CSE1006 – Blockchain and Cryptocurrency Technologies

MODULE - 3

Mechanics of Bitcoin

Dr. Jayanthi.R

Content

What is Crypto Currency?

- Cryptocurrency is a digital currency in which **encryption techniques** are used to regulate the generation of **units of currency** and verify the **transfer of funds**, operating independently of a central bank.

Top 7 Crypto Currencies

1. Bitcoin
2. Ethereum
3. Litecoin
4. Peercoin
5. Dashcoin
6. Namecoin
7. NovaCoin

Mechanics of Bitcoin – Introduction

What is the easiest way to understand bitcoins?

- Bitcoin is **a decentralized digital currency that you can buy, sell and exchange directly, without an intermediary like a bank.**
- Bitcoin described the need for ***"an electronic payment system based on cryptographic proof instead of trust."***
- Bitcoin is the first **blockchain application**

What is Bitcoin's mechanism?

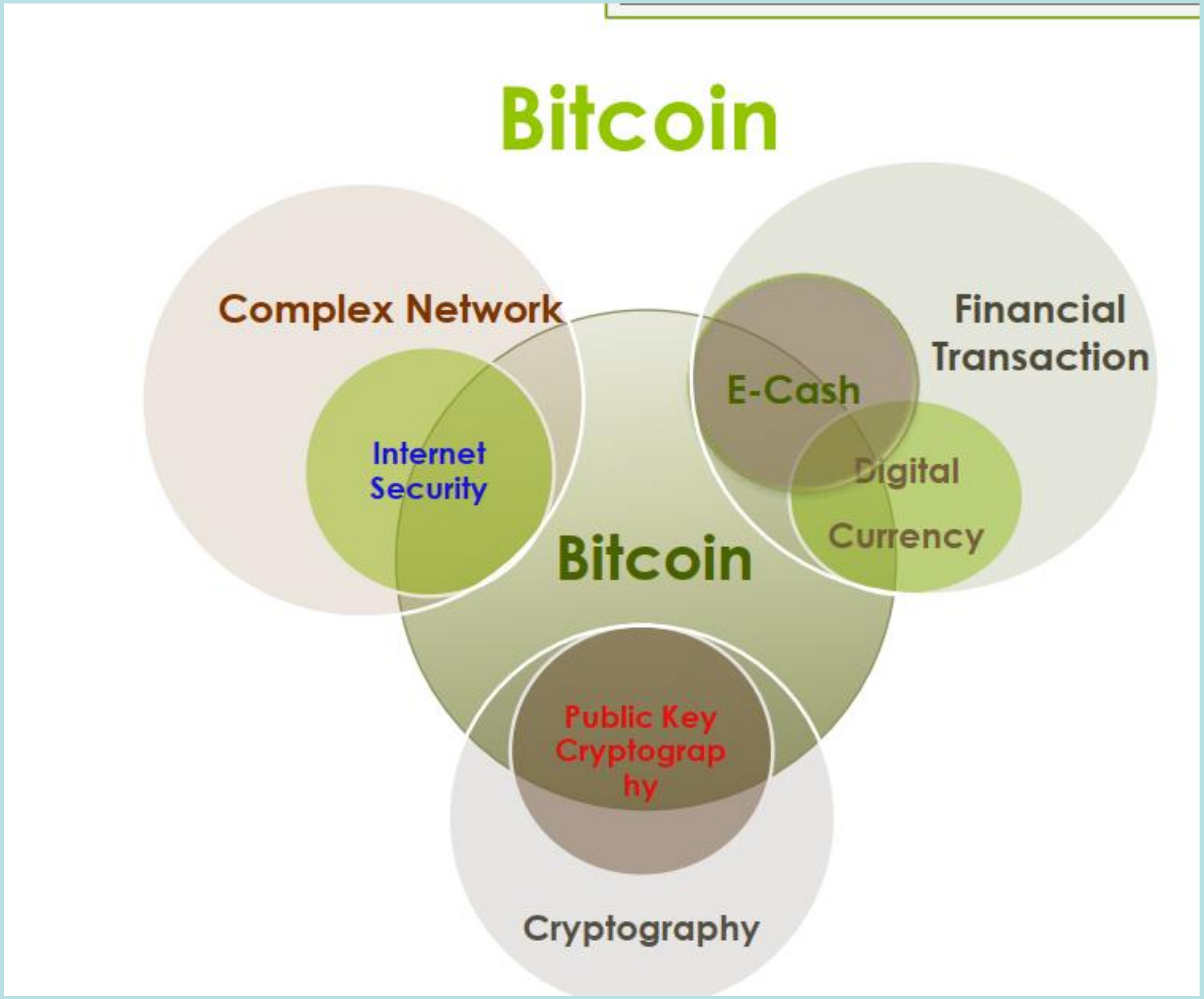
- Bitcoin is a form of **digital currency** that aims to eliminate the need for **central authorities** such as banks or governments.
- Bitcoin **uses blockchain technology to support peer-to-peer transactions between users on decentralized network**

Mechanics of Bitcoin – Introduction

Features of bitcoins

- Bitcoin transactions are relatively faster as compared to bank transfers in traditional currencies.
- Bitcoin transactions are done with nominal or sometimes zero transaction charges.
- These transactions are anonymous(unnamed) with no names involved.
- Every transaction is a public record that anyone can see.
- one of the major advantages of bitcoin usage is that **taxes** are not added to any purchases.
- Your **private key** is the only link between you and **your bitcoins**.
- As long as the private key is secure, your money is safe.
- It is very easy to send and receive bitcoins because of the ease of operation of **bitcoin accounts.**

Mechanics of Bitcoin – Introduction



WHAT IS BITCOIN MINING?

Bitcoin Mining

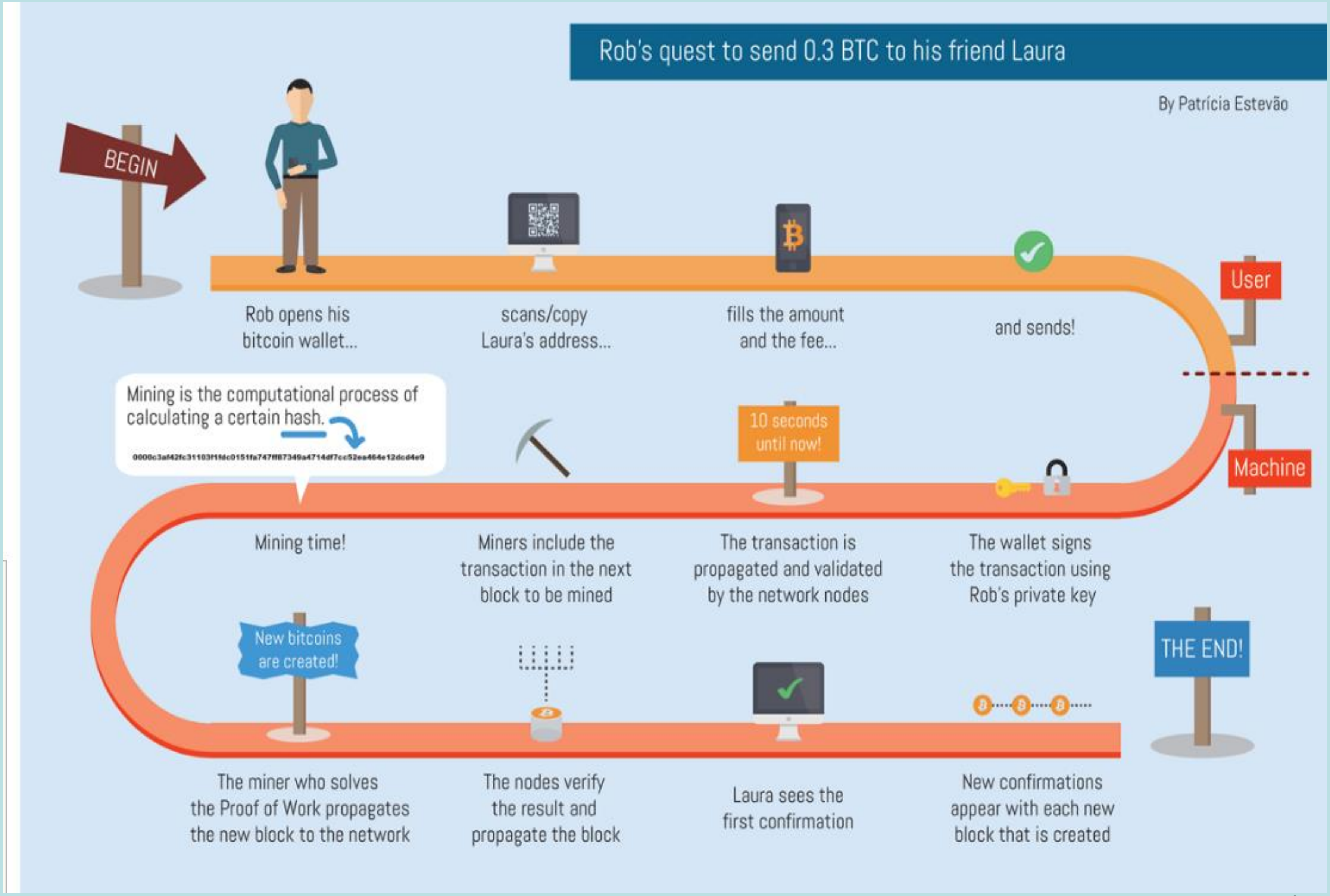
- **Bitcoin mining is the process by which Bitcoin transactions are validated digitally on the Bitcoin network and added to the blockchain ledger.**
- It is done by solving ***complex cryptographic hash puzzles*** to verify blocks of transactions that are updated on the ***decentralized blockchain ledger***

Blockchain (or Ethereum) mining - is used to verify transactions on the network.

WHAT IS BITCOIN MINING?

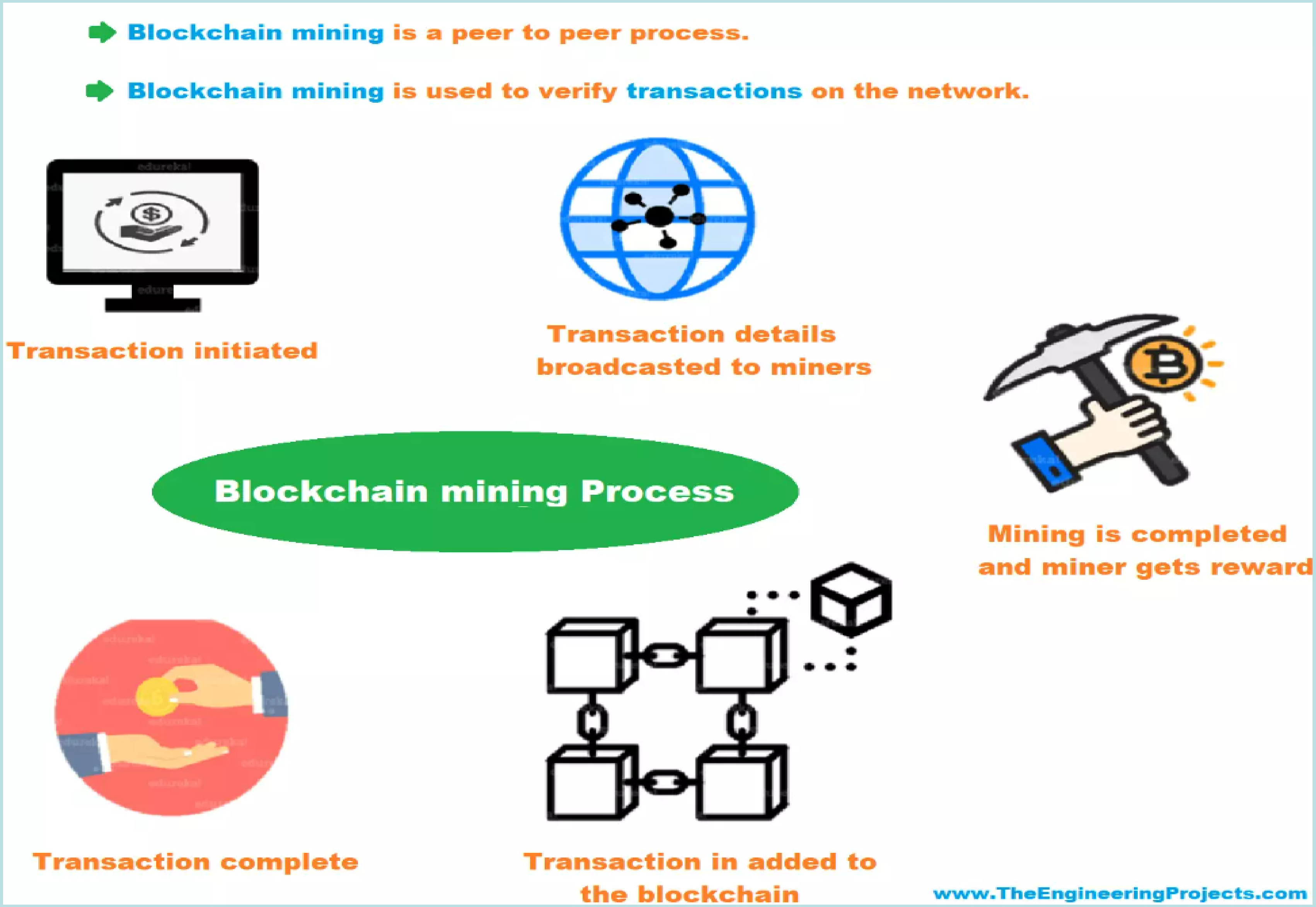
- Bitcoin mining is a ***computation-intensive process*** that uses ***complicated computer code*** to generate a ***secure cryptographic system***.
- Bitcoin mining is the process of ***making computer hardware do mathematical calculations for the Bitcoin network to confirm transactions*** and **increase security.**
- Bitcoin mining is the process of adding transaction records to the public accounting book of past transactions or blockchain.
- The difficulty of the algorithm increases with the number of bitcoins and transfers, so **block cracking** becomes more **difficult** and requires a lot of resources.
- A single computer may take **three** full years to crack a single block

BITCOIN TRANSACTION LIFECYCLE


Rob's quest to send 0.3 BTC to his friend Laura
By Patrícia Estevão
BEGIN
Rob opens his bitcoin wallet...
scans/copy Laura's address...
fills the amount and the fee...
and sends!
User
Machine
The wallet signs the transaction using Rob's private key
10 seconds until now!
The transaction is propagated and validated by the network nodes
Miners include the transaction in the next block to be mined
Mining is the computational process of calculating a certain hash.
0000c3af42fc31103f1fdc0151fa747ff87349a4714df7cc52ea464e12dcd4e9
Mining time!
New bitcoins are created!
The miner who solves the Proof of Work propagates the new block to the network
The nodes verify the result and propagate the block
Laura sees the first confirmation
New confirmations appear with each new block that is created
THE END!

- Today there are at least a ***few dozen Bitcoin mining*** programs, most of which allow users to connect to mining groups and mine together blocks worth **25 bitcoins**
- The bitcoin miner is the person who solves mathematical puzzles(also called proof of work) to validate the transaction.
- Anyone with mining hardware and computing power can take part in this. Numerous miners take part simultaneously to solve the complex mathematical puzzle, the ***one who solves it first***, **wins 6.25 bitcoin** as a part of the reward.
- Miner verifies the transactions(after solving the puzzle) and then adds the ***block to the blockchain*** when confirmed.
- The blockchain contains the history of every transaction that has taken place in the blockchain network.
- Once the minor adds the block to the blockchain, bitcoins are then transferred which were associated with the transaction.

Mining Process



- Blockchain mining is a peer to peer process.
- Blockchain mining is used to verify transactions on the network.

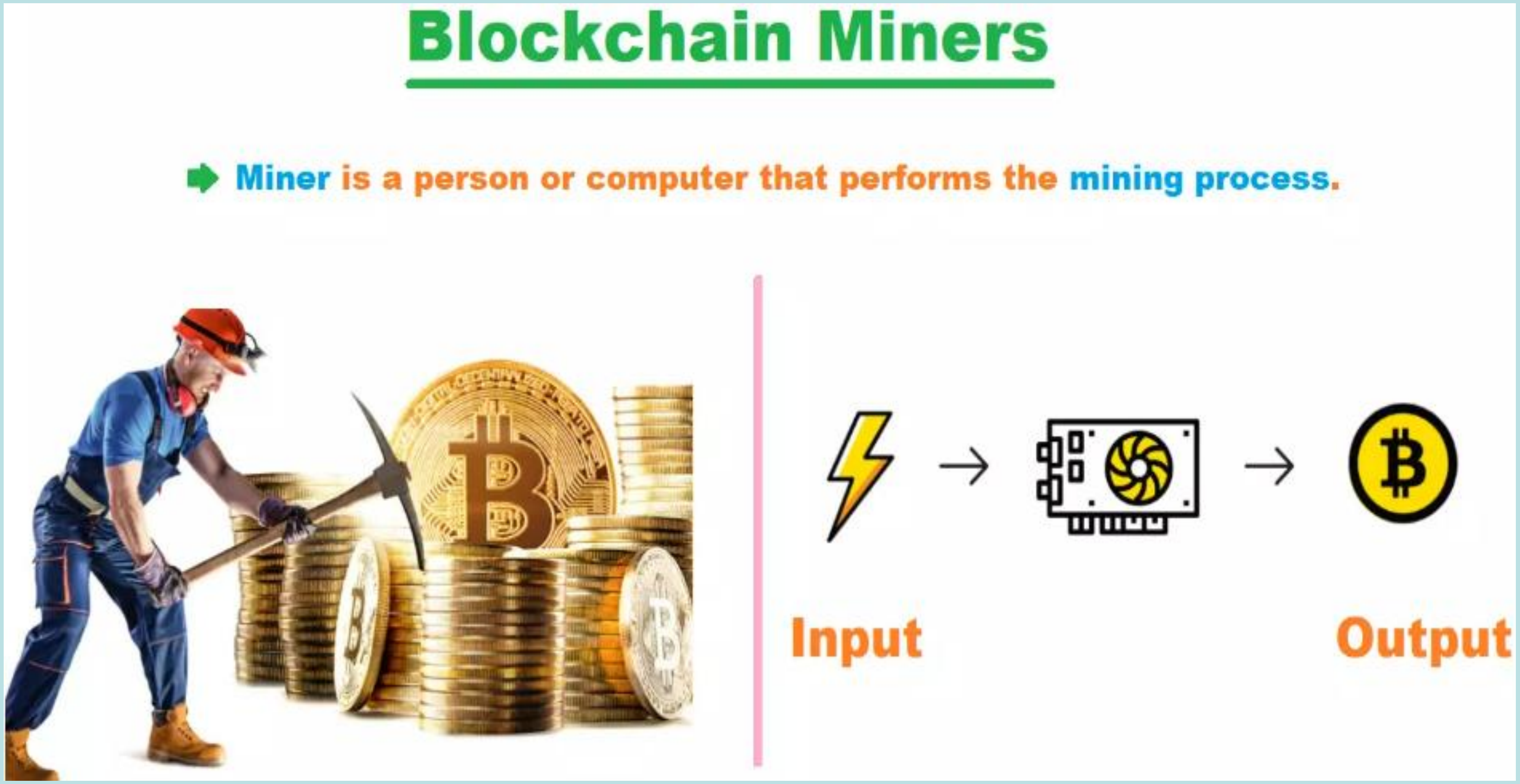
Transaction initiated
Transaction details broadcasted to miners
Blockchain mining Process
Mining is completed and miner gets reward
Transaction complete
Transaction in added to the blockchain
www.TheEngineeringProjects.com

Miners' role in blockchain

- In this process, **miners** (the person or node who performs mining) ***add transaction records*** to the **decentralized distributed ledger** of the blockchain.
- The transactions are added to the blocks, and the blocks are secured and linked after mining in the form of a chain.
- Mining requires computational power and effort from miners and basically, ***it is the process of adding blocks***.
- In this way, the transactions get confirmed and money and assets move from one account to another.
- This was the idea of mining.

Miners’ role in blockchain

- Miners provide their time, efforts, and computer resources for mining and serving the blockchain system


Blockchain Miners
Miner is a person or computer that performs the mining process.
Input
Output

Miners' role in blockchain

- Miners get transaction fees as a reward for their contribution.
- The transaction's sender (or initiator) pays this fee to the miner.
- A **miner adds** a *number of transactions* in the **new block** and collects the ***transaction fee*** for each of them.
- Any person can become a blockchain miner.
- A blockchain mining software needs to be installed and executed by the miner. This allows them to communicate with the blockchain network.
- A computer performing this process then becomes a node. The nodes interact with each other to collaborate for **verifying transactions.**

Why Does Bitcoin Needs Miners?

- Bitcoin miners are very essential for the smooth functioning of the ***bitcoin network*** for the following reasons:

1. Miners' job is just like **auditors** i.e. to verify the legitimacy of the bitcoin transactions.
2. Miners help to prevent the ***double-spending problem***.
3. Miners are minting the currency.
4. In the absence of miners, Bitcoin as the network would still exist and be usable but there would be no additional bitcoin.

Why Mine Bitcoins?

There are several pros of mining a bitcoin:

- Mining bitcoin helps support the Bitcoin ecosystem.
- Bitcoin mining helps miners to earn rewards in form of bitcoins.
- It is the only way to release new cryptocurrencies into circulation

How Does Bitcoin Mining Work?

- The nodes of the blockchain network are based on the concept that no one in the network can be trusted.
- Proof of work is accepted by nodes to validate any transaction.
- Proof of work involves doing hefty calculations to find a 32-bit hash value called nonce to solve the mathematical puzzle.
- The ***miners create new blocks*** by abiding by the fact that the transaction volume must be less than **21 million**.
- 21 million is the total number of bitcoins that can be generated.
- The verified transaction gets a **unique identification code** and is linked with the previous verified transaction.

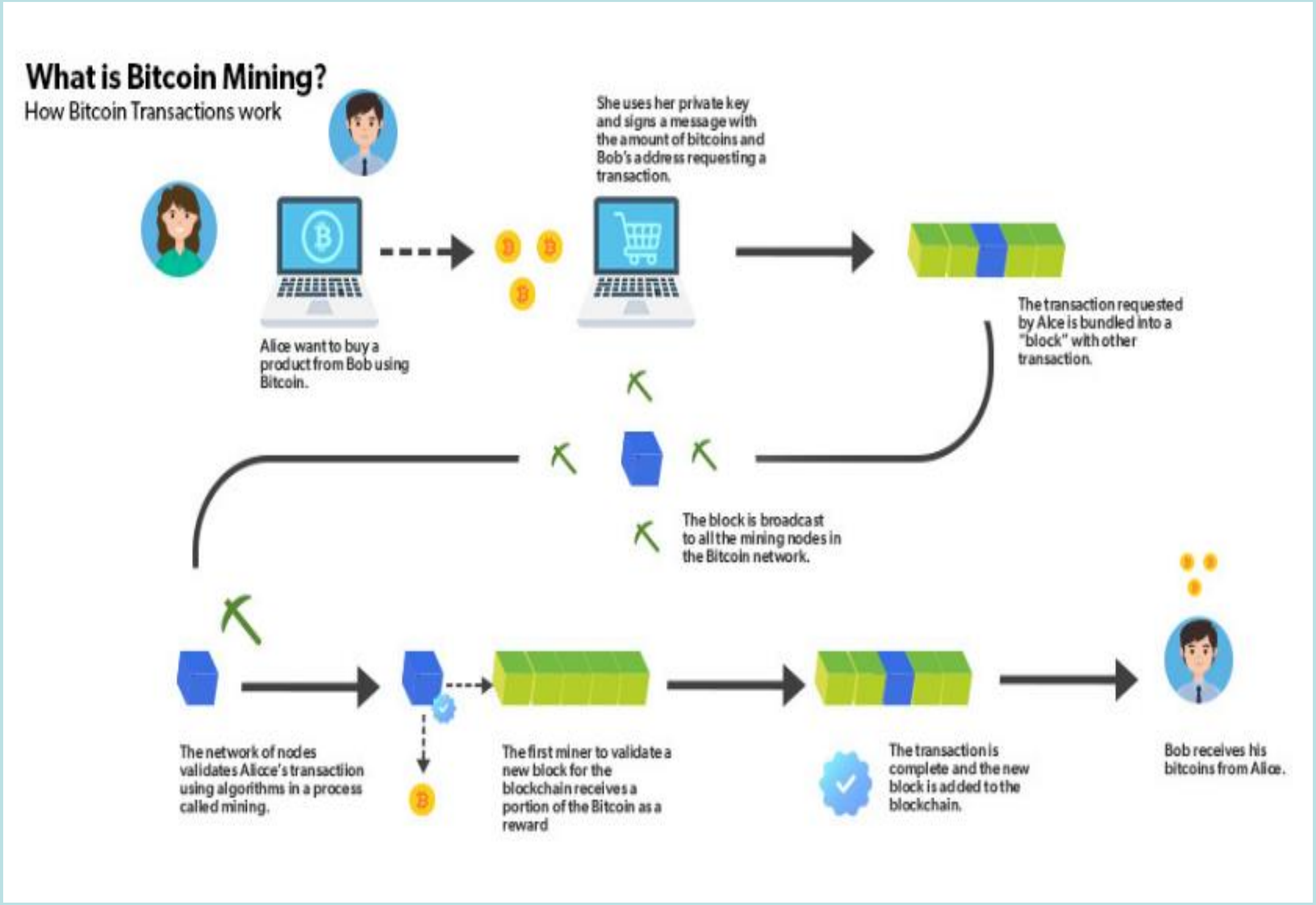
Let's understand this with the help of an example-

- Suppose ***Alice*** wants to transfer 10 BTC to ***Bob.***
- Now the transaction data of ***A*** is ***shared with the miners*** from ***the memory pool***.
- A memory pool is a place where an ***unconfirmed or unverified transaction*** waits for its **confirmation.**

How Does Bitcoin Mining Work?

- Miners start competing with themselves to solve the mathematical riddle in order to ***validate and verify the transaction*** using **proof of work.**
- The miner who solves the problem first shares his result with other nodes(miners).
- Once maximum nodes agree with the solution, the transaction block is verified and is then added to the blockchain.
- At the same time, the miner who solved the puzzle gets a reward of **6.25 bitcoins.**
- Now, after the addition of the transaction block, the 10 BTC associated with the transaction data is transferred to Bob from Alice.

How Does Bitcoin Mining Work?


What is Bitcoin Mining?
How Bitcoin Transactions work
She uses her private key and signs a message with the amount of bitcoins and Bob's address requesting a transaction.
Alice want to buy a product from Bob using Bitcoin.
The transaction requested by Alice is bundled into a "block" with other transaction.
The block is broadcast to all the mining nodes in the Bitcoin network.
The network of nodes validates Alice's transaction using algorithms in a process called mining.
The first miner to validate a new block for the blockchain receives a portion of the Bitcoin as a reward
The transaction is complete and the new block is added to the blockchain.
Bob receives his bitcoins from Alice.

Requirements to Mine Bitcoin

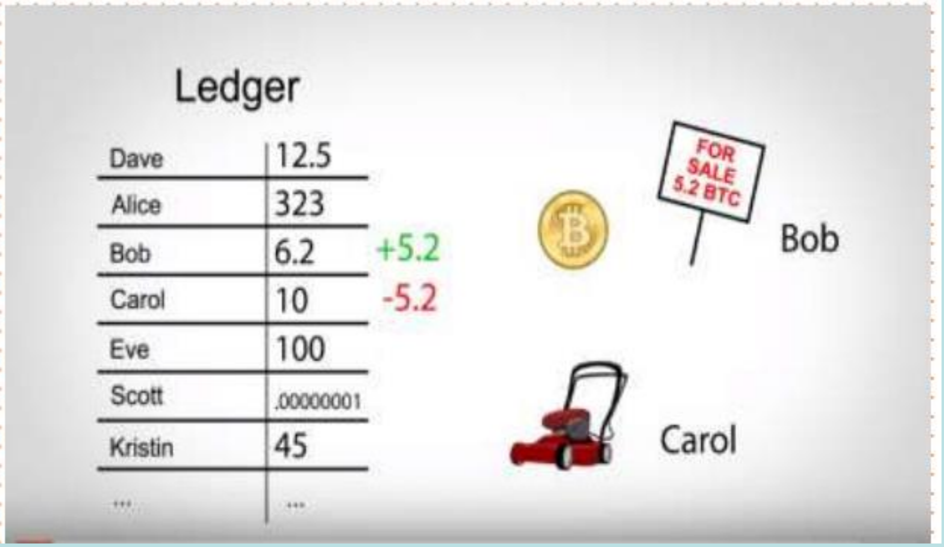
- In the past, users of the system used to mine bitcoins using their home computers but as the technology has improved, this is no longer the case.
- The general time a bitcoin network takes to verify a new transaction is **10min**. Within that time, there are ***more than one million miners competing with each other to find the hash value***.
- When there is more computing power working together to mine for bitcoins, the difficulty level of mining increases.
- Therefore, in order to mine bitcoins, the user must possess-Specialized mining hardware called
  - ✓ "application-specific integrated circuits," or ASICs.
  - ✓ A Bitcoin mining software to join the Blockchain network.
  - ✓ Powerful GPU (graphics processing unit).

HOW BITCOIN WORKS?

- There can be only **one owner** of any Bitcoin.
- For every bitcoin an owner holds, that bitcoin has a **unique address**, and this **address changes** only when a transaction of that bitcoin takes place.
- This address is nothing but **a unique set of characters** that ***one bitcoin*** can have and it ***does not overlap*** with others.
- **Miners** are responsible for handling the Blockchains.
- They make sure all the transactions are recorded and they generate **new addresses** for Bitcoins ***when the transaction is completed***.
- The best thing about Bitcoin is that despite using a publicly **distributed system** where people can see the transactions.
- The use of **private keys** hides the **owner's identity** thereby preserving privacy.

HOW BITCOIN WORKS?

- At a very basic level, Bitcoin is just a digital file or ledger that contains names and balances.
- Suppose **Bob** sells **Carol** a lawn mover for ***5.2 Bitcoin***, Bob's balance ***goes up*** by **5.2**, and Carol's balance ***down*** by **5.2**.



Ledger

Dave	12.5	
Alice	323	
Bob	6.2	+5.2
Carol	10	-5.2
Eve	100	
Scott	.00000001	
Kristin	45	
...	...	

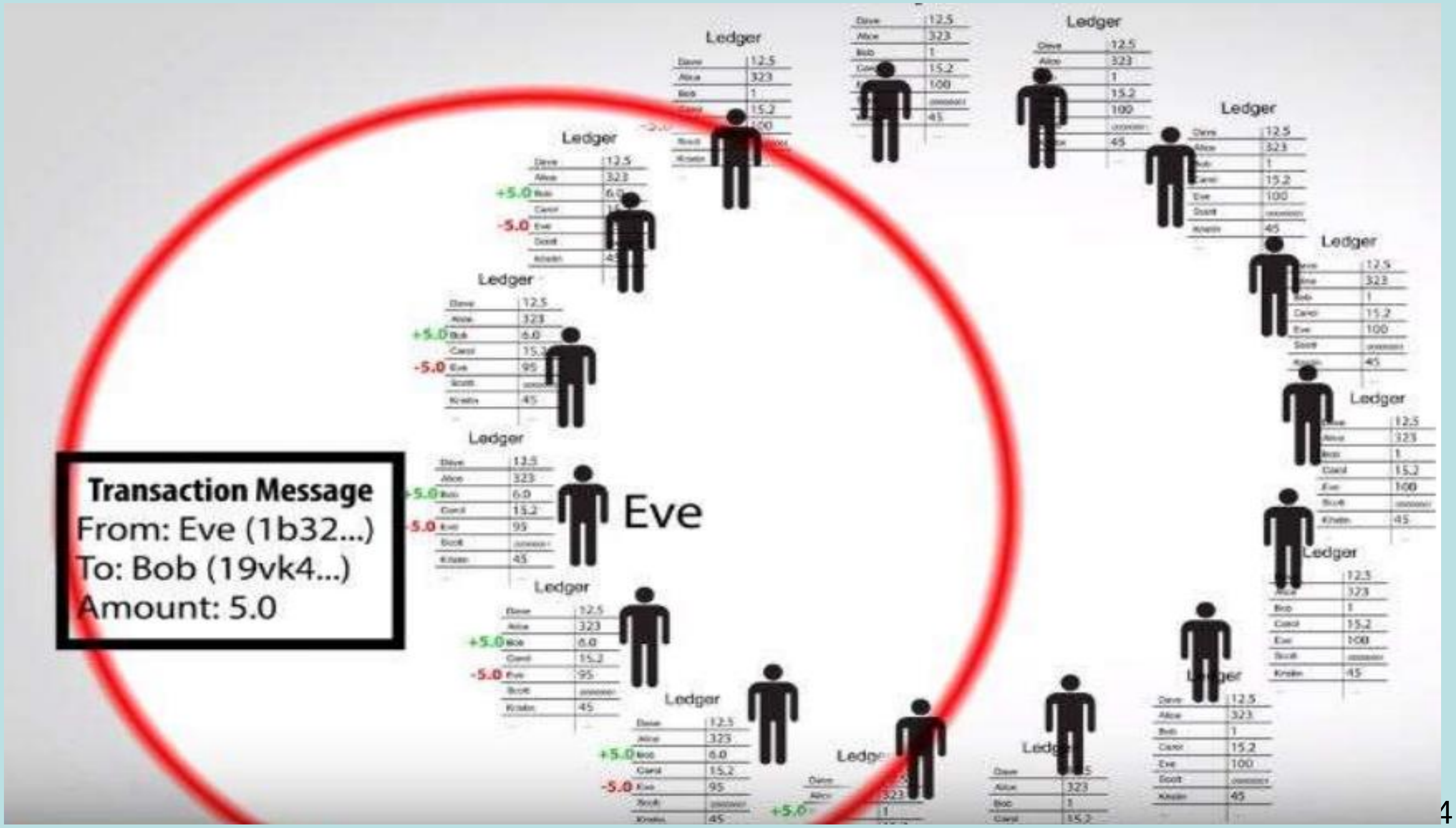
HOW BITCOIN WORKS?

- Everyone can see **everyone's balance** contains ***name and balance.***

Ledger
13v8NB9ScR... 12.5
15v76Prde3... 323
1W5436gh7... 1
1jhg543bfd...
1ghg65656... .00000001
1gfg543455... 45

HOW BITCOIN WORKS?

- If you **want to send money**, you simply tell everyone by ***broadcasting a message*** with **your account**, the **receiver's account**, and the **amount of money**.



Transaction Message
From: Eve (1b32...)
To: Bob (19vk4...)
Amount: 5.0

HOW BITCOIN WORKS?

- It avoids the concept of any centralized control.
- Every participant maintains their own copy of the ledger.
- One surprising fact of this is that everyone can see everyone else's balances.
- Like a pen & paper check, Bitcoin requires a kind of signature to prove that the sender is the real owner of the account.
- Signature is based on math rather than handwriting.
- Bitcoin Addresses are ***160-bit long hash outputs*** of the **private key.**
- When a new account number is created, It comes along with a **private key**.
- **private key** is mathematically linked **to that account number.**

- Before the transaction **a signature** is created using a cryptographic function.
- To create a signature, **a private key,** and the **text** from a
- transaction (or say messages) are **fed** into a special
- cryptographic function.
- It allows other people to check the signature, making sure it was created by the account owner, and that it applies to that specific transaction.
- A Bitcoin wallet address is similar to a **bank account** number.
- It's a unique ***26-35 digit combination*** of letters and numbers and it looks something like this: 1ExAmpLe0FaBiTco1NADr3sSV5tsGaMF6hd
- A private key is usually a **256-bit number** something like this:

E9873D79C6D87DC0FB6A5778633389F4453213303DA61F20BD67FC233AA33262

BITCOIN TRANSACTIONS

- Let’s start with transactions, Bitcoin’s fundamental building block.
- We’re going to use a **simplified model** of a **ledger** for the moment.
- Instead of blocks, let’s suppose individual transactions are added to the **ledger** *one at a time*.
- How can we **build** a currency on **top of such a ledger**?
- The first model you might think of, which is actually the mental model many people have for how Bitcoin works, is that you have an **account-based system**.
- You can add some transactions that **create new coins** and **credit them to somebody** then later you can **transfer** them.
- ***A transaction would say something like “we’re moving 17 coins from Alice to Bob”, and it will be signed by Alice***.
- That’s all the information about the transaction that’s contained in the ledger

BITCOIN TRANSACTIONS

What are the fundamental building blocks of bitcoin transactions?

Transaction Outputs and Inputs.

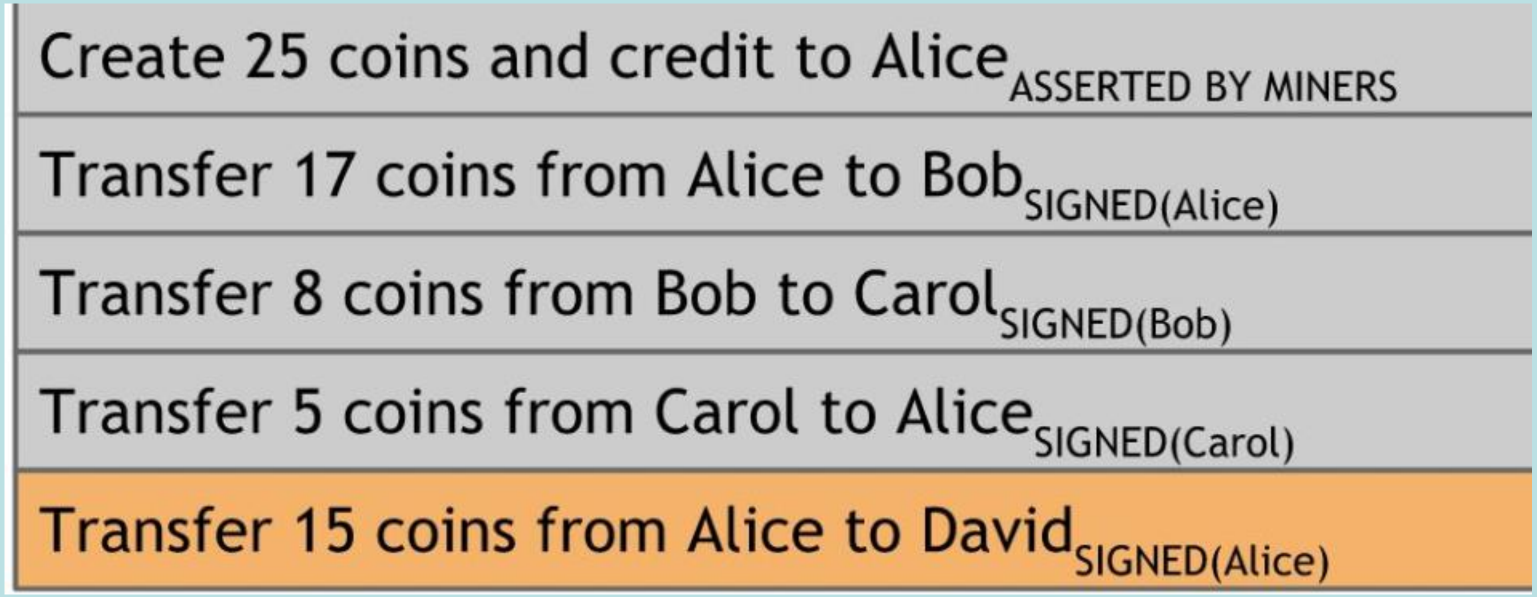
- The fundamental building block of a bitcoin transaction is an ***"unspent transaction output"*** or **UTXO.**
- **UTXO** is ***indivisible chunks*** of bitcoin ***currency locked*** to a *specific owner*, ***recorded*** on the blockchain, and recognized as **currency units** by the *entire network*.
- The bitcoin network tracks all available (unspent) UTXO currently numbering in the millions.
- Whenever a user receives bitcoin, that amount is recorded within the blockchain as a UTXO.

Transaction Outputs and Inputs

- Thus, a user's bitcoin might be scattered as UTXO amongst hundreds of transactions and hundreds of blocks
- There is no ***stored balance*** of a bitcoin address or account; there are only ***scattered*** UTXO, ***locked*** to specific owners.
- The concept of a user's bitcoin balance is a "derived construct" created by the **wallet application.**
- The wallet calculates the **user's balance** by **scanning** the blockchain and aggregating all UTXO belonging to that **user.**

Note: There are **no accounts or balances in bitcoin**; there are only *unspent transaction outputs* (UTXO) scattered in the blockchain.

- The below Figure(**3.1 from the ebook)** shows account based ledger, here **Alice** receives **25 coins** in the first transaction and then **transfers 17 coins** to **Bob.**
- In the second, she'd have ***8 Bitcoins left in her account.***



Create 25 coins and credit to Alice$_{\text{ASSERTED BY MINERS}}$

Transfer 17 coins from Alice to Bob$_{\text{SIGNED(Alice)}}$

Transfer 8 coins from Bob to Carol$_{\text{SIGNED(Bob)}}$

Transfer 5 coins from Carol to Alice$_{\text{SIGNED(Carol)}}$

Transfer 15 coins from Alice to David$_{\text{SIGNED(Alice)}}$

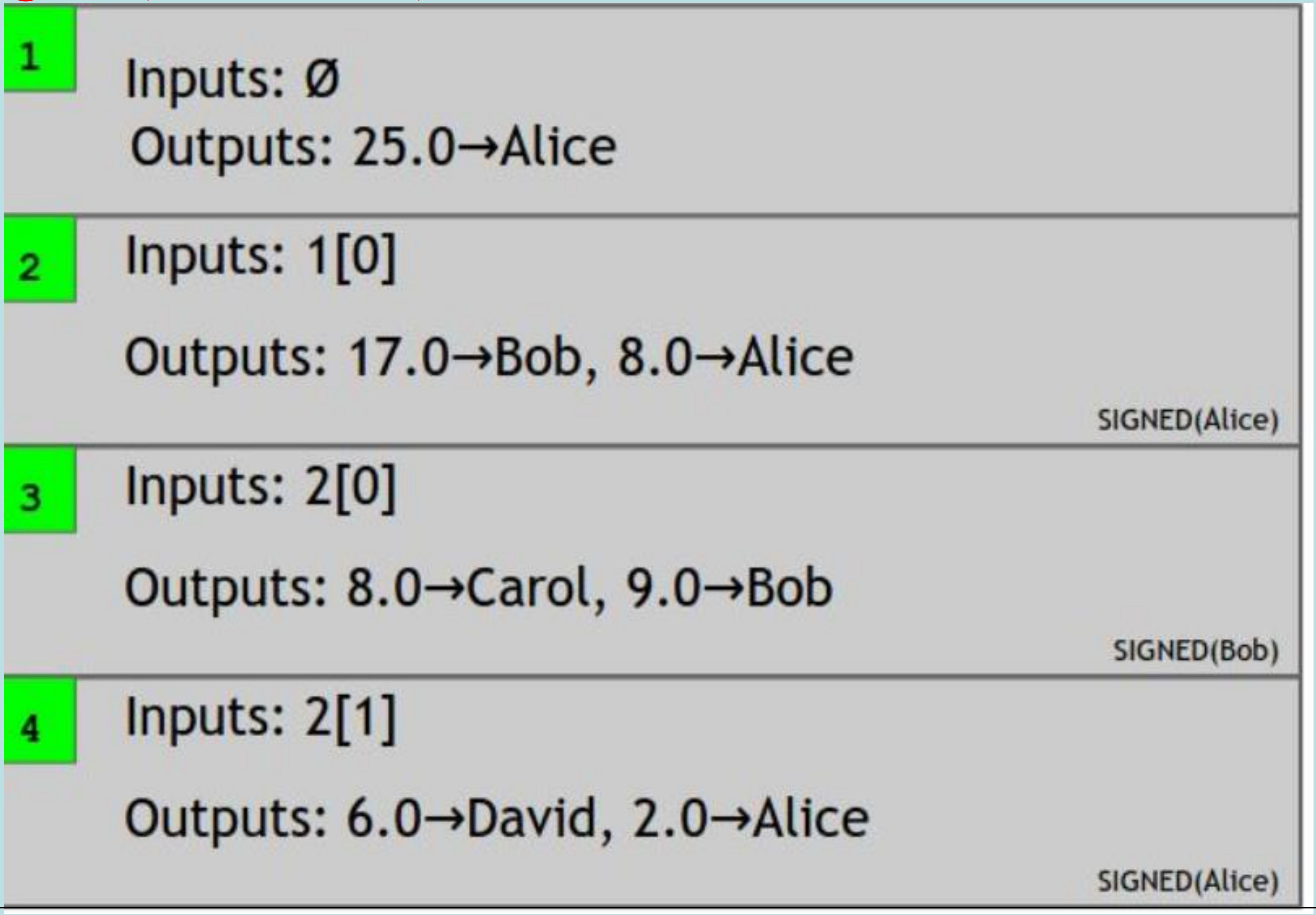
- Anyone who wants to determine if a transaction is valid will have to keep track of these account balances.

Take another look at the Figure,

- Does Alice have the ***15 coins*** that she's trying to transfer to David?
- To figure this out, you'd have to look backward in time forever to see every transaction affecting Alice, and whether or not her net balance at the time that she tries to transfer 15 coins to David is greater than ***15 coins***.
- Of course, we can make this a little bit more **efficient** with some data structures that track Alice's balance after each transaction.
- But that's going to require a lot of extra housekeeping besides the ledger itself.
- Because of these **downsides**, Bitcoin doesn't use an **account-based model**.
- Instead, Bitcoin uses a **ledger** that just keeps track of transactions similar to ScroogeCoin in Chapter 1.

A transaction-based ledger, which is very close to Bitcoin

Fig 3.2 (as in ebook)



1
Inputs: Ø
Outputs: 25.0→Alice

2
Inputs: 1[0]
Outputs: 17.0→Bob, 8.0→Alice
SIGNED(Alice)

3
Inputs: 2[0]
Outputs: 8.0→Carol, 9.0→Bob
SIGNED(Bob)

4
Inputs: 2[1]
Outputs: 6.0→David, 2.0→Alice
SIGNED(Alice)

A transaction-based ledger, which is very close to Bitcoin

- Transactions **specify** the ***number of inputs*** and ***number of outputs*** ("*recall PayCoins in ScroogeCoin*").
- You can think of the ***inputs*** as **coins** being *consumed* (ie: created in a ***previous transaction***) and the ***outputs*** as *coins being created*.
- For **transactions** in which ***new currency*** is being **minted**(issued), there are no coins being consumed(spent) (recall **CreateCoins** in **ScroogeCoin**).
- Each transaction has a ***unique identifier***.
- Outputs are **indexed** beginning ***with 0***, so we will refer to the first output as **"output 0".**

A transaction-based ledger, which is very close to Bitcoin

Figure 3.2.,

- Transaction 1 has **no inputs** because this transaction is ***creating new coins***, and it has an output of ***25 coins*** going to **Alice**.
- This is a transaction where **new coins** are being created, **no *signature is required***.
- Now let's say that **Alice** wants to **send** some of those coins over **to Bob**.
- To do so, she **creates** a new transaction, **transaction 2** in our example.
- In the transaction, she has to explicitly refer to the previous transaction where these coins are coming from.
- Here, she refers to ***output 0 of transaction 1*** (indeed(really) the only output of ***transaction 1***), which assigned ***25 bitcoins to Alice***.
- She also must specify the **output addresses** in the transaction.

A transaction-based ledger, which is very close to Bitcoin

- In this example, **Alice** specifies **two outputs**, **17 coins** to **Bob**, and **8 coins** to **Alice**.
- This whole thing is signed by **Alice** so that we know that **Alice** actually ***authorizes this transaction***.

Change addresses.

Why does Alice have to send money to herself in this example?

- Just as coins in ScroogeCoin are **immutable**, in Bitcoin, the entirety of a ***transaction output*** must be **consumed** by another transaction, or **none of it.**
- Alice only wants to pay **17 bitcoins** to Bob, but the output that she owns is worth **25 bitcoins**.
- So she needs to **create a new output** where **8 bitcoins** are sent back to herself.
- It could be a different address from the one that owned the 25 bitcoins, but it would have to be owned by her. This is called a ***change of address***

A transaction-based ledger, which is very close to Bitcoin

Efficient verification.

- When a new transaction is added to the ledger, how easy is it to check if it is valid?
- In this example, we need to look up the transaction output that Alice **referenced,** make sure that it has a value of **25 bitcoins**, and that it hasn't already been spent.
- Looking up the transaction output is easy since we're using hash pointers.
- To ensure it hasn't been spent, we need to scan the blockchain between the referenced transaction and the latest block.
- We don't need to go all the way back to the beginning of the blockchain, and it doesn't require keeping any additional data structures. (although, as we'll see, additional data structures will speed things up).

A transaction-based ledger, which is very close to Bitcoin

Consolidating funds.

- As in ScroogeCoin, since transactions can have many inputs and many outputs, splitting and merging value is easy.

For example

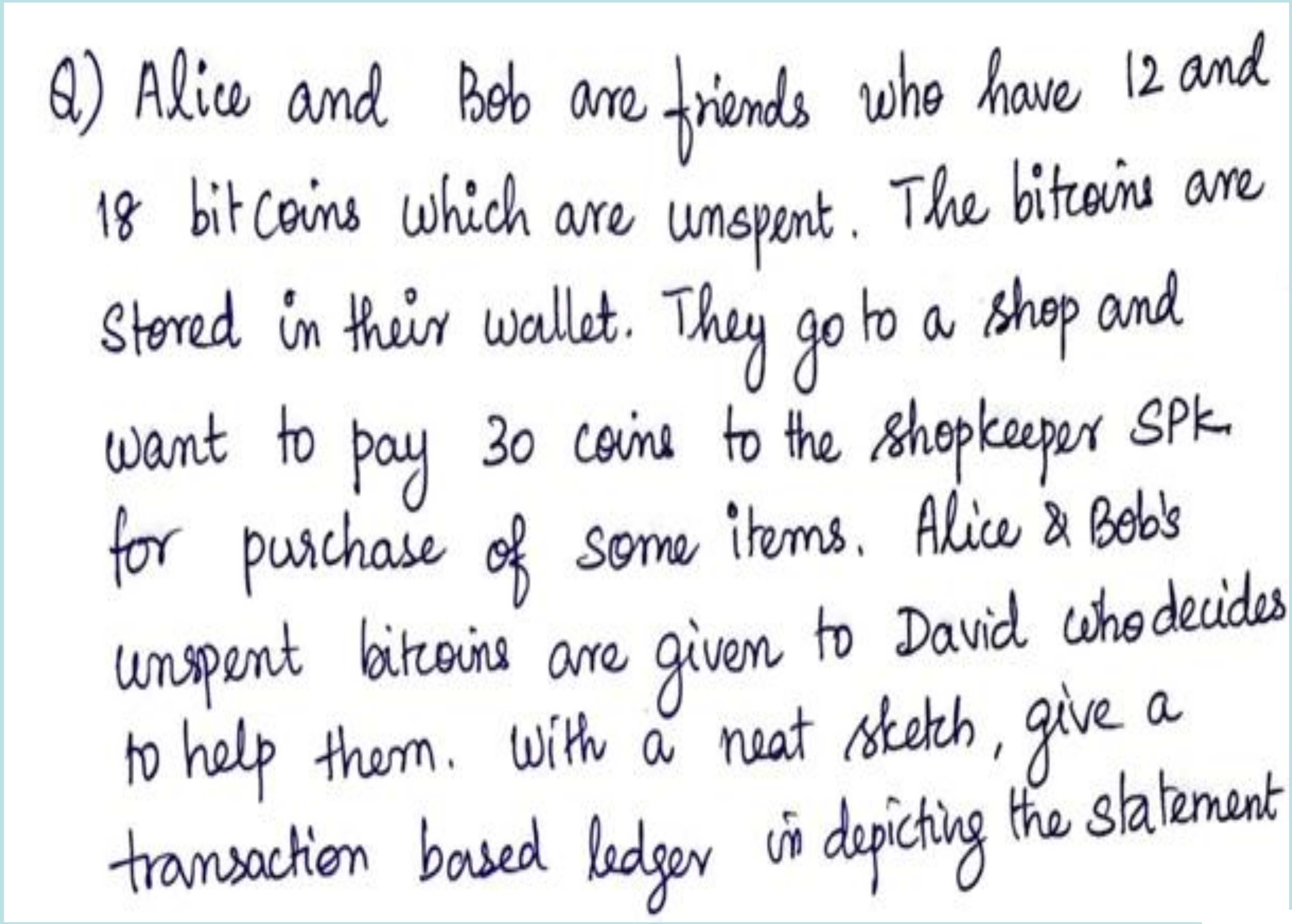
- Bob received the money in ***two different*** transactions — 17 bitcoins **in one**, and **2** in another.
- Bob might say, I'd like to have one transaction and I can spend later where I have all 19 bitcoins.
- That's easy — he creates a transaction with **two inputs and one output**, with the output address being **one** that he owns. That lets him consolidate those ***two transactions***

A transaction-based ledger, which is very close to Bitcoin

Joint payments.

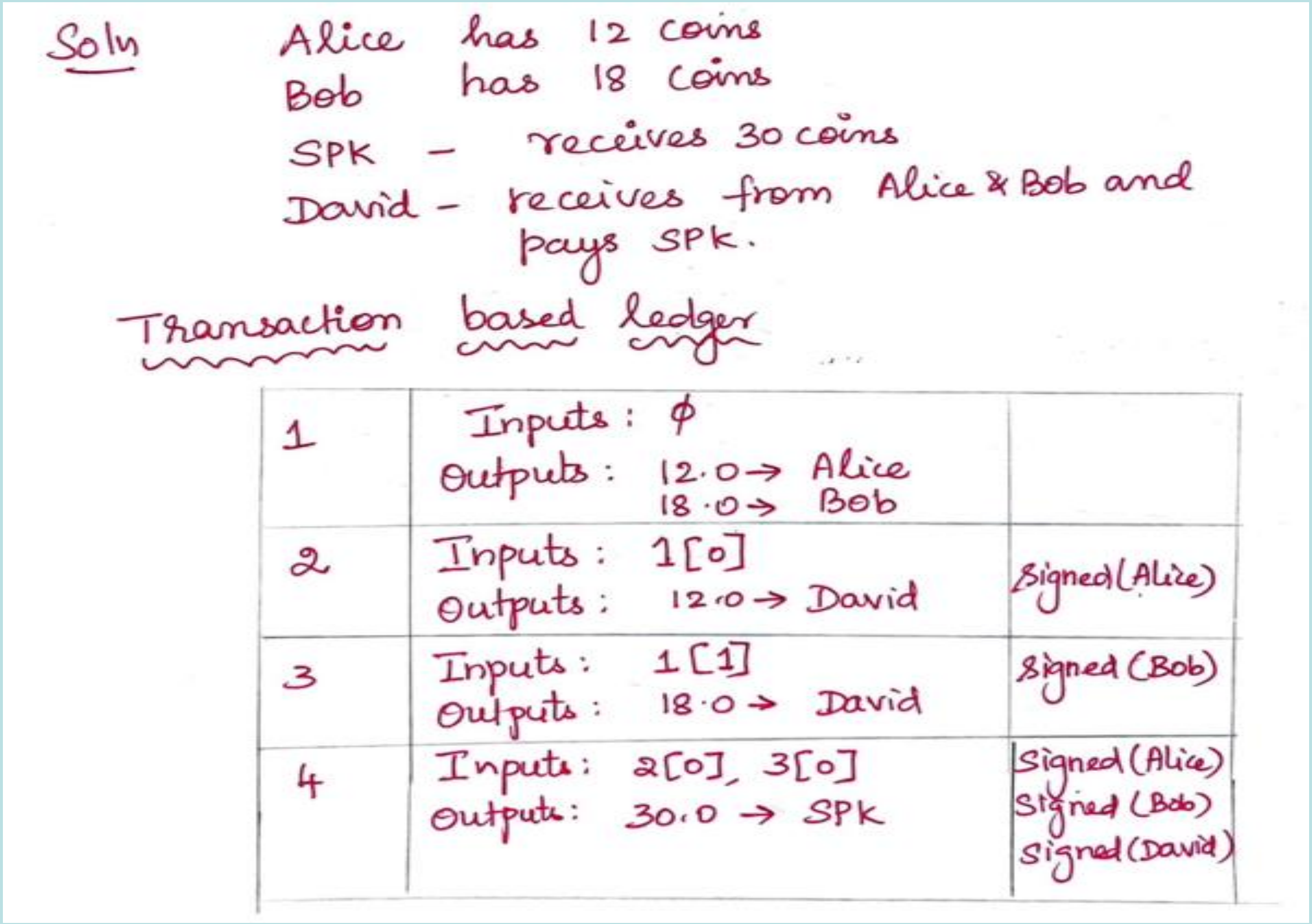
- Similarly, joint payments are also easy to do. Say **Carol and Bob both want to pay** David.
- They can create a transaction with ***two inputs and one output***, but with the ***two inputs owned by two different people***.
- And the only difference from the previous example is that since the ***two outputs from prior transactions that are being claimed here are from different addresses***, the transaction will need ***two separate signatures*** — **one by Carol and one by Bob**.

Example



Q) Alice and Bob are friends who have 12 and 18 bit coins which are unspent. The bitcoins are stored in their wallet. They go to a shop and want to pay 30 coins to the shopkeeper SPK for purchase of some items. Alice & Bob's unspent bitcoins are given to David who decides to help them. With a neat sketch, give a transaction based ledger in depicting the statement

Example



Soln

Alice has 12 coins
Bob has 18 coins
SPK – receives 30 coins
David – receives from Alice & Bob and pays SPK.

Transaction based ledger

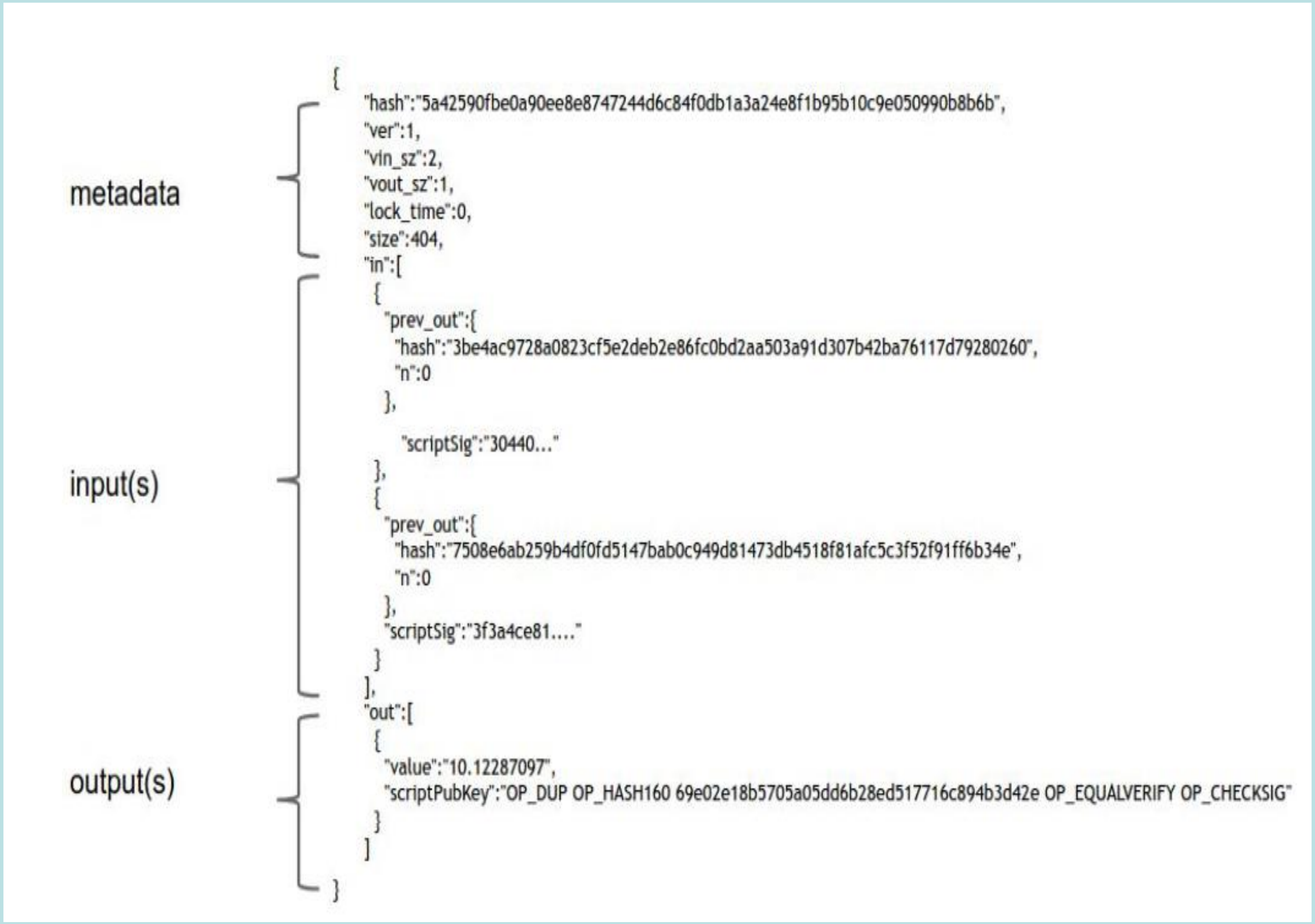
1	Inputs: ϕ Outputs: 12.0 → Alice 18.0 → Bob	
2	Inputs: 1[0] Outputs: 12.0 → David	Signed(Alice)
3	Inputs: 1[1] Outputs: 18.0 → David	Signed (Bob)
4	Inputs: 2[0], 3[0] Outputs: 30.0 → SPK	Signed (Alice) Signed (Bob) Signed (David)

A transaction-based ledger, which is very close to Bitcoin

Transaction syntax.

- Conceptually that's really all there is to a Bitcoin transaction.
- Now let's see how it's represented at a **low level** in Bitcoin.
- Ultimately, every data structure that's sent on the network is a **string of bits.**
- What's shown in below Figure is very low-level, but this further gets **compiled** down to a compact binary format **that's not human-readable.**

A transaction-based ledger, which is very close to Bitcoin



```
{
  "hash":"5a42590fbe0a90ee8e8747244d6c84f0db1a3a24e8f1b95b10c9e050990b8b6b",
  "ver":1,
  "vin_sz":2,
  "vout_sz":1,
  "lock_time":0,
  "size":404,
  "in":[
   {
    "prev_out":{
     "hash":"3be4ac9728a0823cf5e2deb2e86fc0bd2aa503a91d307b42ba76117d79280260",
     "n":0
    },

      "scriptSig":"30440..."
   },
   {
    "prev_out":{
     "hash":"7508e6ab259b4df0fd5147bab0c949d81473db4518f81afc5c3f52f91ff6b34e",
     "n":0
    },
    "scriptSig":"3f3a4ce81...."
   }
  ],
  "out":[
   {
    "value":"10.12287097",
    "scriptPubKey":"OP_DUP OP_HASH160 69e02e18b5705a05dd6b28ed517716c894b3d42e OP_EQUALVERIFY OP_CHECKSIG"
   }
  ]
}
```

A transaction-based ledger, which is very close to Bitcoin

- As you can see in the above fig, there are **three parts** to a transaction:

1. metadata,
2. a series of inputs, and
3. a series of outputs.

Metadata .

- There's some **housekeeping information – ie:**

— the size of the transaction, the number of inputs, and the number of outputs.

- There's the ***hash of the entire transaction*** which serves as a ***unique ID for the transaction***.
- That's what allows us to use hash pointers ***to reference transactions***. Finally, there's a **"lock_time"** field, which we'll come back to later

A transaction-based ledger, which is very close to Bitcoin

Input

- The ***transaction inputs*** form an array, and **each input** has the **same form**.
- An input specifies ***a previous transaction***, so it contains a hash of that transaction, which acts as a ***hash pointer to it***.
- The input also contains the ***index of the previous transaction's* outputs** that are being claimed. And then there's a **signature.**
- Remember that we have to sign to show that we actually have the ability to **claim** those ***previous transaction outputs.***

A transaction-based ledger, which is very close to Bitcoin

Outputs.

- The outputs are **again an array**.
- Each output has just **two fields.**
- They each have ***a value, and the sum of all the output values*** has to be ***less than or equal to the sum of all the input values***.
- If the ***sum of the output values is less than the sum of the input values***, the difference is a **transaction fee** to the **miner** who **publishes this transaction**.
- Then there's a line that looks like what we want to be the recipient address.
- Each output is supposed to go to a **specific public key**, and indeed there is something in that field that looks like it's the ***hash of a public key***.
- But there's also some other stuff that looks like a **set of commands**. Indeed, this **field is a script**, and we'll discuss this presently.

Bitcoin Scripts

- Bitcoin Script is the language Bitcoin uses to do everything it can do, from sending funds from a wallet to allowing the creation of multi-user accounts.
- When we talk about bitcoin script, we talk about a programming language simply used in Bitcoin for the processing of transactions that are read from **left to right**.
- This is based on a series of **linear structures**, known as the **stack**, which contain existing data in order **LIFO** (Last In - First Out).
- Each instruction in this language is executed one after the **other consecutively.**
- This language is not fully Turing because its **functionality** is **limited** and **cannot loop.**

Bitcoin Scripts

- This limitation is intentional as this **prevents infinite** or **endless looping** and **error execution.**
- In **Bitcoin** to communicate our wishes, the **opcodes (OP CODES)**, serve ***various functions***.
- Like memory manipulation, math, loops, and function calls, among many others.
- Therefore, ***Bitcoin Script is essentially a set of programmed instructions that are recorded with every transaction made.***
- These instructions describe how users can access and make ***use of the bitcoins*** available on the network.

Bitcoin Scripts

What is an OP_CODE or Operation Code?

- In computing, an OP_CODE (Operation Code, in English), is a portion of a machine language instruction that specifies the operation to be performed.
- Its specification and format will be determined by the **instruction set architecture (ISA)** of the component that processes the instruction.
- Generally, this processing is done by computer hardware (usually a CPU).
- But it can also be software specially prepared to emulate the operation of a ***CPU and process these instructions.***
- In general, a complete machine language instruction contains an **OP_CODE** and, optionally, the specification of **one or more operands,** on which the operation code must act.
- Some operations have implicit operands or none at all.

Bitcoin Scripts

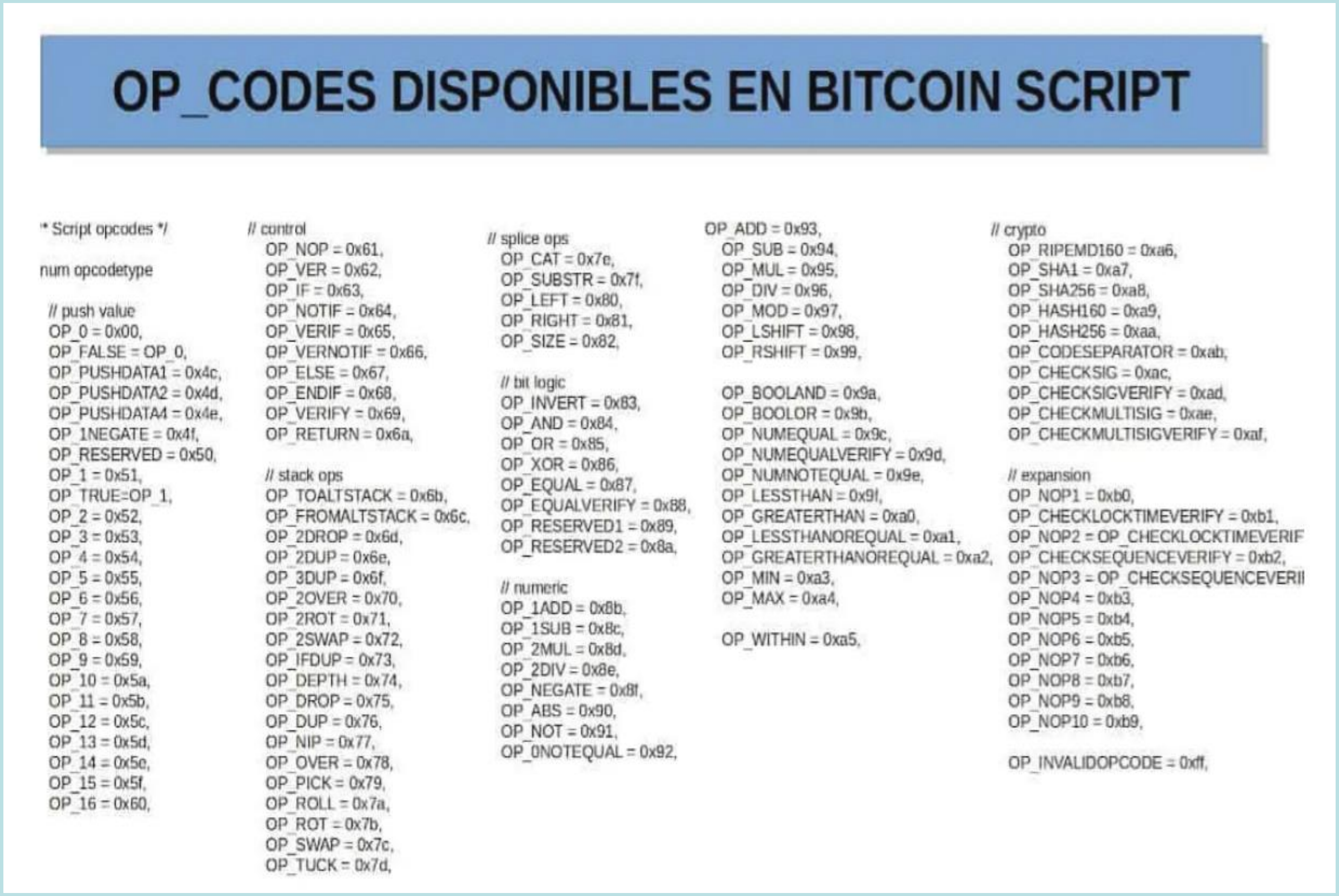
What is an OP_CODE or Operation Code?

- In computing, an OP_CODE (Operation Code, in English), is a portion of a machine language instruction that specifies the operation to be performed.
- Its specification and format will be determined by the **instruction set architecture (ISA)** of the component that processes the instruction.
- Generally, this processing is done by computer hardware (usually a CPU).
- But it can also be software specially prepared to emulate the operation of a ***CPU and process these instructions.***
- In general, a complete machine language instruction contains an **OP_CODE** and, optionally, the specification of **one or more operands,** on which the operation code must act.
- Some operations have implicit operands or none at all.

list of common Script instructions and their functionality

- These **OP_CODES** are the ones that allow you to carry out the different operations in Bitcoin and its transaction scheduling, such as data flow control, constant management, stack management, logical management, arithmetic, time lock, pseudo-words, cryptographic operations, and reserved words.
- You can see a complete and updated list of the different **OP_CODES** directly in the bitcoin code.

list of common Script instructions and their functionality



OP_CODES DISPONIBLES EN BITCOIN SCRIPT

```
* Script opcodes */

num opcodetype

 // push value
 OP_0 = 0x00,
 OP_FALSE = OP_0,
 OP_PUSHDATA1 = 0x4c,
 OP_PUSHDATA2 = 0x4d,
 OP_PUSHDATA4 = 0x4e,
 OP_1NEGATE = 0x4f,
 OP_RESERVED = 0x50,
 OP_1 = 0x51,
 OP_TRUE=OP_1,
 OP_2 = 0x52,
 OP_3 = 0x53,
 OP_4 = 0x54,
 OP_5 = 0x55,
 OP_6 = 0x56,
 OP_7 = 0x57,
 OP_8 = 0x58,
 OP_9 = 0x59,
 OP_10 = 0x5a,
 OP_11 = 0x5b,
 OP_12 = 0x5c,
 OP_13 = 0x5d,
 OP_14 = 0x5e,
 OP_15 = 0x5f,
 OP_16 = 0x60,

// control
  OP_NOP = 0x61,
  OP_VER = 0x62,
  OP_IF = 0x63,
  OP_NOTIF = 0x64,
  OP_VERIF = 0x65,
  OP_VERNOTIF = 0x66,
  OP_ELSE = 0x67,
  OP_ENDIF = 0x68,
  OP_VERIFY = 0x69,
  OP_RETURN = 0x6a,

  // stack ops
  OP_TOALTSTACK = 0x6b,
  OP_FROMALTSTACK = 0x6c,
  OP_2DROP = 0x6d,
  OP_2DUP = 0x6e,
  OP_3DUP = 0x6f,
  OP_2OVER = 0x70,
  OP_2ROT = 0x71,
  OP_2SWAP = 0x72,
  OP_IFDUP = 0x73,
  OP_DEPTH = 0x74,
  OP_DROP = 0x75,
  OP_DUP = 0x76,
  OP_NIP = 0x77,
  OP_OVER = 0x78,
  OP_PICK = 0x79,
  OP_ROLL = 0x7a,
  OP_ROT = 0x7b,
  OP_SWAP = 0x7c,
  OP_TUCK = 0x7d,

// splice ops
 OP_CAT = 0x7e,
 OP_SUBSTR = 0x7f,
 OP_LEFT = 0x80,
 OP_RIGHT = 0x81,
 OP_SIZE = 0x82,

 // bit logic
 OP_INVERT = 0x83,
 OP_AND = 0x84,
 OP_OR = 0x85,
 OP_XOR = 0x86,
 OP_EQUAL = 0x87,
 OP_EQUALVERIFY = 0x88,
 OP_RESERVED1 = 0x89,
 OP_RESERVED2 = 0x8a,

 // numeric
 OP_1ADD = 0x8b,
 OP_1SUB = 0x8c,
 OP_2MUL = 0x8d,
 OP_2DIV = 0x8e,
 OP_NEGATE = 0x8f,
 OP_ABS = 0x90,
 OP_NOT = 0x91,
 OP_0NOTEQUAL = 0x92,

OP_ADD = 0x93,
  OP_SUB = 0x94,
  OP_MUL = 0x95,
  OP_DIV = 0x96,
  OP_MOD = 0x97,
  OP_LSHIFT = 0x98,
  OP_RSHIFT = 0x99,

  OP_BOOLAND = 0x9a,
  OP_BOOLOR = 0x9b,
  OP_NUMEQUAL = 0x9c,
  OP_NUMEQUALVERIFY = 0x9d,
  OP_NUMNOTEQUAL = 0x9e,
  OP_LESSTHAN = 0x9f,
  OP_GREATERTHAN = 0xa0,
  OP_LESSTHANOREQUAL = 0xa1,
  OP_GREATERTHANOREQUAL = 0xa2,
  OP_MIN = 0xa3,
  OP_MAX = 0xa4,

  OP_WITHIN = 0xa5,

// crypto
  OP_RIPEMD160 = 0xa6,
  OP_SHA1 = 0xa7,
  OP_SHA256 = 0xa8,
  OP_HASH160 = 0xa9,
  OP_HASH256 = 0xaa,
  OP_CODESEPARATOR = 0xab,
  OP_CHECKSIG = 0xac,
  OP_CHECKSIGVERIFY = 0xad,
  OP_CHECKMULTISIG = 0xae,
  OP_CHECKMULTISIGVERIFY = 0xaf,

  // expansion
  OP_NOP1 = 0xb0,
  OP_CHECKLOCKTIMEVERIFY = 0xb1,
  OP_NOP2 = OP_CHECKLOCKTIMEVERIF
  OP_CHECKSEQUENCEVERIFY = 0xb2,
  OP_NOP3 = OP_CHECKSEQUENCEVERI
  OP_NOP4 = 0xb3,
  OP_NOP5 = 0xb4,
  OP_NOP6 = 0xb5,
  OP_NOP7 = 0xb6,
  OP_NOP8 = 0xb7,
  OP_NOP9 = 0xb8,
  OP_NOP10 = 0xb9,

  OP_INVALIDOPCODE = 0xff,
```

scritpSig and scriptPubKey, the essential parts of all Bitcoin Script

- In the Bitcoin network, each Bitcoin Script is divided into **two types** of scripts, the **scriptNext** y **scriptPubKey.**
- First, **the scriptSig is the unlock script, which requires a public key and a digital signature**.
- In fact, after detecting various problems in the early versions of the Bitcoin software, signature checks were included.
- Therefore, the system only accepts to carry out transactions if the signatures and their verification comply with a series of established rules that guarantee proper behavior on the network.
- The **second, the scriptPubKey, is the blocking script, which contains a public key hash, also called a Bitcoin address.**
- Bitcoin scripts require **multi-signatures**, that is, the authorization of several users to carry out the transaction.
- In this case, the script is more complicated because it is a much larger operation than the standard **peer-to-peer operation**.
- In fact, the scheduling of Bitcoin transactions as such is stored in this part of the script.

Bitcoin Scripts

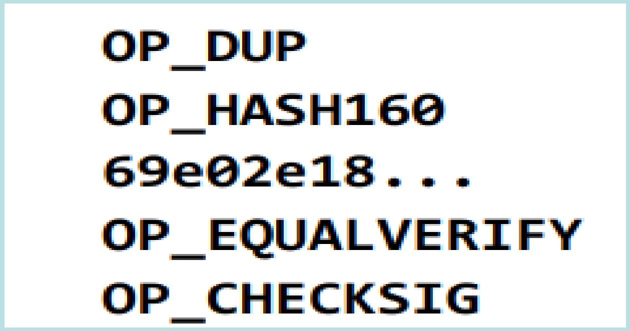
- Each transaction **output** doesn't just specify a **public key**.
- It actually specifies **a script**.

What is a script, and why do we use scripts?

- In this section, we'll study the Bitcoin ***scripting language*** and understand why a **script** is used instead of ***simply assigning a public key***.
- The most common type of transaction in **Bitcoin** is to ***redeem a previous transaction*** output by signing with the correct key.
- In this case, we want the transaction output to say, "**this can be redeemed by signature from the owner of address X.**"

- Recall that an address is a ***hash of a public key.***
- So merely specifying the address X doesn't tell us what the public key is, and doesn't give us a way to check the
- signature!
- So instead the transaction output must say: **"this can be redeemed by a public key that hashes to X,** along with a **signature from the owner of that public key."**
- As we'll see, this is exactly what the most common type of script in Bitcoin says.

<u>**Example**</u> "**Pay-to-PubkeyHash script**", the most common type of output script in Bitcoin



```
OP_DUP
OP_HASH160
69e02e18...
OP_EQUALVERIFY
OP_CHECKSIG
```

Bitcoin Scripts

What happens to this script?

- Who runs it, and how exactly does this sequence of instructions enforce the above statement?
- The secret is that the **inputs** also contain ***scripts instead of signatures.***
- To validate that a **transaction redeems a previous transaction output correctly**,
- We combine the new transaction's **input script** and the earlier transaction's **output script.**
- We simply **concatenate them**, and the **resulting script must run successfully in order for the transaction to be valid**.
- These **two scripts** are called ***scriptPubKey*** and ***scriptSig*** because the output script just specifies a **public key** (or an address to which the public key hashes), and the **input script** specifies a **signature** with that **public key**.
- The combined script can be seen below in Figure.

Bitcoin Scripts

```
<sig>
<pubKey>
--------------
OP_DUP
OP_HASH160
<pubKeyHash?>
OP_EQUALVERIFY
OP_CHECKSIG
```

- To check if a transaction correctly redeems an output, we create a combined script by appending the scriptPubKey of the referenced output transaction (bottom) to the scriptSig of the redeeming transaction (top).
- Notice that <pubKeyHash?> contains a '?'.
- We use this notation to indicate that we will later **check to confirm that this is equal to the hash of the public key** provided in
- the redeeming script.

Bitcoin scripting language.

- The scripting language was built specifically for Bitcoin and is just called 'Script' or the Bitcoin scripting language.
- It has many similarities to a language called **Forth**, which is an old, simple, stack-based, programming language.
- The key design goals for Script were to have something simple and compact, yet with native support for cryptographic operations.

For Example,

- There are special-purpose instructions to compute **hash functions** and to **compute and verify signatures**.
- The scripting language is **stack-based**.
- This means that **every instruction** is executed **exactly once**, in a **linear manner**.
- In particular, **there are no loops** in the Bitcoin scripting language.
- So the number of instructions in the script gives us an upper bound on how long it might take to run and how much memory it could use.

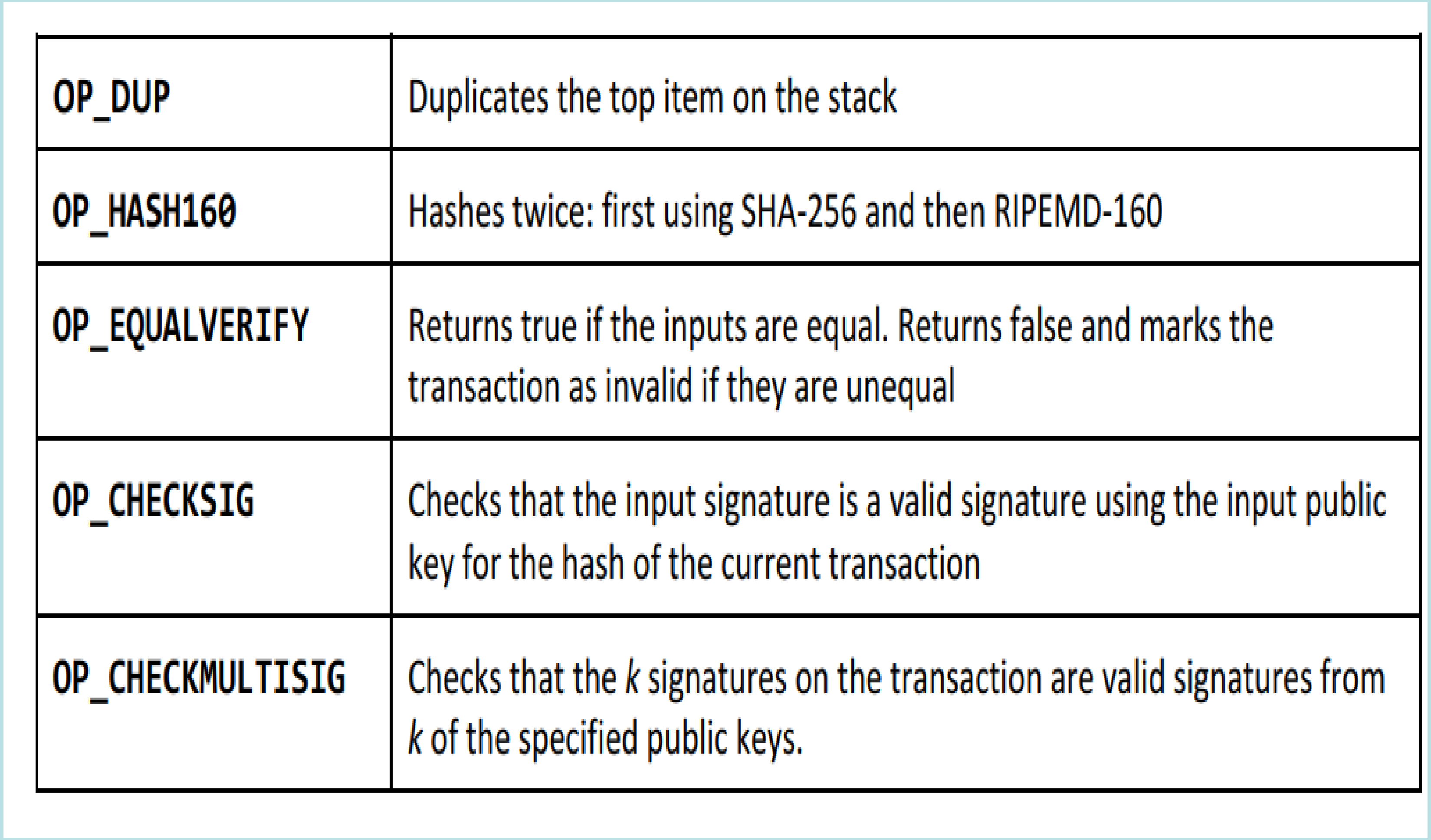
- The language is not **Turing-complete**, which means that it doesn't have the ***ability to compute arbitrarily powerful functions***.
- And this is by design — **miners** have to run these scripts, which are submitted by arbitrary(random) participants in the **network.**
- We don't want to give them the power to submit a script that might have an **infinite loop.**
- There are only **two possible outcomes** when a Bitcoin script is executed.
- It either executes successfully with **no errors,** in which case the **transaction is valid**. Or, if there's **any error** while the script is executing, the whole transaction will be **invalid** and shouldn't be accepted into the blockchain.
- The Bitcoin scripting language is very small.
- There's only room for 256 instructions because each one is represented by **one byte**.
- Of those 256, 15 are currently disabled, and 75 are reserved.
- The reserved instruction codes haven't been assigned any specific

- Many of the basic instructions are those you'd expect to be in any programming language.
- There's basic arithmetic, basic logic — like 'if' and 'then' —, throwing errors, not throwing errors, and returning early.
- Finally, there are crypto instructions which include hash functions, instructions for **signature verification**, as well as a special and important instruction **called CHECKMULTISIG** that lets you check *multiple signatures* with one instruction.
- Figure 3.6 lists some of the most common instructions in the **Bitcoin scripting language**.
- The CHECKMULTISIG instruction requires specifying **n** public keys, and a parameter **t** , for a threshold.
- For this instruction to execute **validly,** there have to be at least **t** signatures from **t** out of **n** of those public keys that **are valid.**
- We'll show some examples of what you'd use multi signatures for in the next section, but it should be immediately clear this is quite a powerful primitive.

- We can express in a compact way the concept that t out of n specified entities must sign in order for the transaction to be valid.
- Incidentally, there's a bug in the multi-signature implementation, and it's been there all along.
- The CHECKMULTISIG instruction **pops** an extra data value off the stack and ignores it.
- This is just a quirk of **the** Bitcoin language and one has to deal with it by putting an extra **dummy variable** onto the stack.
- The bug was in the original implementation, and the costs of fixing it are **much higher than the damage it causes**.
- Section 3.5. At this point, this bug is considered a feature in Bitcoin, in that it's not going away.

list of common Script instructions and their functionality

- **Figure 3.6 below lists common Script instructions and their functionality.**



OP_DUP	Duplicates the top item on the stack
OP_HASH160	Hashes twice: first using SHA-256 and then RIPEMD-160
OP_EQUALVERIFY	Returns true if the inputs are equal. Returns false and marks the transaction as invalid if they are unequal
OP_CHECKSIG	Checks that the input signature is a valid signature using the input public key for the hash of the current transaction
OP_CHECKMULTISIG	Checks that the *k* signatures on the transaction are valid signatures from *k* of the specified public keys.

list of common Script instructions and their functionality

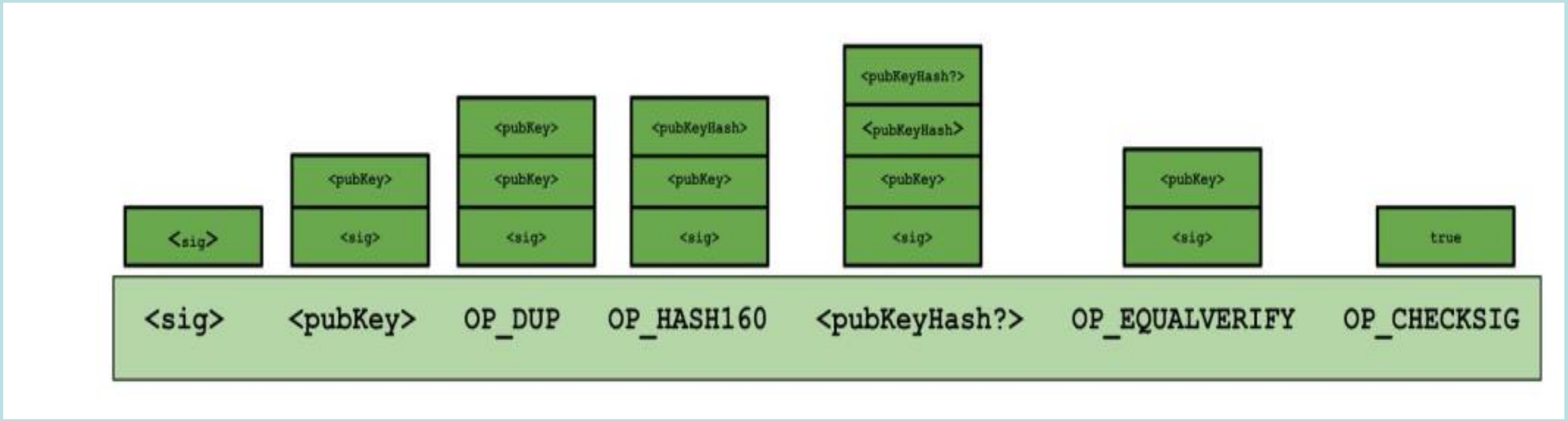
Executing a script.

- To execute a script in a stack-based programming language, all we'll need is a stack that we can **push data** to and **pop data from**. We won't need any other memory or variables. That's
- what makes it so computationally simple? There are two types of instructions: **data instructions and opcodes**.
- When a data instruction appears in a script, that data is simply **pushed** onto the top of the stack.
- **Opcodes,** on the other hand, perform some function, often taking as input data that is on top of the stack.
- Now let's look at how the Bitcoin script in Figure 3.5 is executed. Refer to Figure 3.7, where we show the state of the stack after each instruction.
- The first **two instructions** in this script are data instructions — the signature and the public key used to verify that signature — specified in the **scriptSig** component of a transaction input in the redeeming transaction

list of common Script instructions and their functionality

- As we mentioned, when we see **a data instruction,** we just **push** it onto the stack.
- The rest of the script was specified in the **scriptPubKey** component of a transaction output in the referenced transaction.
- First we have the duplicate instruction, OP_DUP, so we just push a copy of the public key onto the top of the stack.
- The next instruction is OP_HASH160, which tells us to **pop the top value,** compute its cryptographic hash, and push the result onto the top of the stack.
- When this instruction finishes executing, we will have replaced the **public key on** the top of the stack with its hash.

list of common Script instructions and their functionality



- The above fig shows, Execution of a Bitcoin script.
- On the bottom, we show the instruction in the script.
- Data instructions are denoted with surrounding angle brackets, whereas opcodes begin with “OP_”. On the top, we show the **stack** just after that **instruction has been executed.**

list of common Script instructions and their functionality

- Next, we're going to do **one more push** of data **onto the stack.**
- Recall that this data was specified by the sender of the **referenced transaction.**
- It is the **hash of a public key** that the sender specified; the corresponding private key must be used to generate the signature to redeem these coins.
- At this point, there are **two values** at the **top of the stack**.
- There is the hash of the **public key**, as specified by the sender, and the hash of the public key that was used by the recipient when trying to **claim the coins.**

list of common Script instructions and their functionality

- At this point we'll run the **EQUALVERIFY** command, which checks that **the two values** at the top of the **stack are equal.**
- If they aren't, **an error** will be thrown, and the script will stop executing.
- But in **our example**, we'll assume that they're equal, that is, that the recipient of the coins used the **correct public key.**
- That instruction will consume those **two data** items that are at the **top of the stack**
- And the stack now contains **two items** — **a signature** and the **public key.**

list of common Script instructions and their functionality

- We've already checked that this public key is that the referenced transaction specified, and now we have to check if the ***signature is valid.***
- This is a great example of where the Bitcoin scripting language is built with cryptography.
- Even though it's fairly simple language in terms of logic, there are some quite powerful instructions in there, like this **"OP_CHECKSIG" instruction.**
- This single instruction pops those **two values** off of the stack and does the entire signature verification in one go.

list of common Script instructions and their functionality

What is this a signature of? What is the input to the signature function?

- It turns out there's only one thing you can sign in Bitcoin — an entire transaction.
- So the "CHECKSIG" instruction ***pops the two*** values, the public key, and signature, off the stack, and verifies that is a valid signature for the entire transaction using that public key.
- Now we've executed every instruction in the script, and there's nothing **left on the stack.**
- Provided there weren't any errors, the output of this script will simply be true indicating that the **transaction is valid.**

Applications of Bitcoin scripts

Escrow transactions

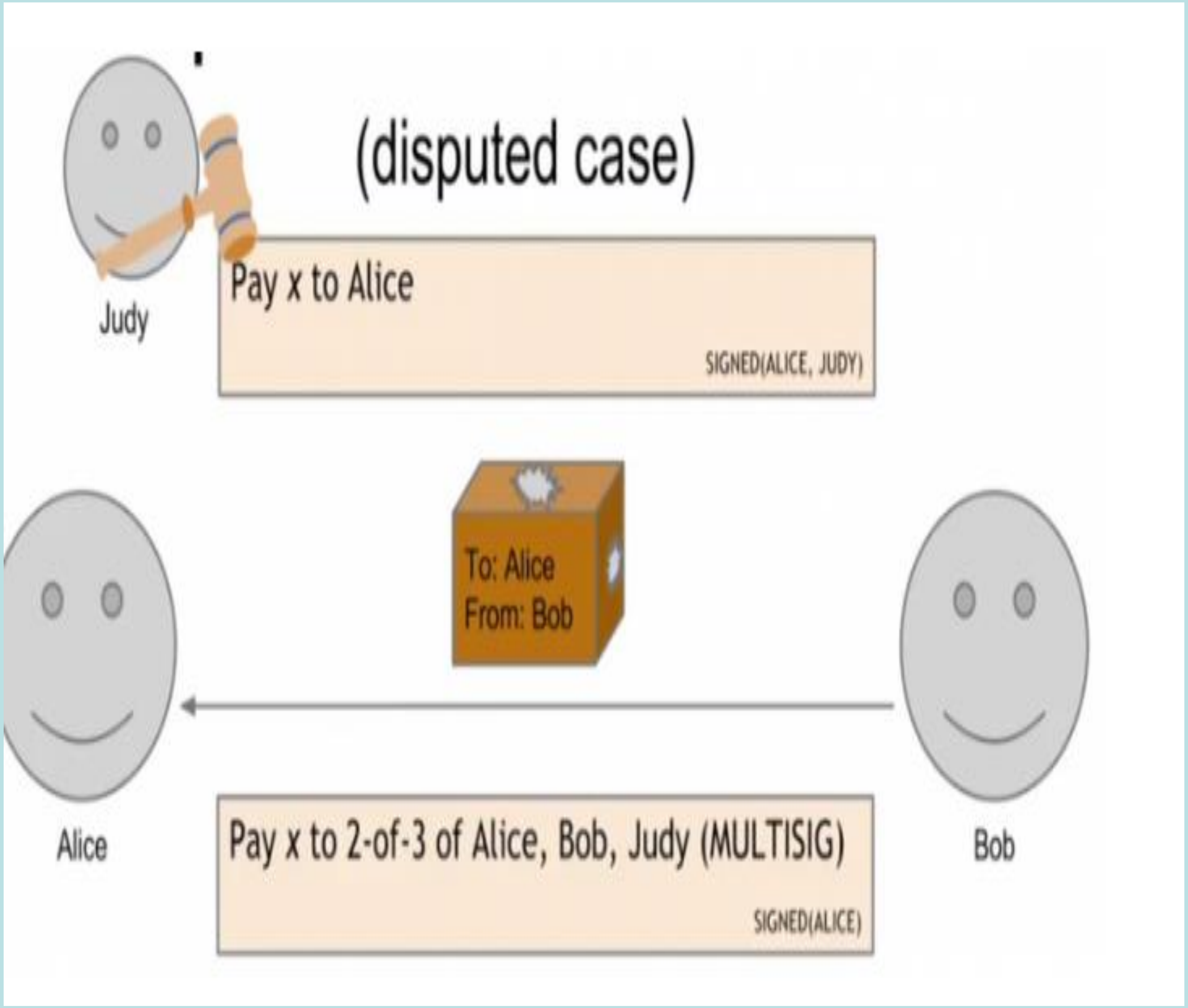
What is an escrow transaction in Bitcoin?

Bitcoin escrow is **a service designed to act as an intermediary between two parties involved in a transaction**. In a typical transaction, one party sends the Bitcoin to the escrow service, where it is held until both parties agree to the terms of the transaction.

What is escrow in blockchain?

Before making a transaction, **tokens are transferred to a third-party smart contract** called escrow. The escrow holds the deposited tokens until the payment conditions are satisfied. Context. The parties involved in the transaction need to ensure that both the agreed product/service is delivered and payment is made.

- Escrow transactions can be implemented by simply using **MULTISIG**.
- Alice doesn't send the money directly to Bob but instead creates a MULTISIG transaction requiring two or three people to sign to redeem the coins.
- Three people will be Alice, Bob, and some third-party judge, Judy, who will come into play in case of **any dispute.**
- Alice creates a **2-of-3 MULTISIG transaction** that sends some coins she owns and specifies that they can be spent if any two of Alice, Bob, and Judy sign.
- This transaction is included in the blockchain, and at this point, these coins are held in escrow between Alice, Bob, and Judy, such that any two of them can specify where the coins should go.
- At this point, Bob is convinced that sending the goods over to Alice is safe, so he'll mail them or deliver them physically.
- Now in the normal case, Alice and Bob are both honest. So, Bob will send over the goods that Alice is expecting, and when Alice receives the goods, Alice and Bob both sign a transaction redeeming the funds from escrow, and sending them to Bob.
- Notice that in this case where Alice and Bob are honest, Judy never had to get involved.
- There was no dispute, and Alice's and Bob's signatures met the 2-of-requirement of the **MULTISIG** transaction**.**
- So in the normal case, this isn't that much less efficient than Alice just sending Bob the money.
- It requires just one extra transaction on the blockchain.


Judy
(disputed case)
Pay x to Alice
SIGNED(ALICE, JUDY)
To: Alice
From: Bob
Alice
Bob
Pay x to 2-of-3 of Alice, Bob, Judy (MULTISIG)
SIGNED(ALICE)

Applications of Bitcoin scripts

Green Address

GreenAddress is **a multi-platform Bitcoin Wallet service**. This wallet provides security, privacy, and ease of use through **multi-platform mobility**. Another GreenAddress app GreenBits is available for Android which is designed for the **native** Android experience, and includes more advanced **Bitcoin validation features.**

Applications of Bitcoin scripts -Green Address -Example

- Imagine that Alice wants to pay Bob, who's offline. Bob can't check the blockchain to see if the transaction that Alice is sending is valid. This can happen for any reason.
- For example if Bob doesn't have **time to connect and check** or doesn't have a connection.
- Normally a transaction is valid when its block is followed by other **six blocks.** This can take **up to an hour.**
- To solve the problem of the recipient not being able to access the blockchain, it is necessary to introduce a **third party: the bank.**
- Alice can ask her bank to transfer money to Bob.
- They will deduct some money from Alice's account and make a transaction to Bob from one of their green addresses.
- The money to Bob will come directly from a bank-controlled address, with a guarantee of no double-spending.
- If Bob trusts the bank, he can accept the transaction as soon as he receives it. The money will eventually be his when it's confirmed in the blockchain.

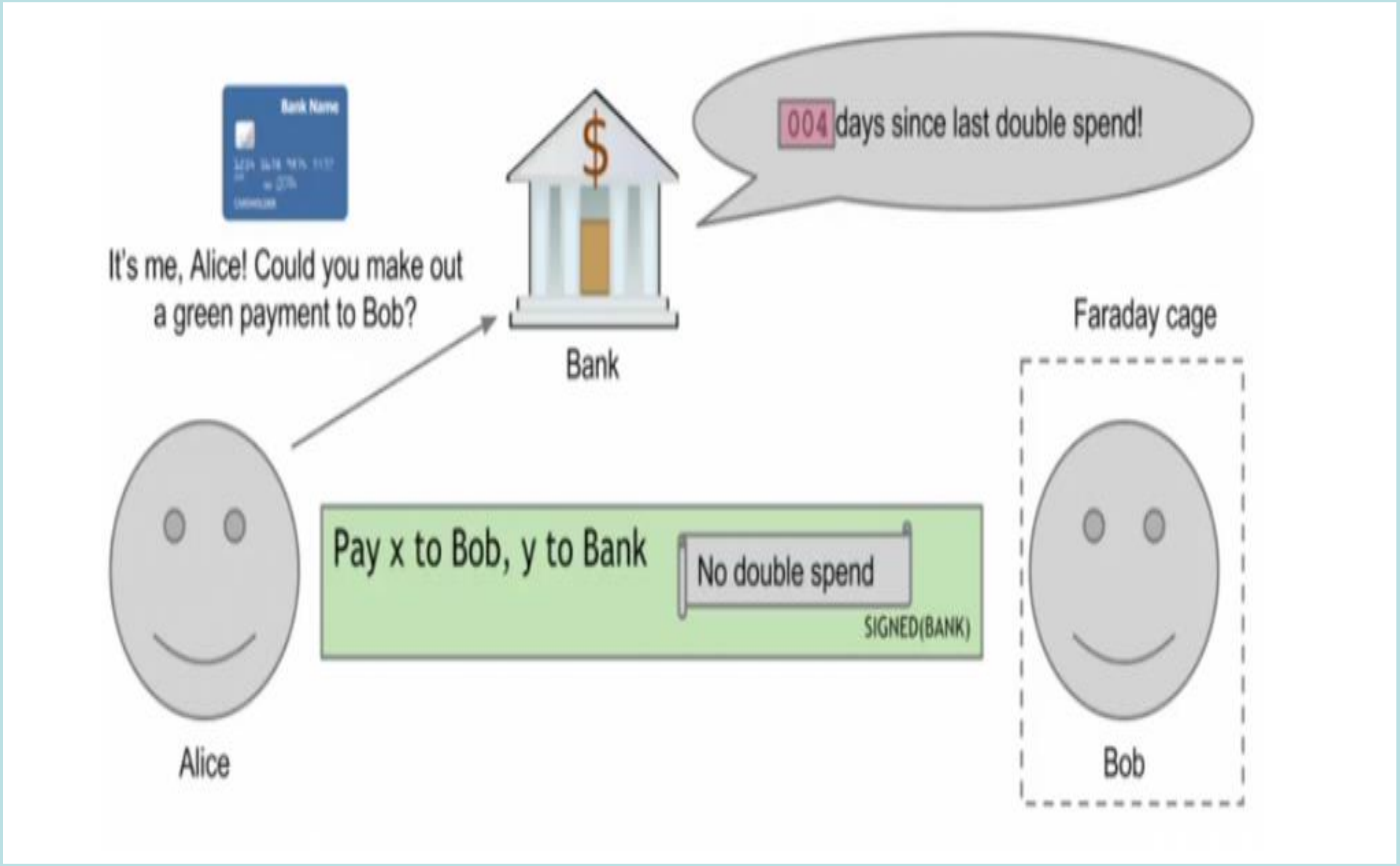
Applications of Bitcoin scripts -**Green Address -Example**

- This feature isn't based on the Bitcoin system, but is a real-world guarantee, so the bank must be trustable.
- If the bank ever does double spending, its system is going to collapse quickly.
- This has already happened to **two famous services** that implemented **green addresses:**

1. Instawallet
2. Mt. Gox

- For this reason, green addresses **aren't used as much in Bitcoin** as when they were first proposed. It is necessary to put too much trust in the bank.

Applications of Bitcoin scripts -Green Address -Example


Bank Name
It's me, Alice! Could you make out a green payment to Bob?
Bank
004 days since last double spend!
Faraday cage
Pay x to Bob, y to Bank
No double spend
SIGNED(BANK)
Alice
Bob

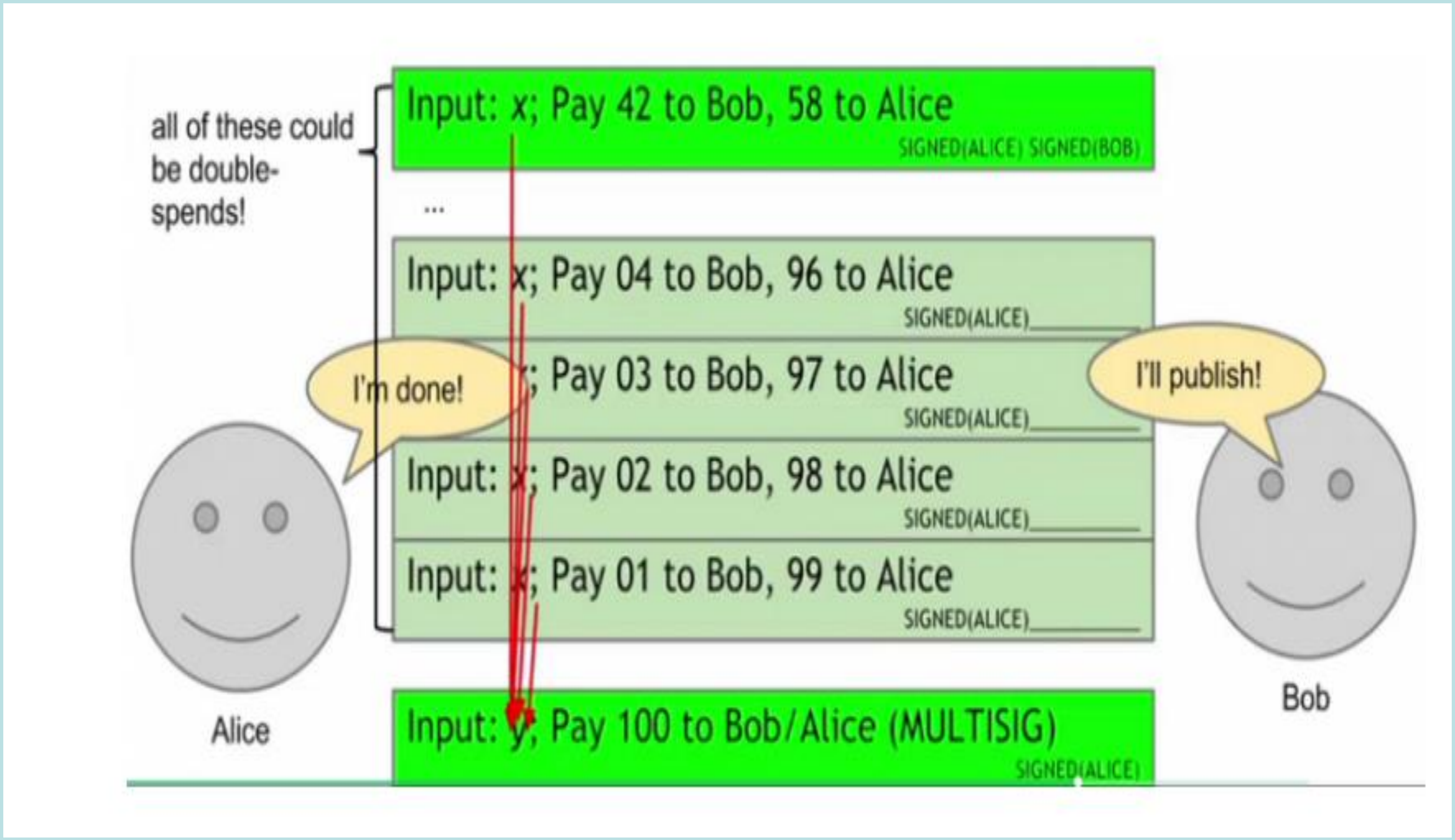
Applications of Bitcoin scripts -**Efficient micro-payments**

- A third example of Bitcoin scripts is a way to do **efficient micro-payments**.
- Alice is a customer who wants to continually pay Bob small amounts of money for some services that Bob provides.
- **For example**, Bob may be Alice's wireless service provider and requires her to pay a small fee for **every minute** that she talks on her phone.
- Creating a Bitcoin transaction for **every minute** that Alice speaks on the phone **won't work.**
- That will **create too many transactions**, and the transaction fees add up.
- If the value of each one of these transactions is on the order of what the transaction fees are, Alice is going to be paying quite a high cost to do this.
- A nice solution would be to combine all these small payments into one big payment at the end. To do this we can:
- Start with a **MULTISIG transaction** that pays the maximum amount Alice would ever need to spend, that requires both Alice's and Bob's signatures to release the coins
- After the **first minute** of conversation, Alice signs the transaction sending one coin to Bob and returning the rest to herself.
- After **another minute**, Alice signs another transaction, paying two coins to Bob and the rest to herself. At this point, Bob hasn't signed anything yet.

Applications of Bitcoin scripts -**Efficient micro-payments**

- Alice repeats this procedure every minute of usage. These transactions aren't published on the blockchain, since Bob’s signature is missing.
- When Bob wants to **get his money,** he can sign the last transaction and publish it on the blockchain. He will receive the money **he deserves**, and the remaining will be sent back to Alice. The **other transactions** will **never be inserted into** the blockchain.
- It is impossible to redeem two different transactions generated by Alice. since they are **all technically double spends** of the same beginning transaction.
- If **both parties** are operating normally, Bob will never sign any transaction but the last one, so the blockchain won't actually see any attempt of **double-spending**.

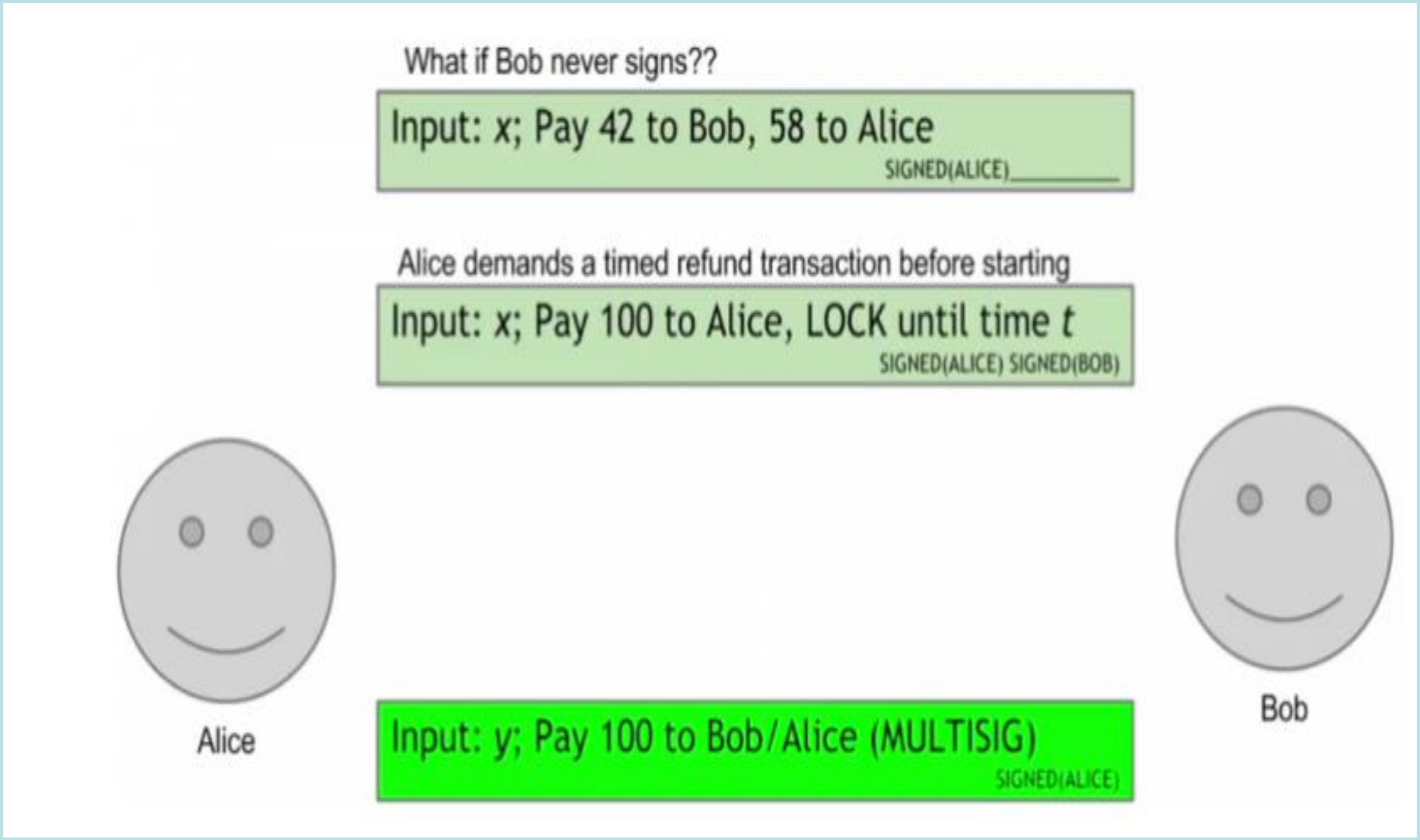
Applications of Bitcoin scripts -**Efficient micro-payments**


all of these could be double-spends!
Input: x; Pay 42 to Bob, 58 to Alice
SIGNED(ALICE) SIGNED(BOB)
...
Input: x; Pay 04 to Bob, 96 to Alice
SIGNED(ALICE)
; Pay 03 to Bob, 97 to Alice
SIGNED(ALICE)
Input: x; Pay 02 to Bob, 98 to Alice
SIGNED(ALICE)
Input: x; Pay 01 to Bob, 99 to Alice
SIGNED(ALICE)
Input: y; Pay 100 to Bob/Alice (MULTISIG)
SIGNED(ALICE)
I'm done!
I'll publish!
Alice
Bob

Applications of Bitcoin scripts -Lock time

- A problem of the micro-payment protocol could be that Bob will never sign any of the transactions, so all the money that Alice first sent to the **multisign address** remains blocked. How can Alice have her money back? She can use another feature called Lock Time.
- Before the micro-payment protocol starts, Alice and Bob will both sign a transaction that refunds all of Alice's money back to her but is **locked until some time in the future.**
- So before Alice signs the **first transaction** for the first minute of service, she requires **this refund transaction from Bob**. If a certain time **T** is reached and Bob hasn't signed any of the small transactions that Alice sent, she can publish this transaction **which refunds all of the money directly to her.**
- To do this, we will use the **LOCK_TIME** parameter in the metadata of Bitcoin transactions. It is possible to specify a value different from **0 for the lock time**, which tells the **miners *not to publish the transaction until some point in time***, based on the **timestamps that are put** in blocks.
- With this feature Alice knows that she can **get her money back** if Bob never signs the transactions.

Applications of Bitcoin scripts -Lock time


What if Bob never signs??
Input: x; Pay 42 to Bob, 58 to Alice
SIGNED(ALICE)
Alice demands a timed refund transaction before starting
Input: x; Pay 100 to Alice, LOCK until time t
SIGNED(ALICE) SIGNED(BOB)
Alice
Bob
Input: y; Pay 100 to Bob/Alice (MULTISIG)
SIGNED(ALICE)

Bitcoin Blocks

What are the blocks in Bitcoin?

- A Block refers to a set of Bitcoin transactions from a **certain time period**
- Blocks are **"stacked"** on top of each other in such a way that one block depends on the previous.
- In this manner, a chain of blocks is created, and thus we come to the term "blockchain".
- We've looked at how individual transactions are constructed and redeemed. But as we saw in **chapter 2,** transactions are grouped together **into blocks**.

Why is this?

- Basically, it's an optimization.
- If miners had **to come to a consensus** on each transaction individually, the rate at which new transactions could be accepted by the system would be **much lower.** Also, a **hash chain of blocks** is much shorter than a hash chain of transactions would be, since a large number of transactions can be put into each block. This will make it much more efficient to verify the blockchain **data structure**

Bitcoin Blocks

Why the transactions are grouped together into blocks?

There are a couple of reasons:

- a block creates a single unit of work for miners, that is bigger than the individual transaction size. Doing hash and adding metadata for every transaction would be too much expensive.
- The presence of blocks makes **the hash_chain shorter** since we only need one block for a large number of transactions. So it will also be easier to verify the blockchain data structure.

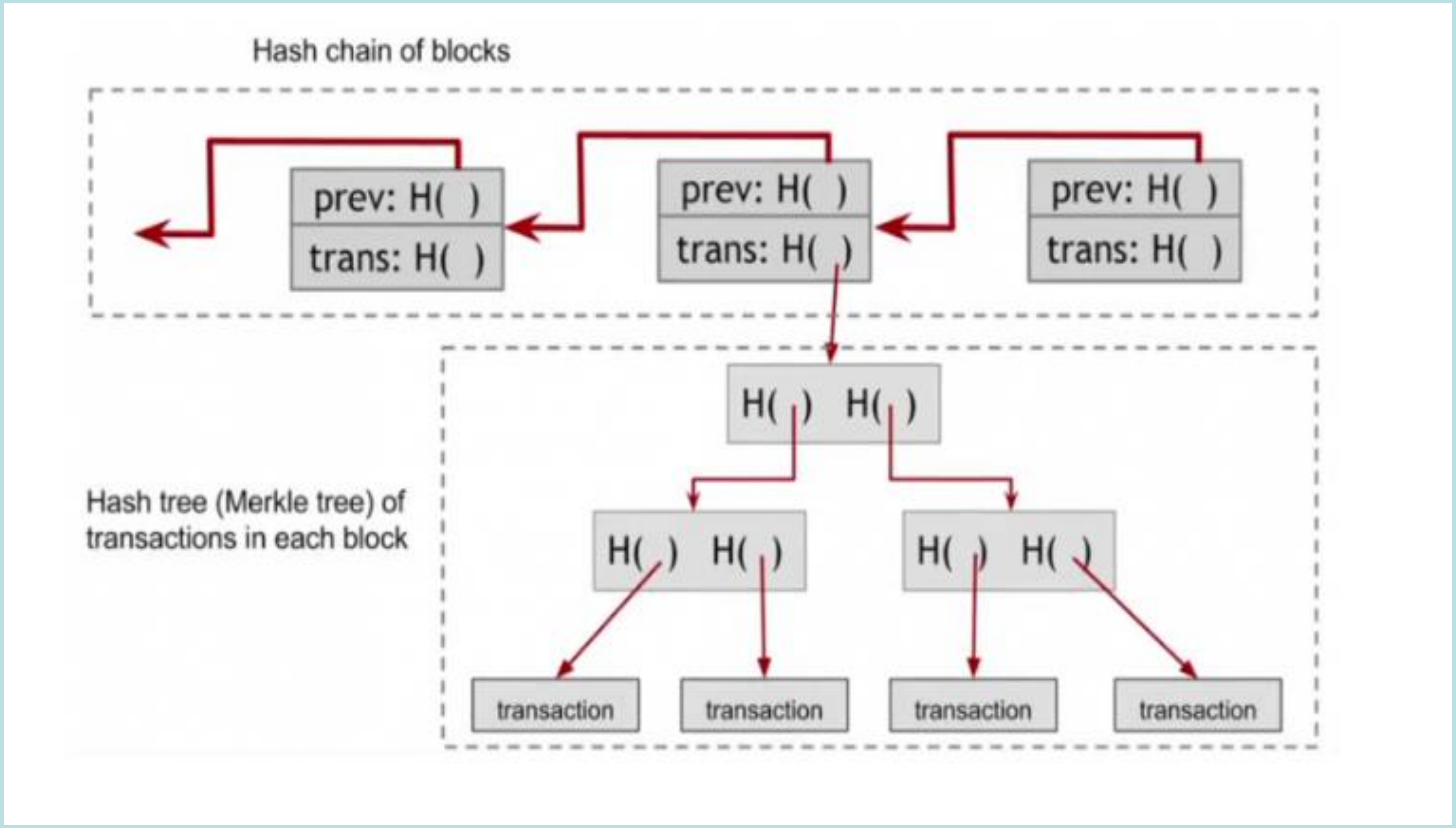
Blockchain data structure

- The blockchain is the combination of **two different hash-based data structures:**

1. hash pointer between one block and the previous one
2. inside every block the transactions are organized with a **Merkle tree** (hash tree).

- To prove that a transaction is inside one block will be enough to provide the path inside the tree. So the check will be **logarithmic in size**

Bitcoin Blocks



Block structure

A block consists of:

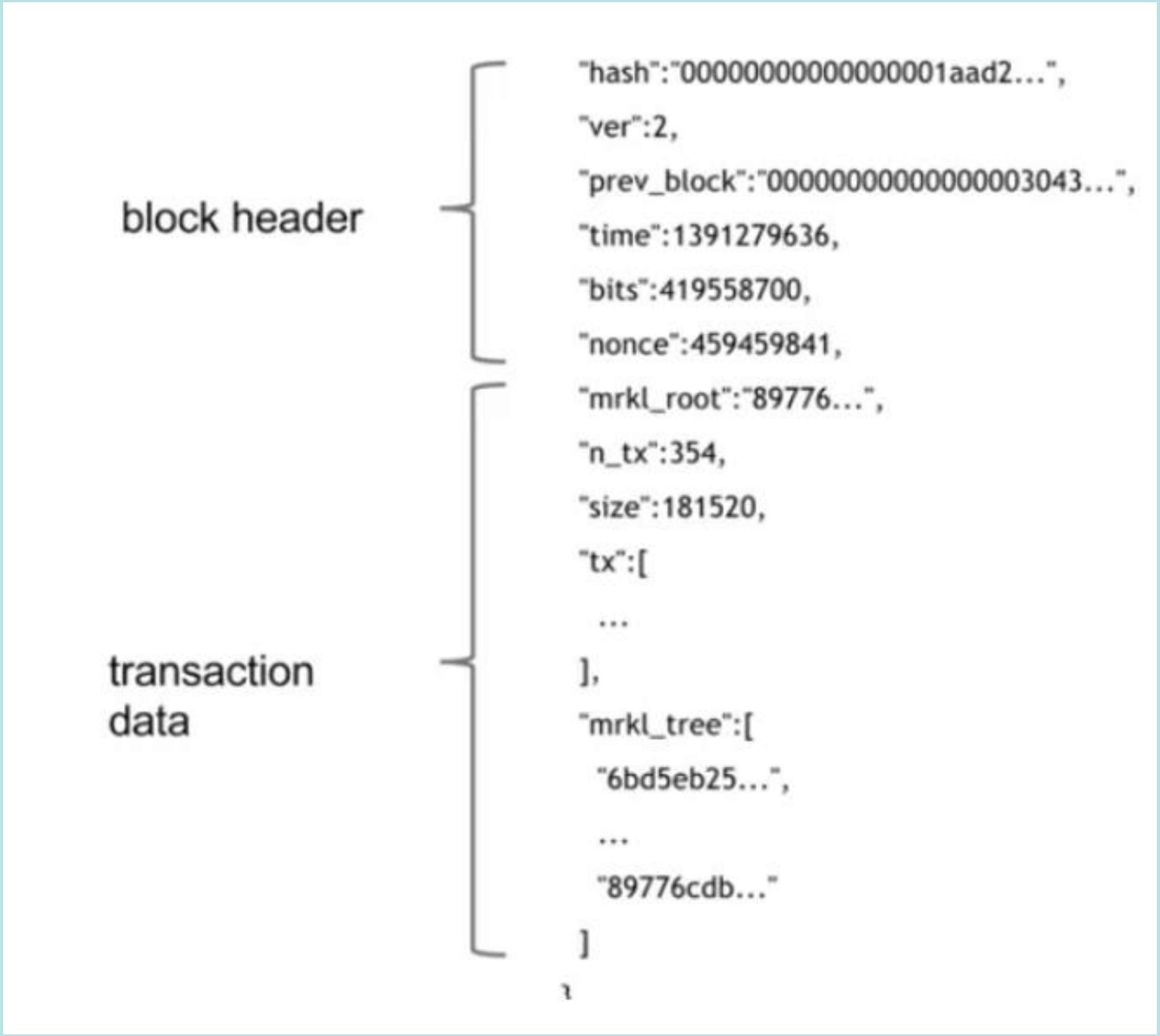
block header:

- contains all metadata for the block.
- The header is the most important part of the **mining puzzle** since we have seen that it has to start with a **large number of zeros** to be inserted into the blockchain.
- The header also contains the **nonce that the miners** can change to solve the puzzle and an indication of how difficult was to find the block.
- Finally, there's a **reference to the block body**, namely the **hash of the Merkle tree root**. So the header contains all the information that is **hashed** during **mining.**

Merkle Tree of transactions:

- A long list of transactions organized **inside this tree.**
- Inside this tree, there's a special transaction: the coinbase transaction.
- This is where the creation of **new coins and Bitcoin happens**.
- It almost looks like a normal transaction, with a value corresponding with the current Bitcoin reward, plus all the block **transaction fees**.
- The difference is that there's **no reference to an output of a previous transaction** since it's a creation of **new coins.**
- There's also a special **coinbase parameter,** that can be populated with some arbitrary data. There are no actual limits on what ***miners can put in there***.

Block structure



```
"hash":"0000000000000001aad2...",
"ver":2,
"prev_block":"00000000000000003043...",
"time":1391279636,
"bits":419558700,
"nonce":459459841,
"mrkl_root":"89776...",
"n_tx":354,
"size":181520,
"tx":[
  ...
],
"mrkl_tree":[
  "6bd5eb25...",
  ...
  "89776cdb..."
]
}
```

Block structure

Block display

- The best way to understand blocks structure better is to view them on websites that provide this feature, for example, **blockchain.info** and many others that let:
  - ✓ check the graph of transactions
  - ✓ see which transactions redeem which other transactions
  - ✓ Look for transactions with complicated scripts
  - ✓ check block structure and the link between blocks

Bitcoin Network

- Here we will see further details about Bitcoin Network, its structure, and ***how transactions can be inserted***.

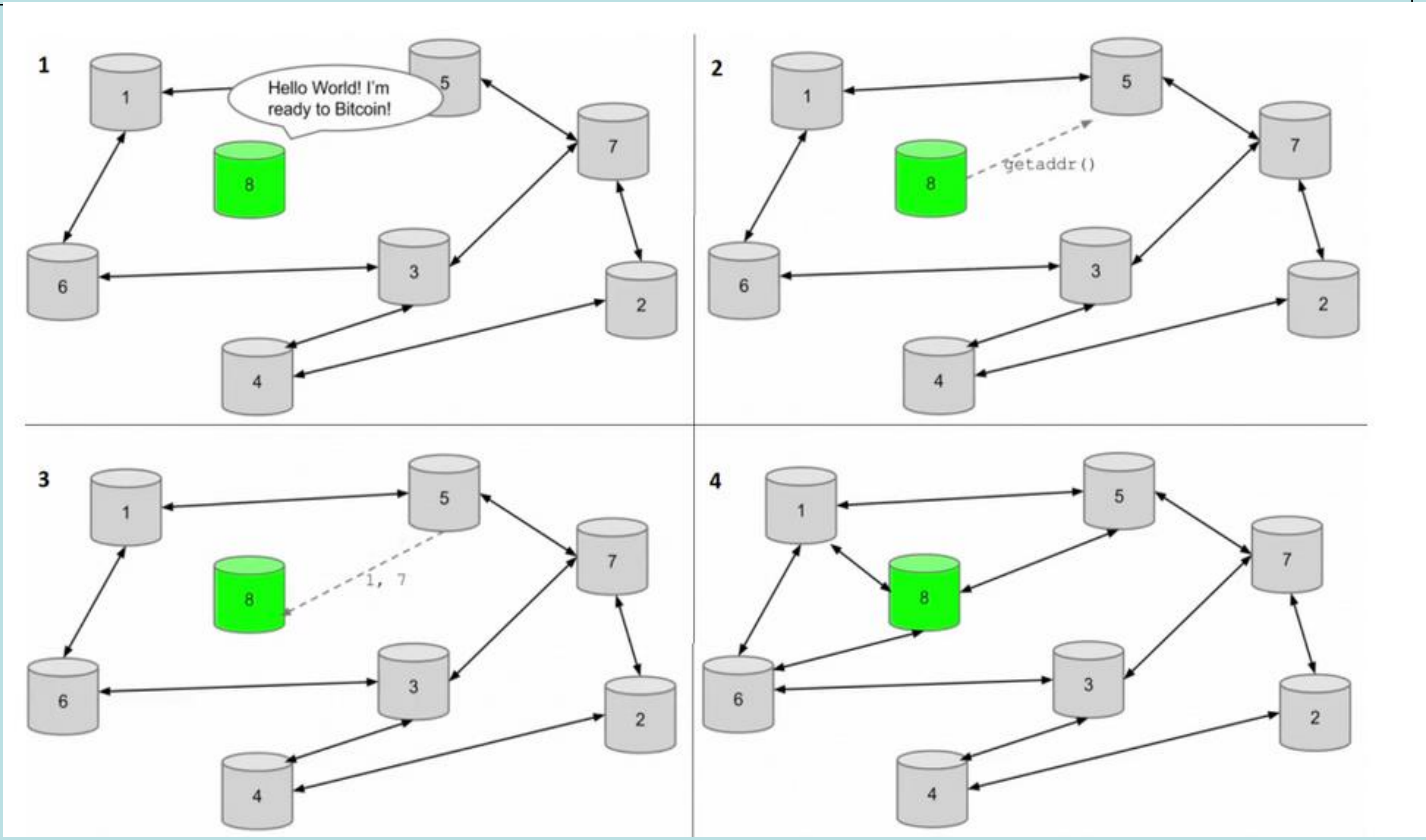
What is a Bitcoin Network?

- It's a peer-to-peer network, where **all nodes are equal.** There's no hierarchy, centralization, special nodes, or master nodes.
- It runs over **TCP** and has a random topology: nodes peered randomly with other nodes. **And new nodes can come at any time.**
- It is possible to download the Bitcoin client today, and start a new Bitcoin node with the **same rights** and capabilities as every other existing node.
- The network is **very dynamic,** nodes are coming and going all the time. Although there's no explicit way to leave the network, if a node doesn't communicate for three hours, other nodes start to forget it.

Example

- A small network of **seven nodes**, connected in a random fashion.
- If a new node wants to join the network, it sends a message to one existing node that's already on the network, called its **seed node**.
- The message will say **"tell me all the peers that you have".** Then the new node proceeds to ask the same to the nodes connected to the **first one,** and so on.
- At the end of the iteration it will have a list of peers to make **connections with**. The new node chooses a number of peers to connect to from the list and becomes a full operating node.

Bitcoin Network

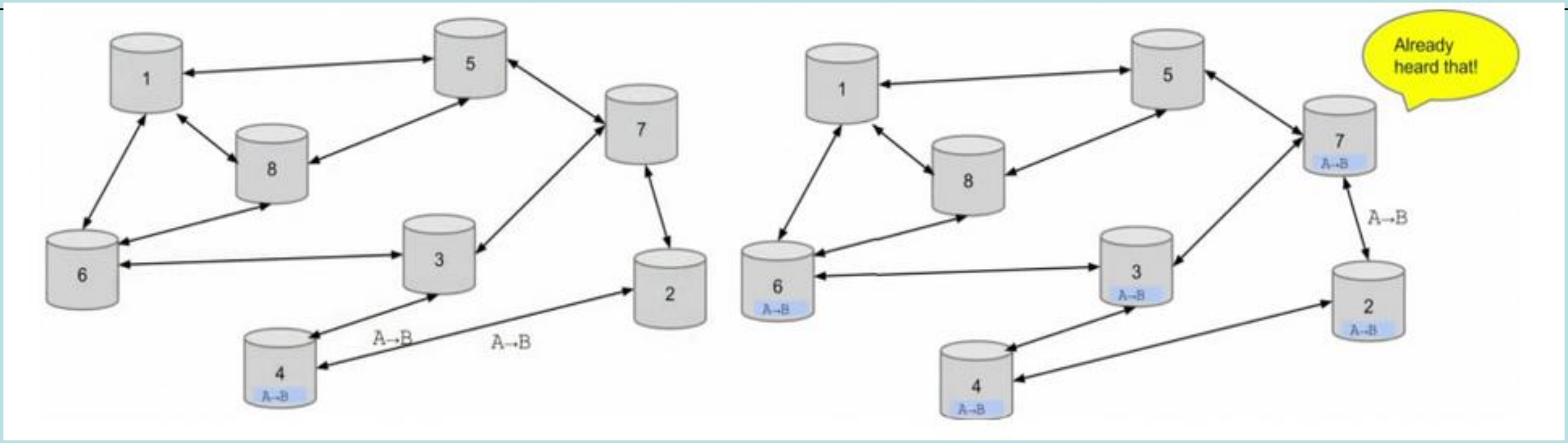


Bitcoin Network

Network purposes

- The network maintains the blockchain, and to publish a transaction the entire **network needs to hear about it.** There's a simple **flooding algorithm to make this happen**, called **gossip protocol.**
- Let's use again our small network for example purposes. Suppose that node **4** hears about a **new transaction.**
- To **spread gossip**, someone tries to tell the news to many people as **he can.**
- **Node four** will do the same, notifying the transaction to all the nodes it is connected to (nodes 2 and 3).
- Each node maintains a list of all transactions he's heard about that haven't been put into the blockchain, yet.
- The **two nodes** will **add the new transaction to their list** and then decide whether to forward it to their **connected nodes or not.**
- Of course, if a node receives a notification of a transaction he's already heard about, he won't spread the transaction again.
- So, when all the nodes know the transaction, **the spread process**

Bitcoin Network



- Besides checking double-heard transactions, a node can decide not to propagate new transactions he's never heard about before.
- In fact, he checks if a transaction is valid or not running the validation script we've seen before.
- He will refuse transactions that try to redeem coins already spent or nonstandard transaction formats.
- The double spending attempt can be identified even before one of the transactions is inserted into the blockchain.
- If a node receives **two unconfirmed transactions** that try to spend the same money, **he won't spread the second one**.

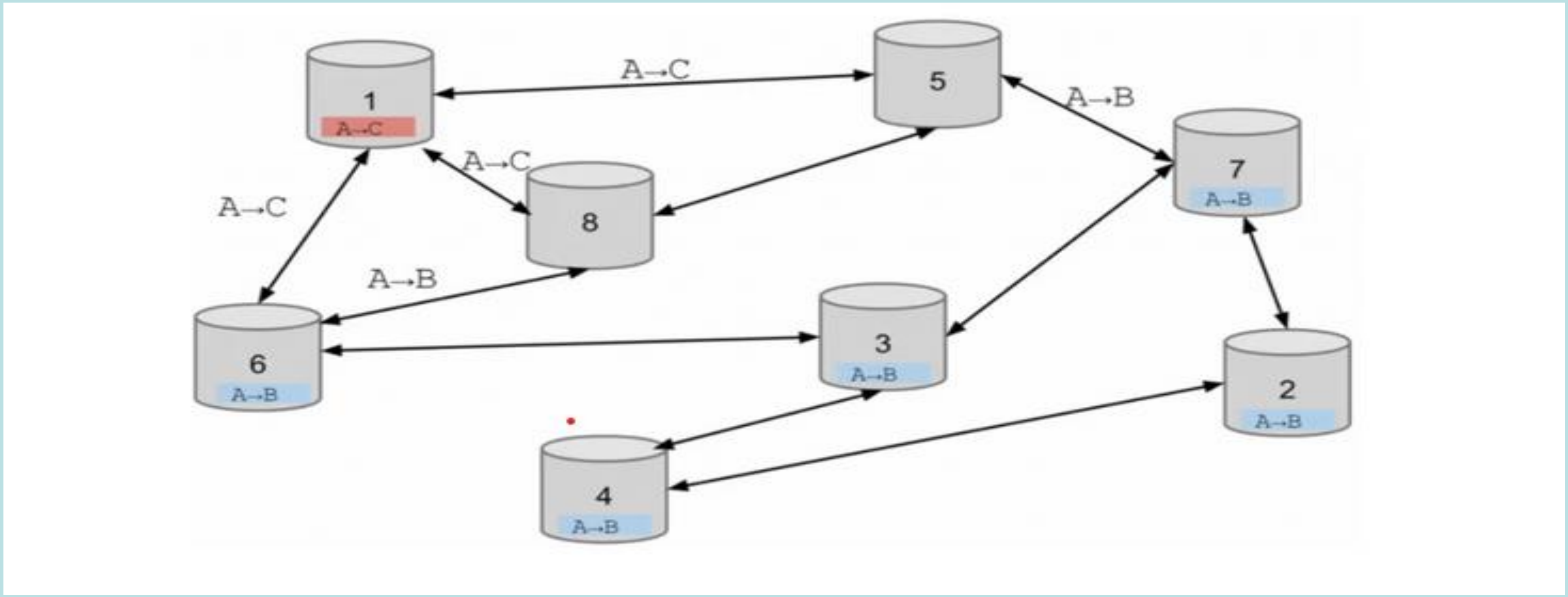
Mining logic

- The Bitcoin network is peer-to-peer and anybody can join, there is always the possibility of a node not following this exact protocol.
- It can forward double spends, nonstandard and **invalid transactions**. That’s the reason why it’s important that every node **repeats the checks**.
- In addition, it is possible that when a transaction is notified to a node, it isn’t aware of some other transactions that are already known by other nodes on the network.

Example

- ***a double spending trial*** where Alice tries to pay Bob and Charlie with the same money.
- Maybe the first node that receives A -> C transaction isn’t aware of the other A -> B transactions that have been transmitted a few moments earlier on the other side of the network.
- So **node 1** considers it **valid and spreads it to its neighbors** (i.e. **node 6** in the image).
- Node 6 has already received transaction A -> B, so refuses the new one and keeps the previous one.
- The network in this case can be divided **in two**, but that’s temporary.
- When one of these two transaction is put into a block, other nodes will see it and consider invalid the other one they’re holding, so they’ll drop it.

Bitcoin Network



- When there’s a similar situation, nodes usually keep the **first transaction** they hear about unless it becomes **clearly invalid.**
- But there’s no authority that forces them to follow this behavior, so every node is **free to implement any other logic it likes**.

Bitcoin Network

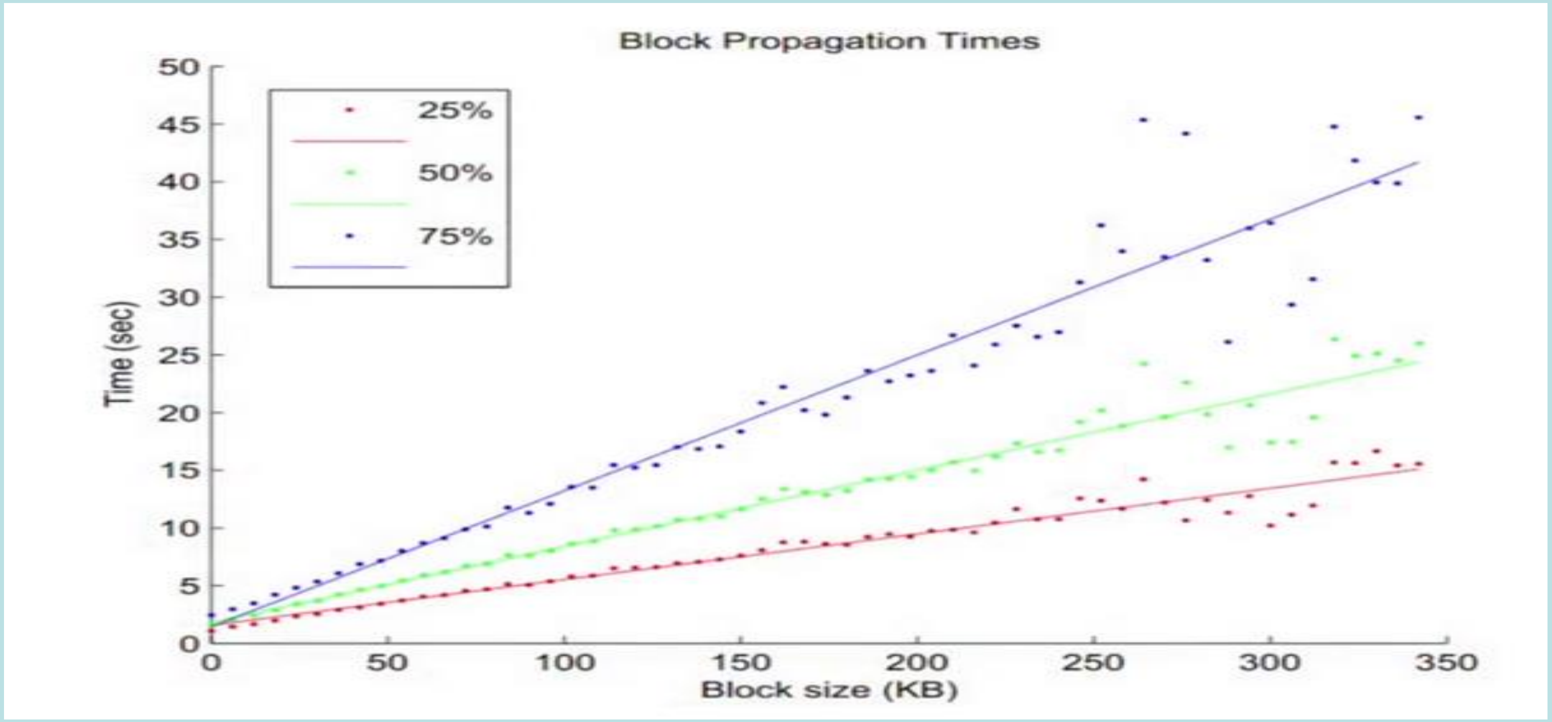
Block propagation

- The same transactions **flood algorithm** is used to announce new blocks around the network.
- In this case, the nodes are going to verify that **the new block is valid** by **computing the hash.**
- So, they can make sure that it starts with a sufficient number of zeroes to meet the difficulty target.
- Besides validating the headers, the miners have to check also that all transactions contained inside it are valid and new.
- In addition, a node shouldn't **forward a block**, unless it builds on its perspective of the current longest chain.
- The nodes have a view of the blockchain, and they should only **forward new blocks** if they come at the very end of the chain, not at some earlier point. This avoids the possibility of forks.

Bitcoin Network

Network statistics

- The **average *time for new blocks to propagate over all the network is around 30 seconds***. This shows that the protocol isn't **so efficient**. This is probably because it was designed to be simple e so that every node is equal to the other. The network isn't optimized for fast communication. But, for Bitcoin, it's more important to have a **decentralized structure** where all the nodes are equal.



Bitcoin Network

- It's difficult to count the number of ***online nodes***. But some researchers have estimated that there are over a **million IP connected** in a month that are both running the Bitcoin protocol and acting as a node. However, the number of full nodes permanently connected and that validated every transaction they hear is lower. It's only about 5 or 10 thousands.

Full nodes v. Lightweight nodes

- To be a fully-validating node, it's necessary to **stay permanently connected** and hear about all data.
- The longer one is **offline**, the more catch-up it will be necessary to do to hear about all the **missed transactions**.
- In addition, it's important a **good network connection** and huge storage (over 100GB). Being a full node also requires storing the unspent transactions in RAM to be able to run scripts on them. This requires around **1GB of RAM**.

Bitcoin Network

- Instead of being a fully **validating node**, there are lightweight nodes (**thin clients** or simple payment verification clients).
- This is the vast majority of nodes on the Bitcoin network.
- The difference is that these nodes aren't attempting **to store** the entire blockchain.
- They only **store the pieces** they need to verify some specific transactions they care about.

For example

- if you run **a wallet,** your wallet might want to be a simple payment verification node.
- And if somebody sends money to you, you'll act as a node.
- You will download the bits of the blockchain necessary to verify that the person sending you the money, actually owned it.
- Then you'll check the transaction actually gets included in the blockchain.
- An SPV client like this won't have the full security level of being a **fully validating** node.
- When they hear a **new block**, they can only check the **block header.** But they **can't check** the validity of transactions inside it, since they don't own the **entire blockchain.**
- They can only validate the transactions that actually affect them.
- So they're essentially trusting the **fully validating** nodes about the validity of other transactions.

Limitations and Improvements of Bitcoin Protocol

- Here we will talk about some built-in Limitations and Improvements of the Bitcoin Protocol.
- There are many **built-in limitations** in the Bitcoin protocol. These were chosen when Bitcoin was proposed **in 2009.** So that happened before anyone could think that it would have become a currency.
- **The most important are:**

  - ✓ average of 10 minutes between two blocks of creation
  - ✓ Maximum of 1Mb size for every block of content
  - ✓ 20,000 signatures in one block
  - ✓ 100 million satoshis in one Bitcoin
  - ✓ 21 million total Bitcoins
  - ✓ 50, 25, 12.5, ... Bitcoin mining reward per block.

Limitations and Improvements of Bitcoin Protocol

Transaction number limit

- These limitations are really never be changed since the economic implications are **too high**.
- **Miners** have invested a lot of real-world resources, relying on those limitations. So the community has agreed on those parameters, whether or not they were wisely chosen.
- The main aspect of Bitcoin that people are worried about is the **limit of transactions** that the system can handle. Because of the 1Mb maximum block size, **10 minutes** between one block and another, and transaction dimension which is at least **250 bytes**, there could be a maximum of **7 transactions per second.**
- This is really a small number, and could never be accepted for other kinds of payment used nowadays.

For example:

✓ Visa network can handle an average of 2000 transactions per second around the world

✓ Paypal can handle 100 transactions per second

Limitations and Improvements of Bitcoin Protocol

Bitcoin cryptography

- Another important limitation in the long term, is that **Bitcoin cryptography** is fixed.
- It is composed of only a couple of algorithms and only one signature algorithm, based on the **elliptic curve secp256k1**.
- In the future this algorithm could be broken. The same concern is true for the **hash function SHA-1** included in Bitcoin. Because this function has already some known cryptographic weaknesses.

Software releases

- To change some aspects, it would be necessary to release a new version of the software and hope that everybody accepts it. So, it would be a **hard-forking** change and **it's impossible** to assume that every node would upgrade.
- If only a group of nodes accept the new version, the blockchain would split.
- So the nodes will accept only one branch of it depending on the version of their software. It's clearly unacceptable.
- **By contrast**, there's an approach called **soft forking,** which tries to avoid the creation of **permanent forks.**
- The idea is that it is possible to add new features to the Bitcoin protocol, only if they restrict the set of valid transactions or valid blocks.
- So that nodes that don't upgrade will consider the new blocks **inserted valid**. While it could be possible that blocks created with the older version, **are rejected by the upgraded ones**. In this way, the nodes are encouraged to install the last version of the software.

Limitations and Improvements of Bitcoin Protocol

- An example of this kind of release was the pay-to-hash script. The new node checks if the hash of the data is equal to the value specified in the **output script.** While the old nodes, never do that **second step**, but only check that the value before being hashed corresponds to a **valid script**. So pay-to-script-hash was considered valid also by the old nodes.
- Other changes might require a **hard fork, for example,** to introduce new **OP_CODES into Bitcoin**, change the limits of block or transaction sizes, or do a lot of **bug fixes.**
- For example, ***the bug regarding the multi-signature transactions*** pops an extra **value off the stack**.