ECC (Part II)
&
Smart Contracts

Sep. 16, 2019

Overview

- Cryptography with ECC
  - How to send secret messages

- Bitcoin’s “Stack Machine” scripts
  - Review
  - Limitations

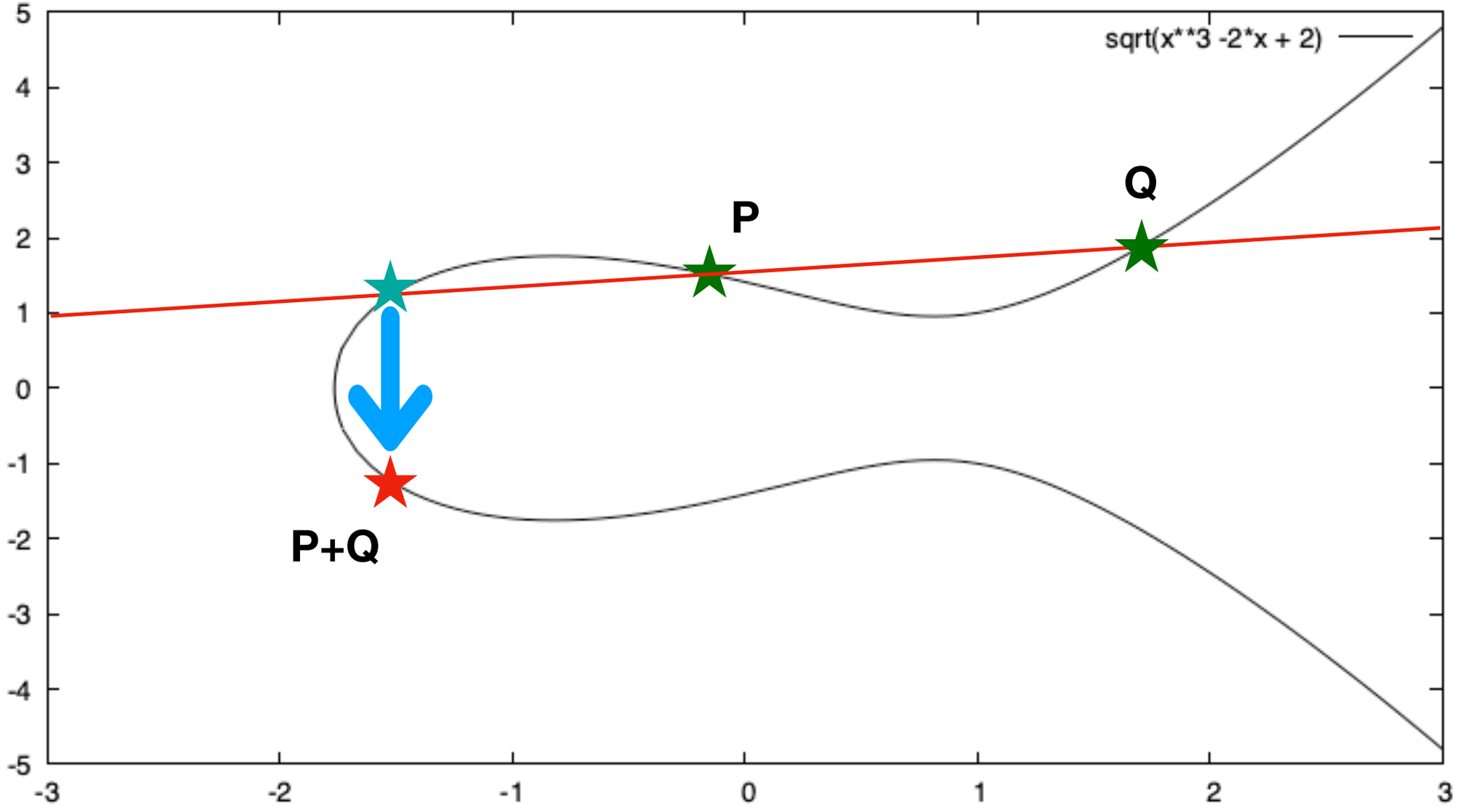
- Ethereum’s answer to those limitations
  - Applications
  - Programming Language “Solidity”
  - Examples
  - Advantages/Disadvantages

Elliptic Curve Cryptography

(Part II)

Points on elliptic curve

- Points can be added $P + Q$
- Points can be added n times nP
- Points can be subtracted $P - Q$



Points on elliptic curves

- The entire math of ECC is based on adding points
  - A point G can be added to itself, the new point is $H = 2G$
  - Added k times to itself results in point $F = kG$
  - Points can be added very fast

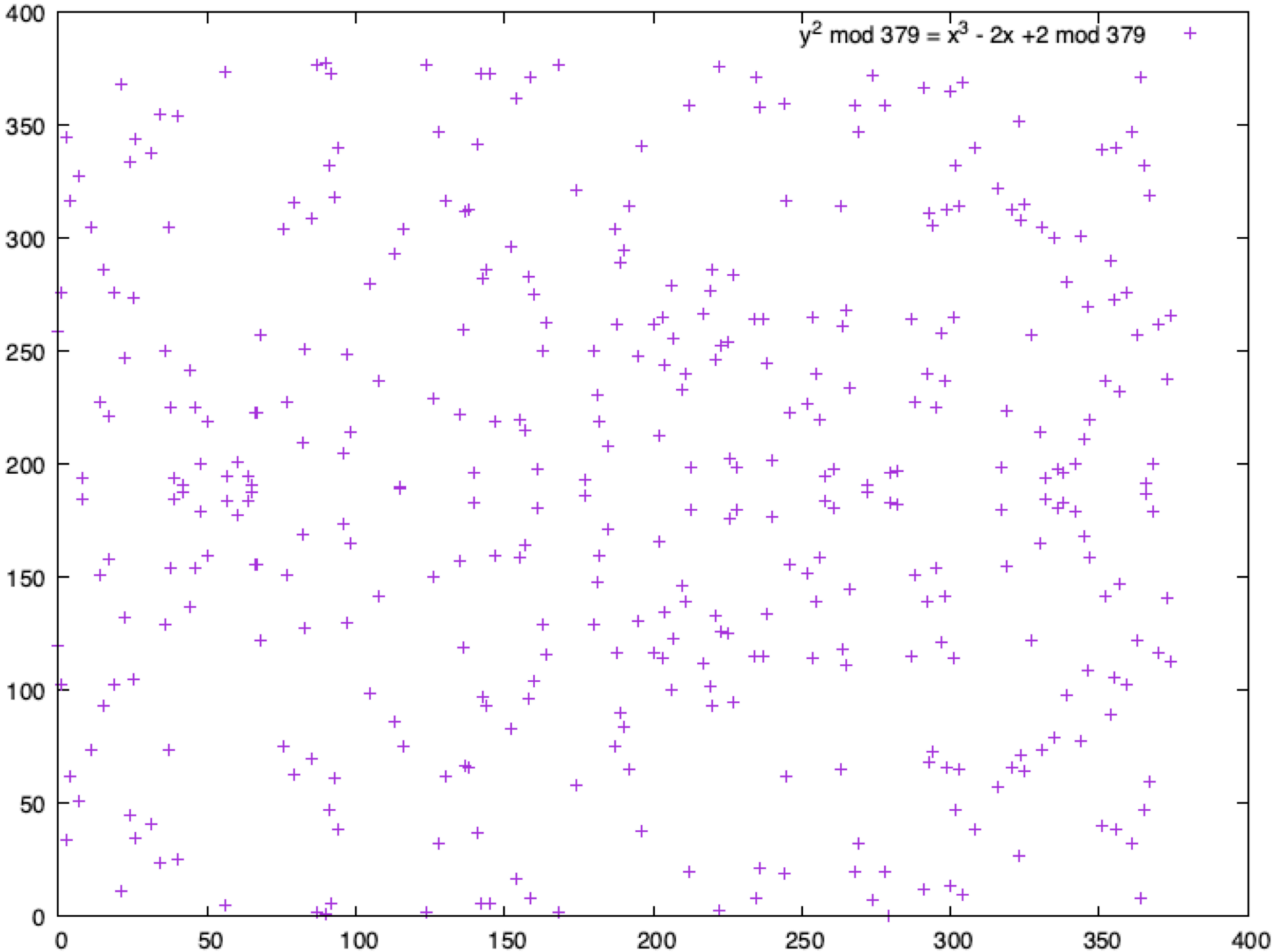
- As a side node, for ECC-based cryptography, everything ‘happens’ $\mod n$
  - A point is on a curve iff

$$y^2 \mod n = x^3 + ax + b \mod n$$

  - Point arithmetics are still well defined

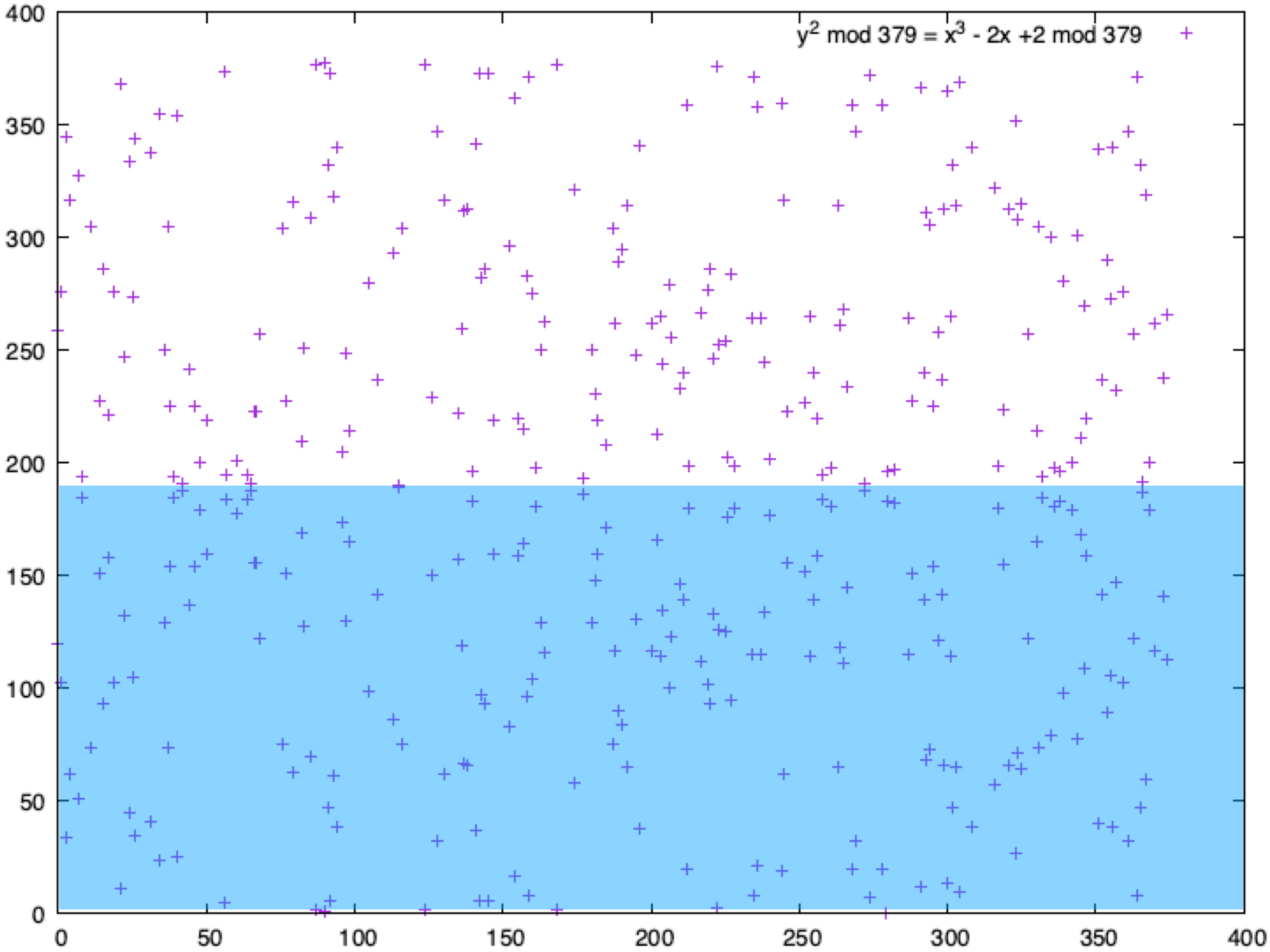
Elliptic curve mod p

$p \in$ PRIMES



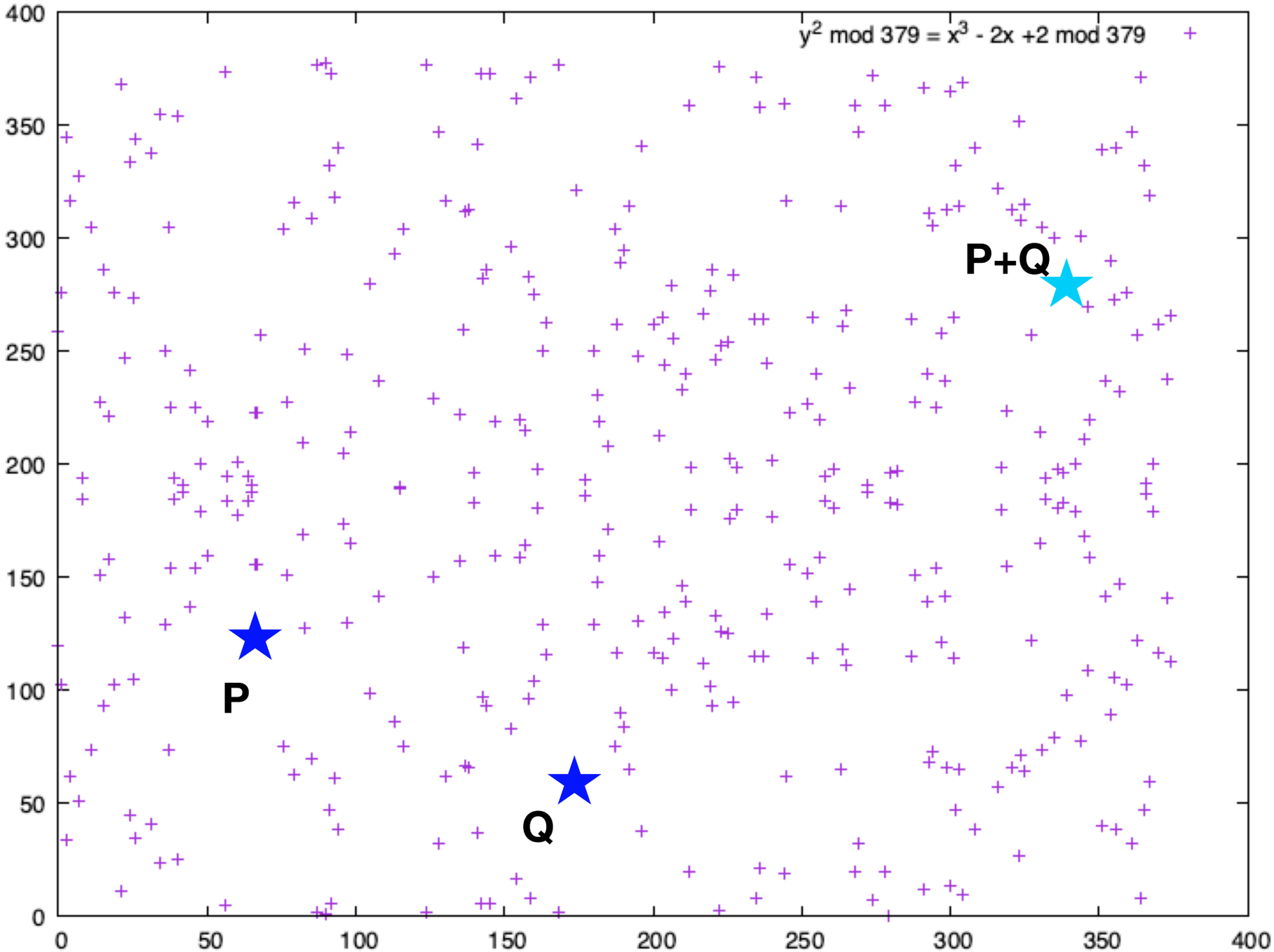
Elliptic curve mod p

$p \in$ PRIMES

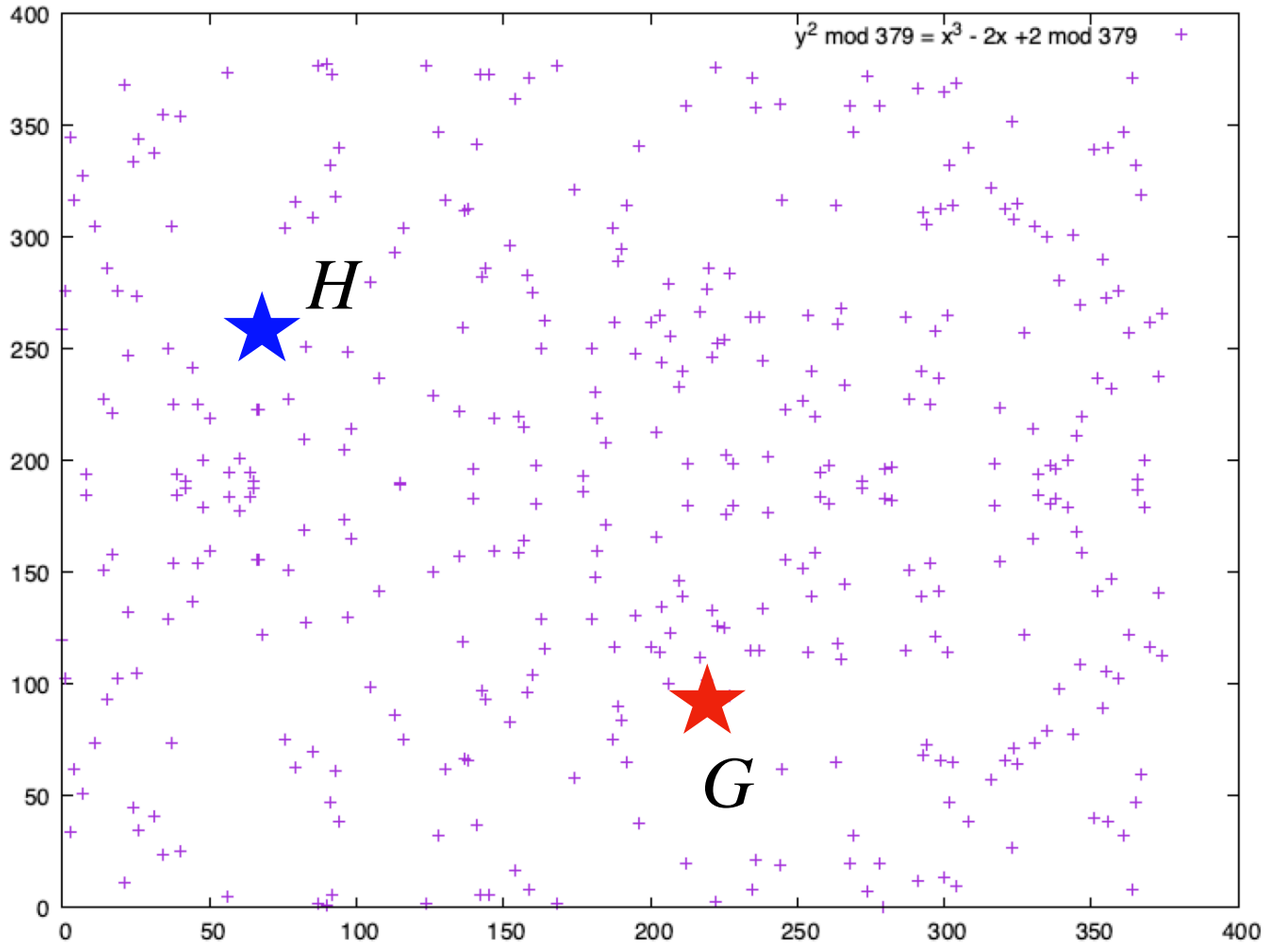


Elliptic curve mod p

$p \in$ PRIMES

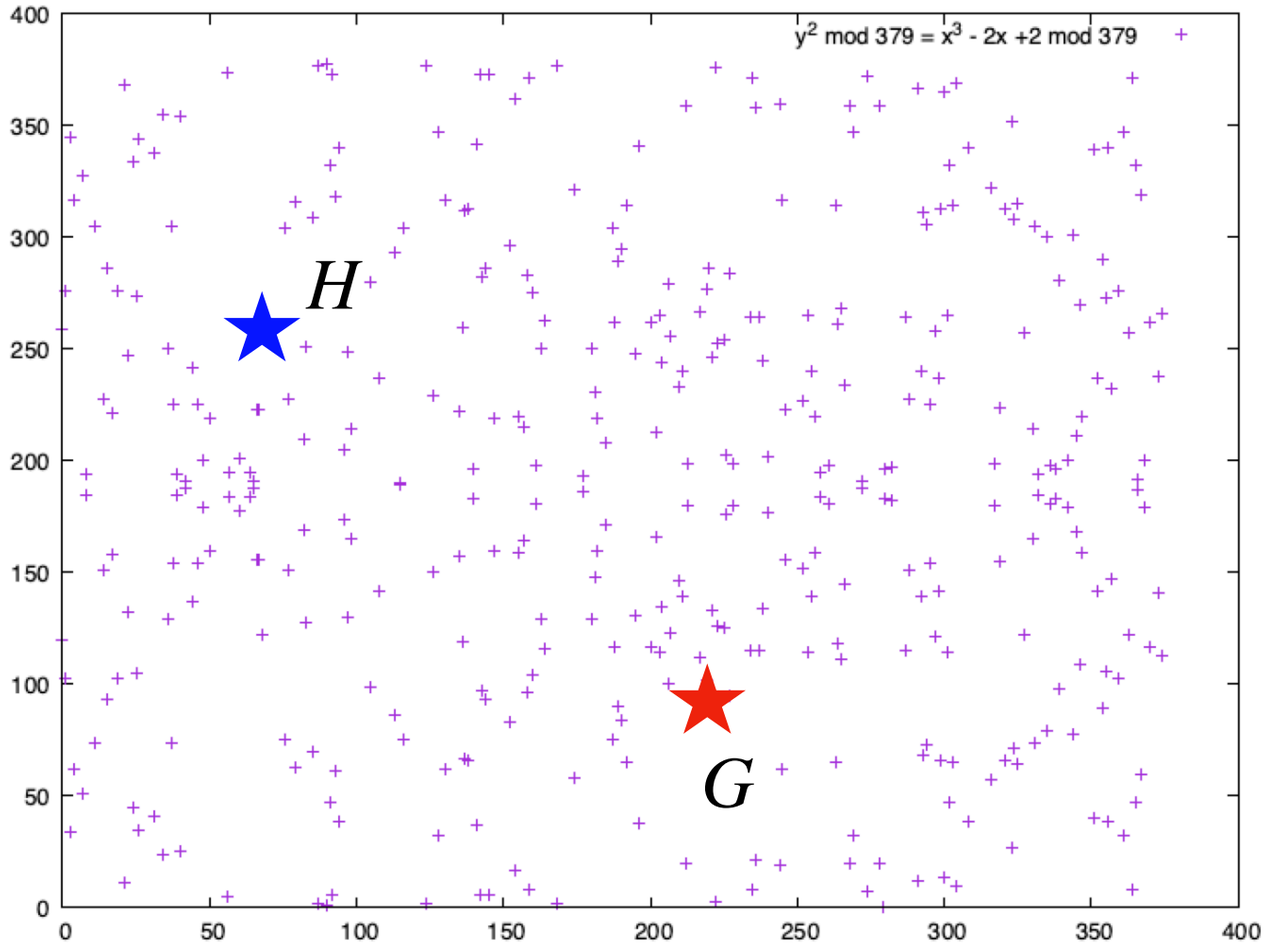


Inverse addition problem



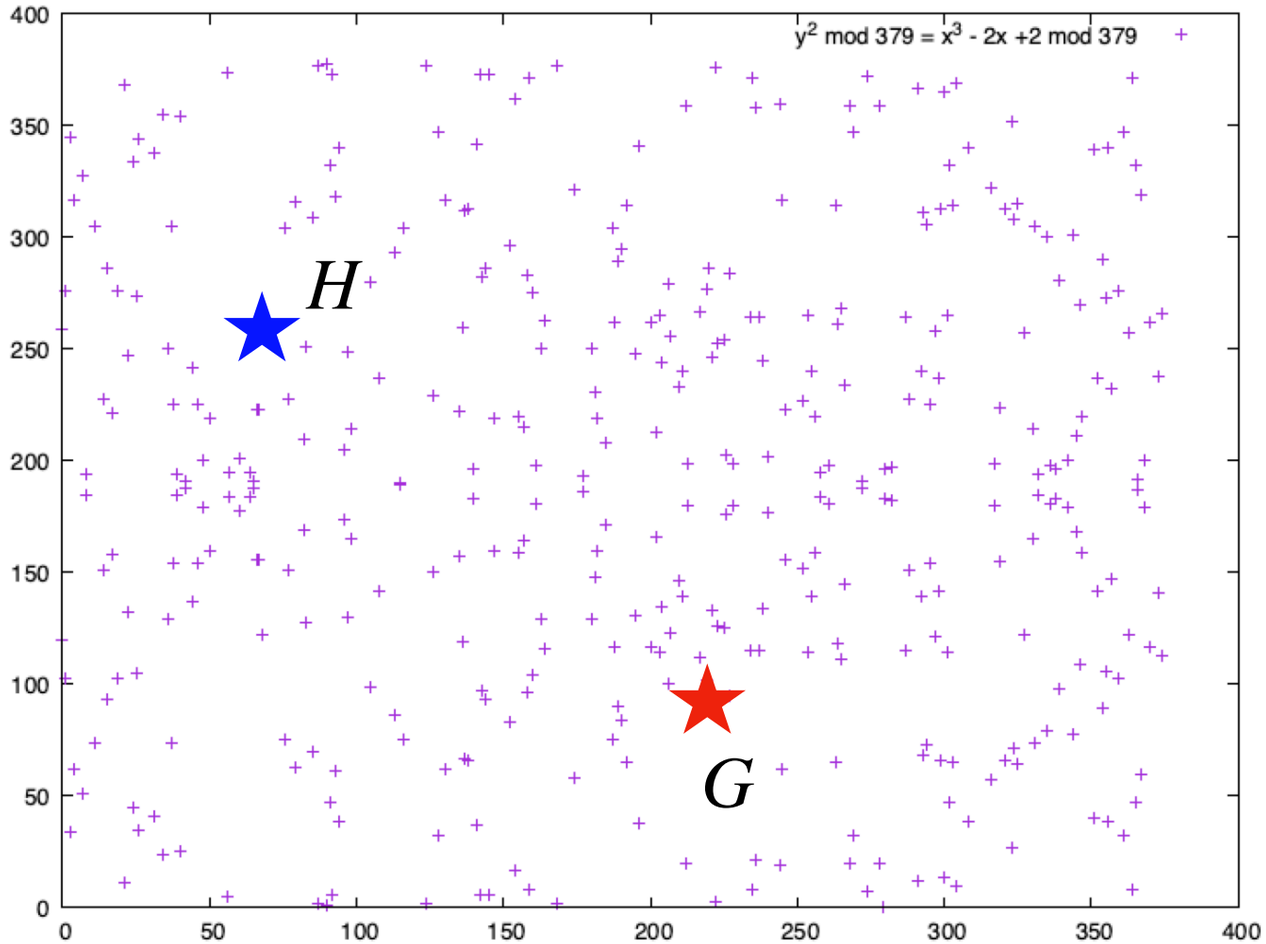
- Given two points, G, H, find $a \in \mathbb{Z}_p$ so that $H = aG$
- How often to I need to add $p = (x, y)$ itself to reach q?

Inverse addition problem



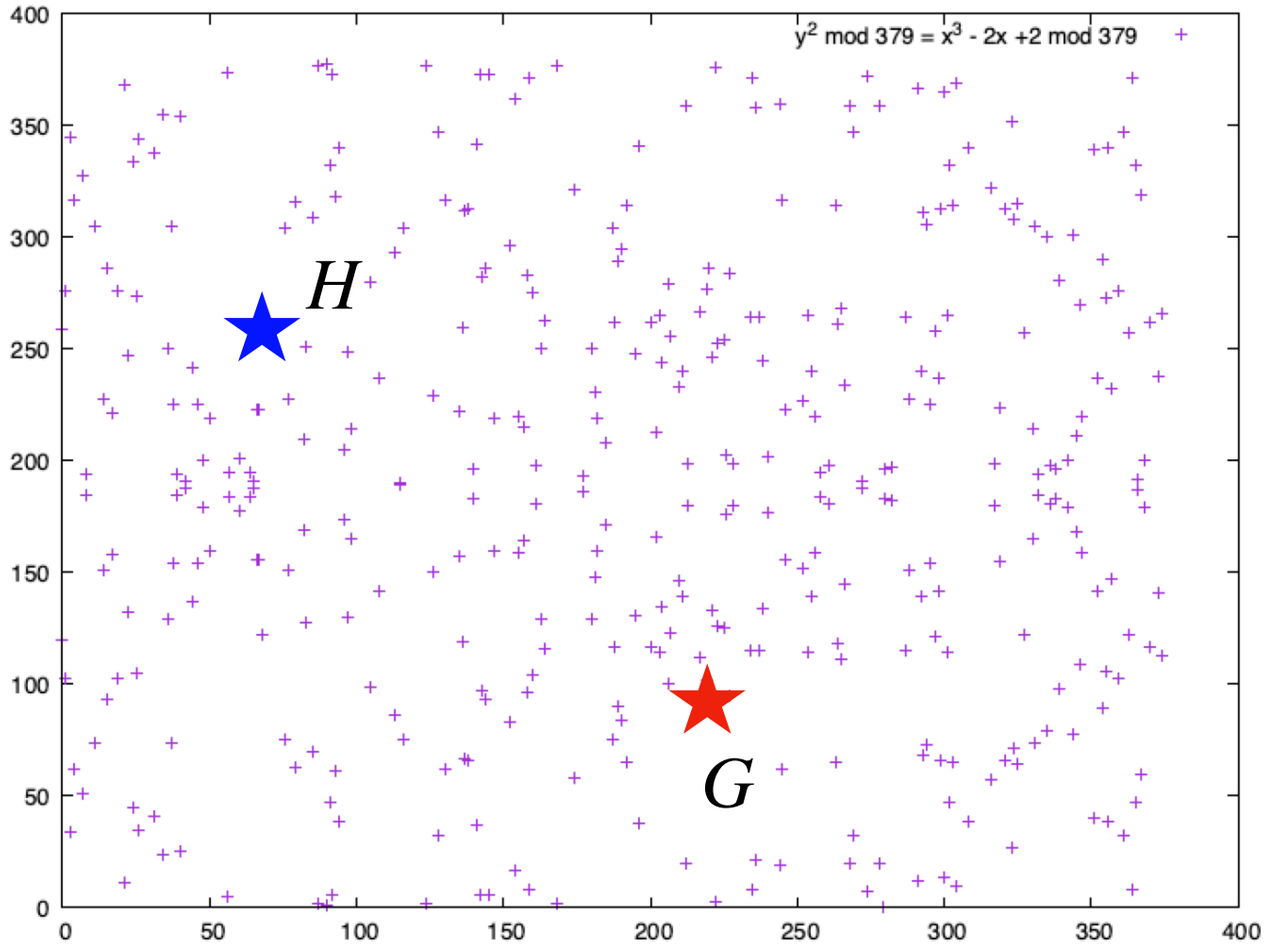
- Super hard, we just can’t do it efficiently

Inverse addition problem



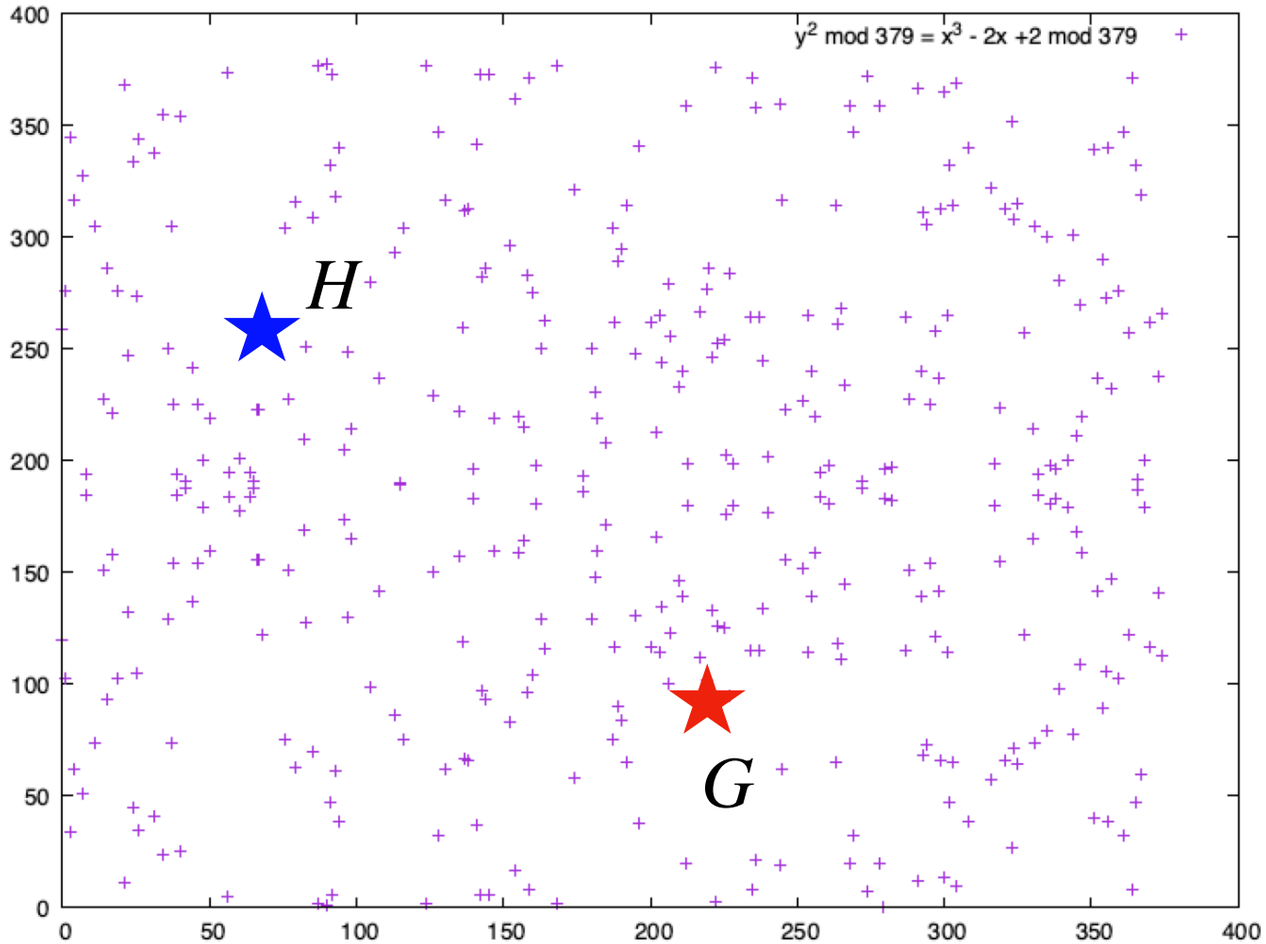
- Super hard, we just can’t do it efficiently
  - NSA supposedly convinced software vendors to use *special* random number generators that made points predictable

Inverse addition problem



- Super hard, we just can't do it efficiently
  - NSA supposedly convinced software vendors to use *special* random number generators that made points predictable
  - Quantum computer can solve this efficiently

Inverse addition problem



- Super hard, we just can’t do it efficiently
  - NSA supposedly convinced software vendors to use *special* random number generators that made points predictable
  - Quantum computer can solve this efficiently
- For now, we this this is a difficult problem

Notation

- We have one operations: combining 2 points
- Before we used the symbol “+” for this
  - $P + Q, H = aG$, etc.

- We can also re-use the multiplication notation $\cdot$
  - $g \cdot h, g^a$

Notation

- ECC math appears in 2 different notations
  - adding points, written as addition and multiplication
  - $P + Q, H = aG$, etc.
    - Usually this goes with the following standard:
      - Numbers: lowercase characters
      - Points: uppercase characters
      - Often in introductory texts, blogs, emails (easy to type)

- multiplying points, written as multiplication and exponentiation
- $g \cdot h, g^a$
  - Usually this goes with the following standard:
    - Everything lowercase
    - exponents (integers) as Greek letters
  - Often in scientific texts
    - Correlation to other fields better visible (cf. $a^b \mod n$)
      - Better readability for complex operations, e.g. $\left(g^a h^\gamma\right)^x$

Notation

"**A foolish consistency is the hobgoblin of little minds,** adored by little statesmen and philosophers and divines. With consistency a great soul has simply nothing to do."

— Ralph Waldo Emerson (poet)

⇒ We will use additive and multiplicative notations in this course

Discrete Logarithm

- In scientific notation, the hard problem is

Find a, so that $y = g^a \mod n$

- called *Discrete Logarithm*

Summary ECC Point Math

- The entire math of ECC is based on adding points
  - A point G can be added to itself, the new point is $H = 2G$
  - Added k times to itself results in point $F = kG$
  - Points can be added very fast

- Inverse problem:
  - Given a base point G and a second point R, it is not possible to efficiently compute integer $r \in \mathbb{Z}_p$ so that $R = rG$

RSA & ECC

$$y = g^a \mod n$$

- Given g and a
  - compute y : easy

- Given y and g
  - compute a : hard

- Given y and a
  - compute g : hard

RSA & ECC

$$y = g^a \mod n$$

- Given g and a
  - compute y : easy

- Given y and g
  - compute a : hard

- Given y and a
  - compute g : hard

One popular curve (Curve25519) has $7.237 \cdot 10^{75}$ points
(7237005577332262213973186563042994240857116359379907606001950938285455250989)

Cryptography with ECC

- Known parameters
  - $a, b \in \mathbb{Z}, n$ for curve
    $\left(y^2 \mod n\right) = \left(x^3 + ax + b \mod n\right)$
    - starting point G on that curve

- Private key
  - random number $n \in \mathbb{N}$ (does not have to be prime)

- Public key
  - Point nG

Cryptography with ECC

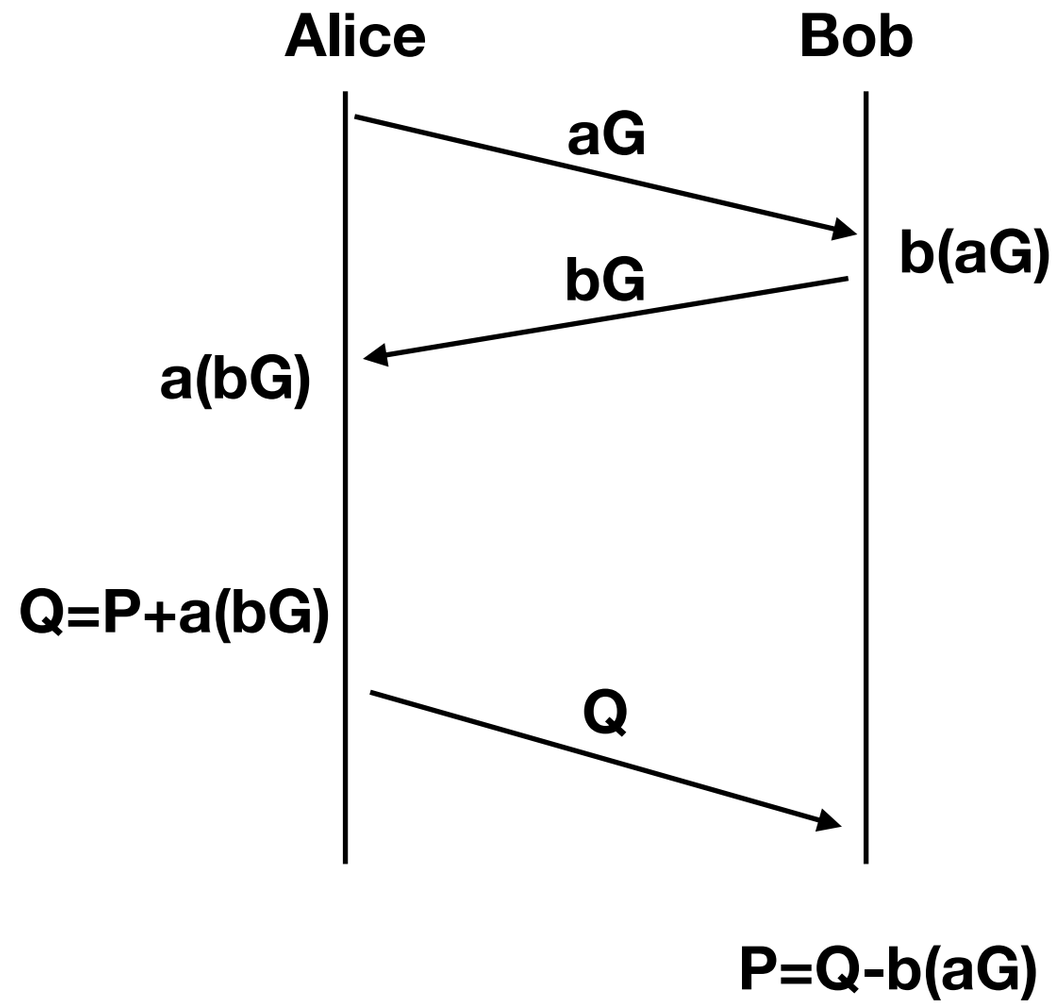
- Diffie-Hellman (DH) Key exchange
- Alice: sk=a, pk=aG
- Bob: sk=b, pk=bG
- exchange of public keys
  - Alice sends aG to Bob
  - Bob sends bG to Alice

Whitfield Diffie and Martin Hellman

- Alice can now compute a(bG)=(ab)G
- Bob can now compute b(aG)=(ba)G
- **Only Alice and Bob know point (ab)G! No one else does**

Cryptography with ECC

- After exchange, Alice and Bob know (ab)G
  - Alice can now send the point Q=P+(ab)G to Bob
- Bob can extract the point P = Q-(ab)G
- Nobody, even when listening to all communication, can compute the point P. All they see is Q



Cryptography with ECC

- Alice can send the point P to Bob
  - P is known to only Alice & Bob

- P is a point (x,y), i.e. all I need to do in to encode my secret message as a point
  - often used DH to share a different key for other cryptography systems
  - One way is to encode message as y coordinate and compute x

Cryptography with ECC Summary

- Points on a curve

- A method to combine points
  - Well defined operation, some use “+", others use “*”
  - Inverse operation (discrete logarithm) is very hard

- Diffie-Hellman key exchange
  - Find a new point, only known to Alice and Bob
  - Usable to send secret messages

- Signatures, Arithmetics, Zero-Knowledge
  - Will be introduces throughout the course

Bitcoin's Stack Machine

- Verify public/private key
  - Given $y=H(pk)$, msg
    - Require *sig, pk*: *verify(pk, msg, sig)=True AND y=H(pk)*
      - Show that you own private key to (the hash of) a public key

Program (scriptPubKey)

1. Duplicate top element
2. Hash top element
3. Push compare value onto stack
4. Verify that the two top elements are equal
5. Verify that (sign., pk, transactionData) is valid

Init state of stack (scriptSig)

Bitcoin's Stack Machine

- Verify preimage of a hash
  - Given y
    - Require x: $y=H(x)$
      - Pay to whoever knows the preimage of a hash

Program (scriptPubKey)

1. Hash top element
2. Push compare value onto stack
3. Verify that the two top elements are equal

Init state of stack (scriptSig)

Bitcoin's Stack Machine

- Require several parties to agree on a transaction
  - Multi-signature
    - Pay if all / 3-out-of-5 / n-out-of-m signatures are present

Program (scriptPubKey)

1. Push value n onto the stack (i.e. 2)
2. Push pubKey1
3. Push pubKey2
4. Push pubKey3
5. …
6. Push value m onto the stack (i.e. 3)
7. Check n-out-of-m multiSig

Init state of stack (scriptSig)

one signature

another signature

Bitcoin's Stack Machine

3-out-of-5 signatures are present

Program (scriptPubKey)

1. Push value *3* onto the stack
2. Push pubKey1
3. Push pubKey2
4. Push pubKey3
5. Push pubKey4
6. Push pubKey5
7. Push value *5* onto the stack
8. Check n-out-of-m multiSig

Init state of stack (scriptSig)

signature E

signature A

signature C

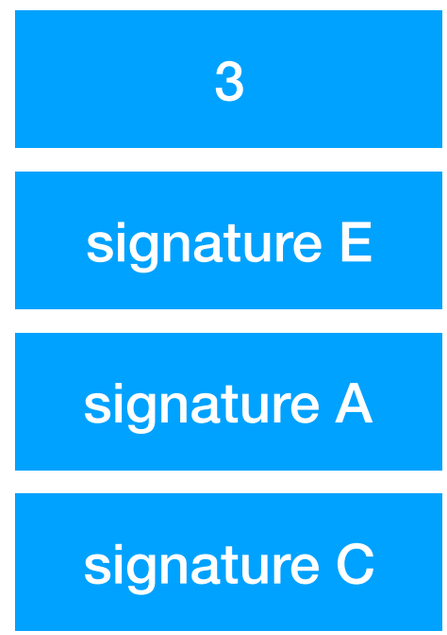
Bitcoin's Stack Machine

3-out-of-5 signatures are present

Program (scriptPubKey)

1. Push value *3* onto the stack
2. Push pubKey1
3. Push pubKey2
4. Push pubKey3
5. Push pubKey4
6. Push pubKey5
7. Push value *5* onto the stack
8. Check n-out-of-m multiSig

Init state of stack (scriptSig)



Bitcoin's Stack Machine

3-out-of-5 signatures are present

Program (scriptPubKey)

1. Push value *3* onto the stack
2. Push pubKey1
3. Push pubKey2
4. Push pubKey3
5. Push pubKey4
6. Push pubKey5
7. Push value *5* onto the stack
8. Check n-out-of-m multiSig

Init state of stack (scriptSig)

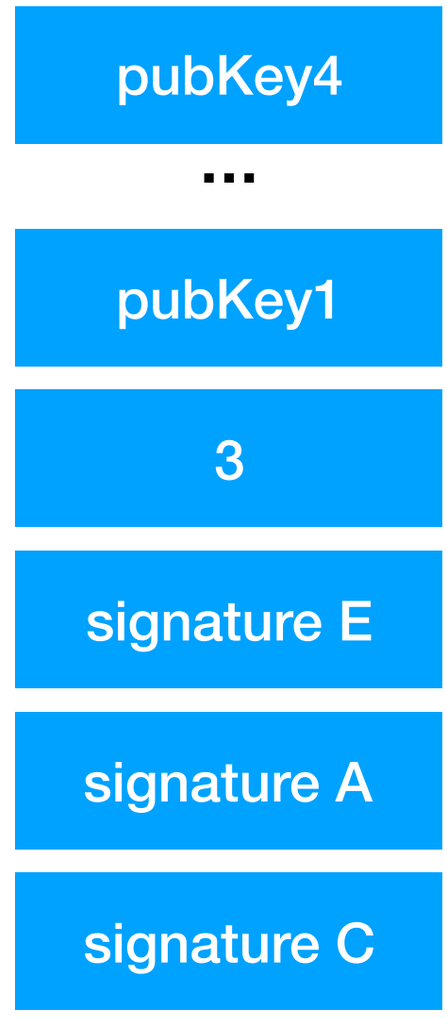
Bitcoin's Stack Machine

3-out-of-5 signatures are present

Program (scriptPubKey)

1. Push value *3* onto the stack
2. Push pubKey1
3. Push pubKey2
4. Push pubKey3
5. Push pubKey4
6. Push pubKey5
7. Push value *5* onto the stack
8. Check *n*-out-of-*m* multiSig

Init state of stack (scriptSig)



Bitcoin's Stack Machine

3-out-of-5 signatures are present

Program (scriptPubKey)

1. Push value *3* onto the stack
2. Push pubKey1
3. Push pubKey2
4. Push pubKey3
5. Push pubKey4
6. Push pubKey5
7. Push value *5* onto the stack
8. Check *n*-out-of-*m* multiSig

Bitcoin's Stack Machine

3-out-of-5 signatures are present

Program (scriptPubKey)

1. Push value *3* onto the stack
2. Push pubKey1
3. Push pubKey2
4. Push pubKey3
5. Push pubKey4
6. Push pubKey5
7. Push value *5* onto the stack
8. Check *n*-out-of-*m* multiSig

`OP_CHECKMULTISIG`:
read `m`
read *m* public keys
read `n`
read *n* signatures
return `TRUE` if all signatures can be verified
(no duplicates)

Bitcoin’s Stack Machine

- What we can do
    - Create scripts that govern who can access the money
    - If it can be implemented in a *Stack Machine*, it can be done in Bitcoin
- Complex scripts could implement a new form of “smart banking”
    - One could write the rules on how an organization can control
    - Bitcoin does not have a physical place in the world (no regulations)
        - Decentralized Autonomous Corporation (DAO)

Decentralized Autonomous Corporation (DAO)

- “Corporations are people, my friend.”

  —Mitt Romney

- What if we encode everything a corporation does/decide into Bitcoin’s scripting language
  - is it possible?
- Can we imagine a corporation without people?
  - Also Mitt Romney: “Everything corporations earn ultimately goes to people. Where do you think it goes?”

Bitcoin’s Stack Machine

- What we can **not** do
  - As it turns out, we are quite limited with this Stack Machine
  - No if-then-else branch
  - No loops
  - No “Turing-completeness”
    - Provably less powerful than a *normal* program
    - Done on purpose to limit the time to validate a block

More powerful scripts

Can we do more?

More powerful scripts

- Let’s re-design Bitcoin to make it more powerful
- Downsides of Bitcoin’s scripting language
  - No Turing-completeness
  - Money can be spend or not, no ‘fine-grained’ access
  - Scripts are always completely executed. No program state
  - Scripts cannot access other Blockchain info (Nonce, etc.)

More powerful scripts

- Obvious solution:
    - Instead of a Stack Machine, allow any computer program to run
    - Expose specific variables “inputs, keys, etc.”
    - Global variables “Nonce, Block-height, other transactions, etc.”
    - Allow halting points

More powerful scripts

- Obvious solution:
  - Instead of a Stack Machine, allow any computer program to run
  - Susceptible to attacks

```
def transactionCode(self):
    done = False
    while(not done):
        pass
```

More powerful scripts

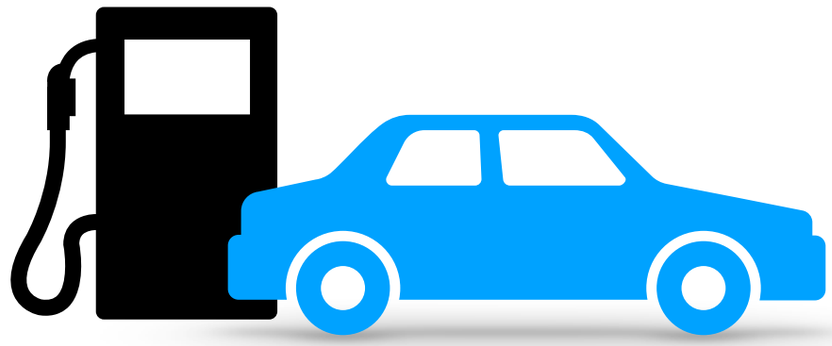
- Obvious solution:
  - Instead of a Stack Machine, allow any computer program to run
    - Susceptible to attacks

```
def transactionCode(self):
    done = False
    while(not done):
        pass
```

- Miner will never finish mining a block
- Miner cannot analyze code to verify if it will eventually stop
  - Halting problem

Limit code run time

- A car runs X miles on a tank of gas



- Introduce a notion of the maximum number of instructions a code can run
    - Provide sufficient *gas*

More powerful scripts

- “Ethereum intends to provide is a blockchain with a built-in fully fledged Turing-complete programming language”

Ethereum

- Just like Bitcoin
  - Blockchain
  - Proof-of-Work
- Not like Bitcoin
  - Allow Turing-complete DAPPs (Distributed Applications)
  - A programming language (Solidity)
  - For each transaction, pay fee in gas
    - 1 Eth = 50 000 000 Gas (currently, may change)

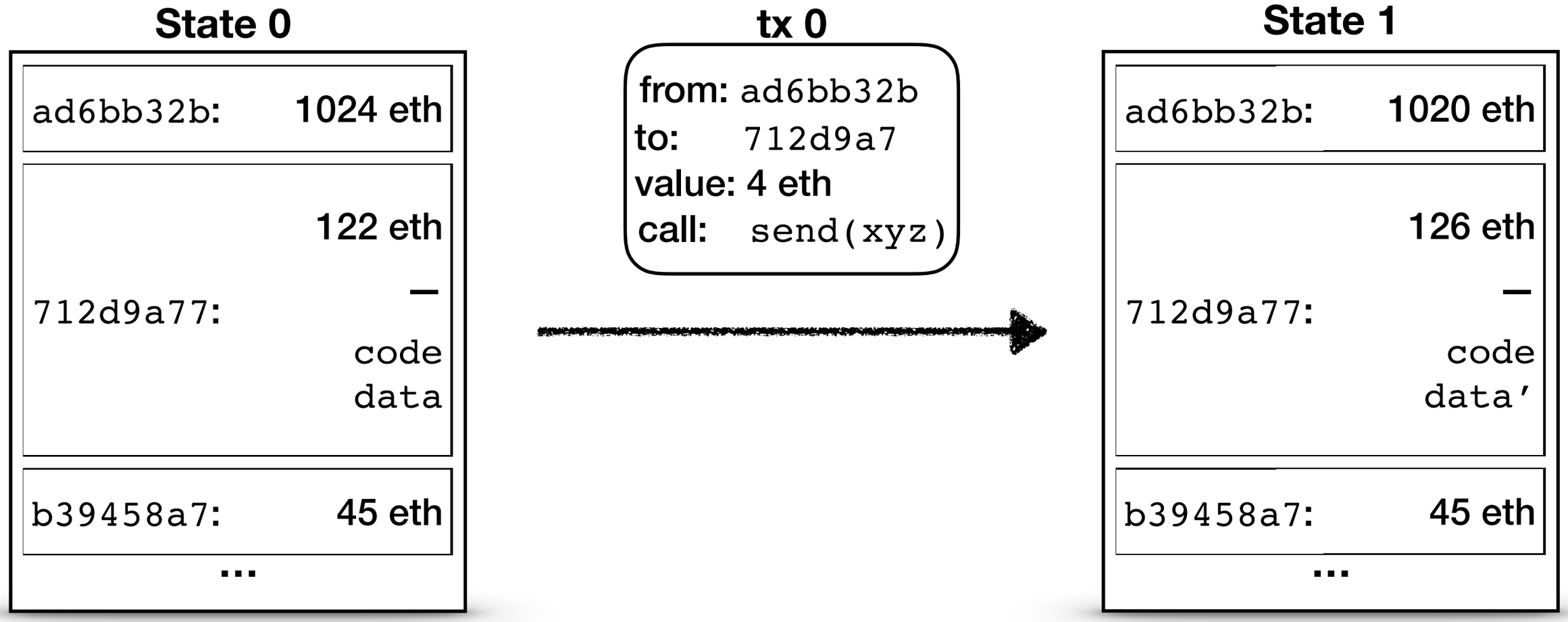
Ethereum

- 2 types of accounts
    - Externally owned accounts
        - Controlled by People (public/private keys)
        - No Code
    - Contract accounts
        - Controlled by code
        - Not something that needs to be fulfilled, but autonomous agents

Ethereum

- Each block describes a state of the Ethereum Virtual Machine (EVM)

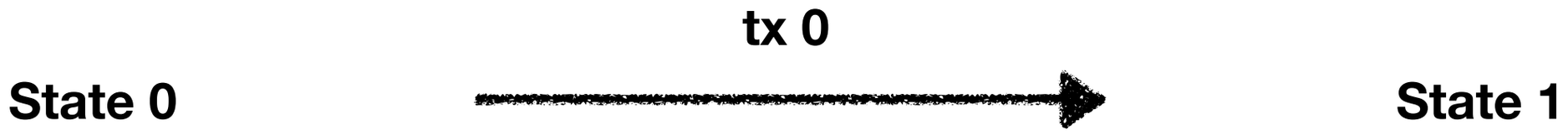
- Transactions are state transitions



Ethereum

- Each block describes a state of the Ethereum Virtual Machine (EVM)

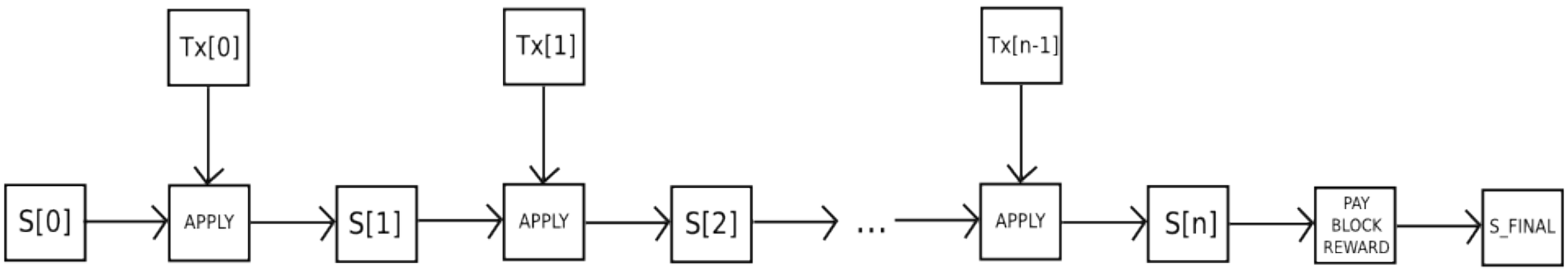
- Transactions are state transitions




```
APPLY(S[0], Tx[0]) → S[1]
```

Ethereum

- Each block describes a state of the Ethereum Virtual Machine (EVM)
- Transactions are state transitions
- Blocks are mined by processing a set of transitions



Ethereum White Paper `https://github.com/ethereum/wiki/wiki/White-Paper`

Smart Contracts

- With Smart Contracts, we can implement new currencies
  - A lookup table as ledger
  - User pay money (eth) to contract to get coins
  - Every transaction happens by external users sending messages to it
    - External user → Ethereum → new coin transactions

Smart Contracts Example (new currency)

if Smart Contracts were written in Python

```
class MyCoin:

    def __init__(self):
        self.balance = dict()
        self.balance[“CentralBank”] = 1000000000
```

Initialize by giving 1B MyCoins to the central bank

Smart Contracts Example (new currency)

if Smart Contracts were written in Python

```
class MyCoin:

    def __init__(self):
        self.balance = dict()
        self.balance[“CentralBank”] = 1000000000

    def getBalance(self, accountID):
        if accountID in self.balance:
            return self.balance
        return 0
```

reading the account balance

0 if the account doesn’t exist

Smart Contracts Example (new currency)

if Smart Contracts were written in Python

```
class MyCoin:

    def __init__(self):
        self.balance = dict()
        self.balance[“CentralBank”] = 1000000000

    def getBalance(self, accountID):
        if accountID in self.balance:
            return self.balance
        return 0

    def transfer(self, accountID, amount, to):
        maxTransferrable = min(self.getBalance(accountID),
                               amount)
        self.balance[accountID] -= maxTransferrable
        self.balance[to] = maxTransferrable
```

transfer `amount` from the sender (`accountID`) to the recipient (`to`).

If the balance is less than the amount, transfer only as much as available

Smart Contracts Example (new currency)

if Smart Contracts were written in Python

```
class MyCoin:

    def __init__(self):
        self.balance = dict()
        self.balance[“CentralBank”] = 1000000000

    def getBalance(self, accountID):
        if accountID in self.balance:
            return self.balance
        return 0

    def transfer(self, accountID, amount, to):
        maxTransferrable = min(self.getBalance(accountID),
                               amount)
        self.balance[accountID] -= maxTransferrable
        self.balance[to] = maxTransferrable

    def buy(self, accountID, amount):
        self.balance[accountID] += amount
```

buy MyCoins by sending eth

Solidity

- The programming language for Ethereum Smart Contracts
  (Python is not a good language for this)

  - Full fledged, object-oriented programming language
  - compiled and runs on the EVM

  - Can send money to other contracts/accounts on the EVM

  - cannot access I/O (i.e. Internet)
    - A block must always be verifiable, but in the future any external reference might have changed
    - immutable objects (history, nonces, blocks, etc.) are fine

Solidity Example

```
contract GavCoin
{
  mapping(address=>uint) balances;
  uint constant totalCoins = 100000000000;

  function GavCoin(){
      balances[msg.sender] = totalCoins;
  }

  function balance(address who) constant
       returns (uint256 balanceInmGAV) {
    balanceInmGAV = balances[who];
  }

  function send(address to, uint256 valueInmGAV)
   {
    if (balances[msg.sender] >= valueInmGAV) {
      balances[to] += valueInmGAV;
      balances[msg.sender] -= valueInmGAV;
    }
  }

}
```

Endows creator of contract with 100B GAV

$100 \cdot 10^9 = 100B$

getter function for the balance

Send
`$(valueInmGAV)`
GAV from the account of
`$(message.caller.address())`,
to an account accessible only by
`$(to.address())`

https://en.wikipedia.org/wiki/Solidity

Smart Contracts

- Used in many different applications
  - Betting/Prediction Markets
  - Standards (like ERC20) for tokens/currencies
  - CryptoKitties
  - …

- Potential problems
  - Everything is open source and cannot change once submitted
    - Bugs are inevitable, this is a security problem
  - Every computation is repeated many times
    - Whenever someone wants to mine/verify block

Smart Contracts

- Smart contracts and their implications are the scope of several other lectures