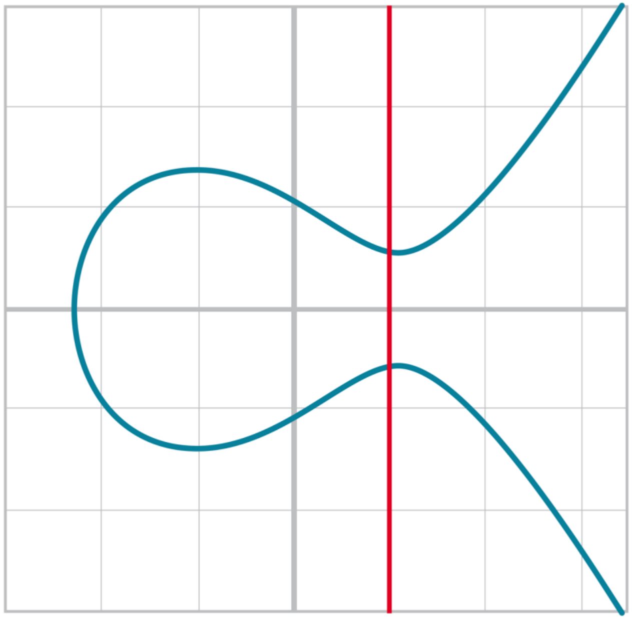
Compressed SEC format

P = (x,y) is a Public key, and lives on the curve $y^2 = x^3 + ax + b$

Compressed SEC format:

1. prefix = 0x02 if y is even, 0x03 if not (1 byte)
2. x_{SEC} = x in 32 bytes big-endian

Return prefix + x_{SEC} (33 bytes)



The SEC format

Compressed SEC format

P = (x,y) is a Public key, and lives on the curve $y^2 = x^3 + ax + b$

Compressed SEC format:

1. prefix = 0x02 if y is even, 0x03 if not (1 byte)
2. x_{SEC} = x in 32 bytes big-endian

Return prefix + x_{SEC} (33 bytes)

```
0349fc4e631e3624a545de3f89f5d8684c7b8138bd94bdd531d2e213bf016b278a

- 02 if y is even, 03 if odd - Marker
- x coordinate - 32 bytes
```

The SEC format

Compressed SEC format

P = (x,y) is a Public key, and lives on the curve $y^2 = x^3 + ax + b$

I have x and the parity of y: How do I compute y in F_p?

We are solving the equation $w^2 = v$ in the finite field F_p

The trick: let us assume that p % 4 = 3 (as is the case in Bitcoin)

p % 4 = 3 → (p+1) % 4 = 0 → (p+1)/4 is a natural number

The SEC format

Compressed SEC format

We are solving the equation w^2 = v in the finite field F_p

p % 4 = 3 → (p+1) % 4 = 0 → (p+1)/4 is a natural number

Little Fermat's Theorem: w^{p-1} % p = 1

$w^2 = w^2 \cdot 1 = w^2 \cdot w^{p-1} = w^{p+1}$ → p is prime > 2 → p+1 is even

$$w = w^{(p+1)/2} = w^{2(p+1)/4} = (w^2)^{(p+1)/4} = v^{(p+1)/4}$$

w^2 = v

The SEC format

Compressed SEC format

P = (x,y) is a Public key, and lives on the curve $y^2 = x^3 + ax + b$

Compressed SEC format:

1. prefix = 0x02 if y is even, 0x03 if not (1 byte)
2. x_{SEC} = x en 32 bytes big-endian

$y_{SEC} = {x_{SEC}}^{(p+1)/4}$

If prefix == 0x02 AND y_{SEC} % 2 == 0 → P = (x_{SEC},y_{SEC}), else P = (x_{SEC},p - y_{SEC})

If prefix == 0x03 AND y_{SEC} % 2 == 0 → P = (x_{SEC},p - y_{SEC}), else P = (x_{SEC},y_{SEC})

The DER format

Digital Encoding Rules

To send signatures we will use DER (again, it's Satoshi's fault; he used OpenSSH en 2008):

signature = *(r,s), r and s* are of 32 bytes; this can not becompressed

1. *prefix = 0x30*
2. *len = length* of the remainder of the signature (in hex; usually 0x44, o 0x45)
3. *marker = 0x02*
4. *r i*n big-endian, if *r[0] ≥ 0x80 r = 0x00 + r* (concatenation of bytes)
5. marker = 0x02
6. s in big-endian, if *s[0] ≥ 0x80 s = 0x00 + s*

Return prefix + len + marker + len(r) + r + marker + len(s) + s

The DER format

Digital Encoding Rules

To send signatures we will use DER (again, it's Satoshi's fault; he used OpenSSH en 2008):

signature = *(r,s), r and s* are of 32 bytes; this can not becompressed

1. *prefix = 0x30*
2. *len = length* of the remainder of the signature (in hex; usually 0x[illegible] o 0x45)
3. *marker = 0x02*
4. *r* in big-endian, if *r[0] ≥ 0x80 r = 0x00 + r* (concatenation of bytes)
5. marker = 0x02
6. s in big-endian, if *s[0] ≥ 0x80 s = 0x00 + s*

Return prefix + len + marker + len(r) + r + marker + len(s) + s

What is this?

The DER format

Digital Encoding Rules

To send signatures we will use DER (again, it's Satoshi's fault; he used OpenSSH en 2008):

signature = *(r,s), r and s* are of 32 bytes; this can not becompressed

1. *prefix = 0x30*
2. *len = length* of the remainder of the signature (in hex; usually 0 …
3. *marker = 0x02*
4. *r* in big-endian, if *r[0] ≥ 0x80 r = 0x00 + r* (concate… ation of bytes)
5. marker = 0x02
6. s in big-endian, if *s[0] ≥ 0x80 s = 0x00 + s*

Return prefix + len + marker + len(r) + r + marker + le…

What is this?

DER allows negative numbers (byte tero tells us if positive)

The DER format

Digital Encoding Rules

3045022100ed81ff192e75a3fd2304004dcadb746fa5e24c5031ccfcf213
20b0277457c98f02207a986d955c6e0cb35d446a89d3f56100f4d7f67801
c31967743a9c8e10615bed

- 30 - Marker
- 45 - Length of sig
- 02 - Marker for r value
- 21 - r value lenth
- 00ed...8f - r value
- 02 - Marker for s value
- 20 - s value length
- 7a98...ed - s value

In hex!!!

Bitcoin addresses

A key in SEC (both compressed and not) can be shortened (from 33/64 to 20 bytes):

- We will shorten it and make it more secure using *ripemd160*
- *ripemd160* is a hash function with 160 bit output

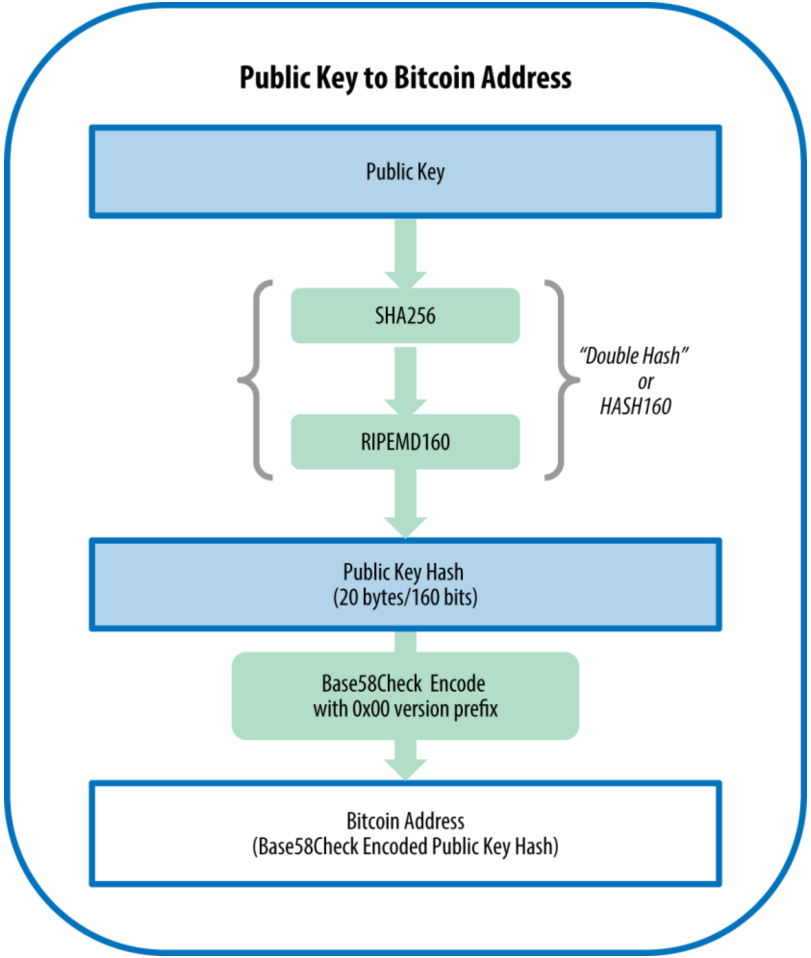
A key is easier to read/process/store if it is shorter:

- We will shorten the representation using base 58
- [0-9] + [a-z] + [A-Z]
- Excluding: 0/O, I/l (zero, capital O, capital I, lowe case l)

How can we define a Bitcoin address with this?

A Bitcoin address

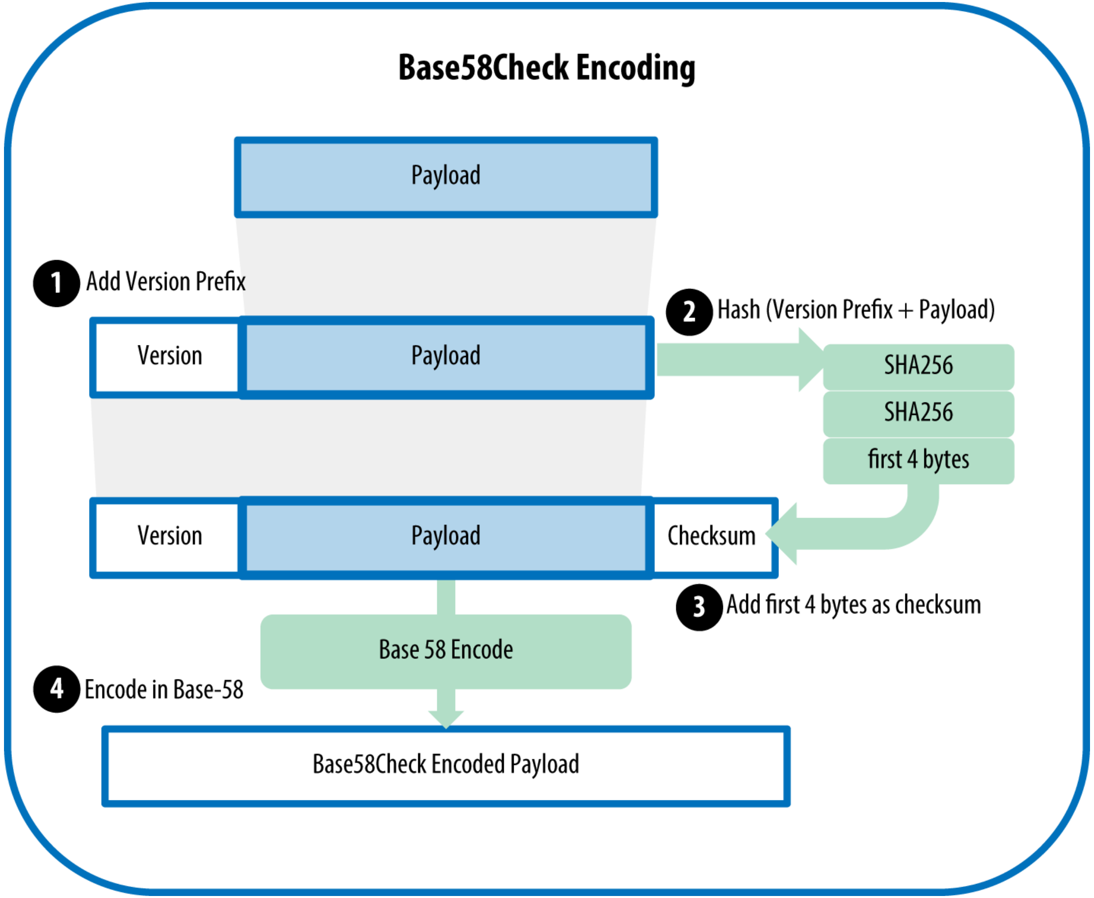
Is not a public key



https://github.com/bitcoinbook/bitcoinbook/blob/develop/ch04.asciidoc

A Bitcoin address

Is not a public key



https://github.com/bitcoinbook/bitcoinbook/blob/develop/ch04.asciidoc

Bitcoin addresses

P a public key in SEC format

How to create a Bitcoin address from this?

1. *version = 0x00* for mainnet, and *0x6f* for testnet
2. *key = ripemd160(sha256(P))*
3. *checksum = sha256(sha256(version + key))*[:4]
4. *res = version + key + checksum*

Return encode_base58(res)

Bitcoin addresses

How will this look like?

```
Private key: 0x69b0392170b2b4809788bd619997ed88371ddd6d6e3fa0524d1eb49325498936
Testnet address: n1VietsybSPTN3wuAWuXvUDtTc7D4GGVDj
Mainnet address: 1LymMqnznQxCawUHSwwA6Z1ZbcWW4Nxb1a
```

Keys in Bitcoin

A comment on base58

```python
# the alhabet we use
BASE58_ALPHABET = '123456789ABCDEFGHJKLMNPQRSTUVWXYZabcdefghijkmnopqrstuvwxyz'

# compute base58 encoding of s
def encode_base58(s):
    count = 0
    for c in s:  # count the number of zeros at the front
        if c == 0:
            count += 1
        else:
            break
    num = int.from_bytes(s, 'big')
    prefix = '1' * count
    result = ''
    while num > 0:  # figure out which symbol to use
        num, mod = divmod(num, 58)
        result = BASE58_ALPHABET[mod] + result
    return prefix + result  # prepend the zeros that conversion deleted
```

Keys in Bitcoin

Base58 is on the way out

These days one mostly uses Bench32 (introduced with segwit – BIP 0173)

References

Reading:

- Jimmy Song, Programming Bitcoin, chapters 1—4
- https://andrea.corbellini.name/2015/05/30/elliptic-curve-cryptography-ecdh-and-ecdsa/

All images are from:
https://github.com/jimmysong/programmingbitcoin/blob/master
https://github.com/bitcoinbook/bitcoinbook/blob/develop/ch04.asciidoc