

Introduction to Computational and Modeling Tools

Lec 8: Applications in real life

AlphaGO, encrypted communication, block chain etc.

Instructor: Junwei Liu

10/28/2023

Monte Carlo method or random process

Monte Carlo methods, or Monte Carlo experiments, are a broad class of computational algorithms that rely on repeated **random sampling** to obtain numerical results. The underlying concept is to **use randomness to solve problems that might be deterministic in principle.**

MC methods are often used in physical and mathematical problems and are most useful when it is difficult or impossible to use other approaches. Monte Carlo methods are mainly used in three problem classes: optimization, numerical integration, and generating draws from a probability distribution.

In this lecture, we will show some example to use MC methods or random process in real life problems.

One Example: AlphaGO

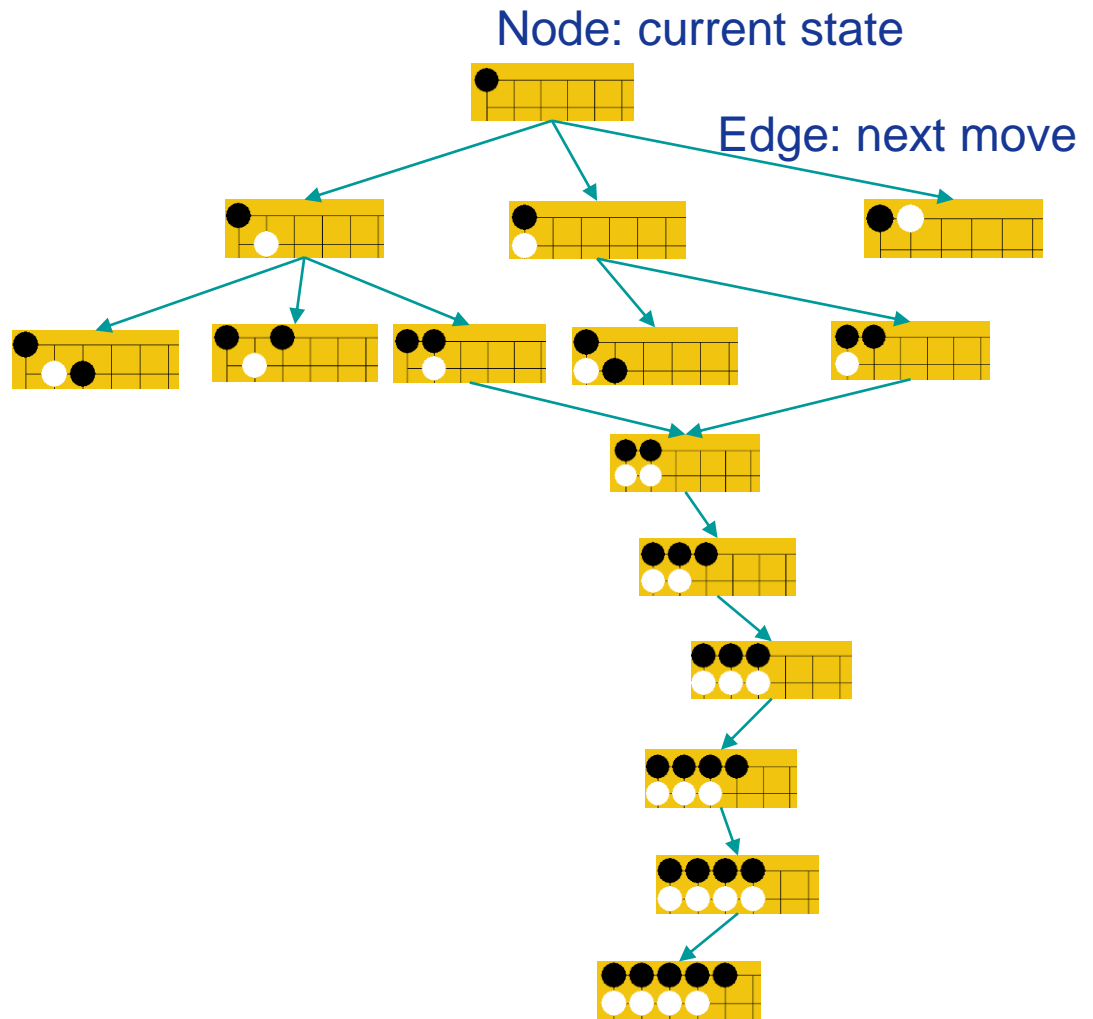


Deep learning →
Action-value
function

Monte Carlo Tree
Search →
Optimized move

Game tree

In game theory, a game tree is a directed graph whose **nodes** are positions in a game and whose **edges** are moves. The complete game tree for a game is the game tree starting at the initial position and containing all possible moves from each position

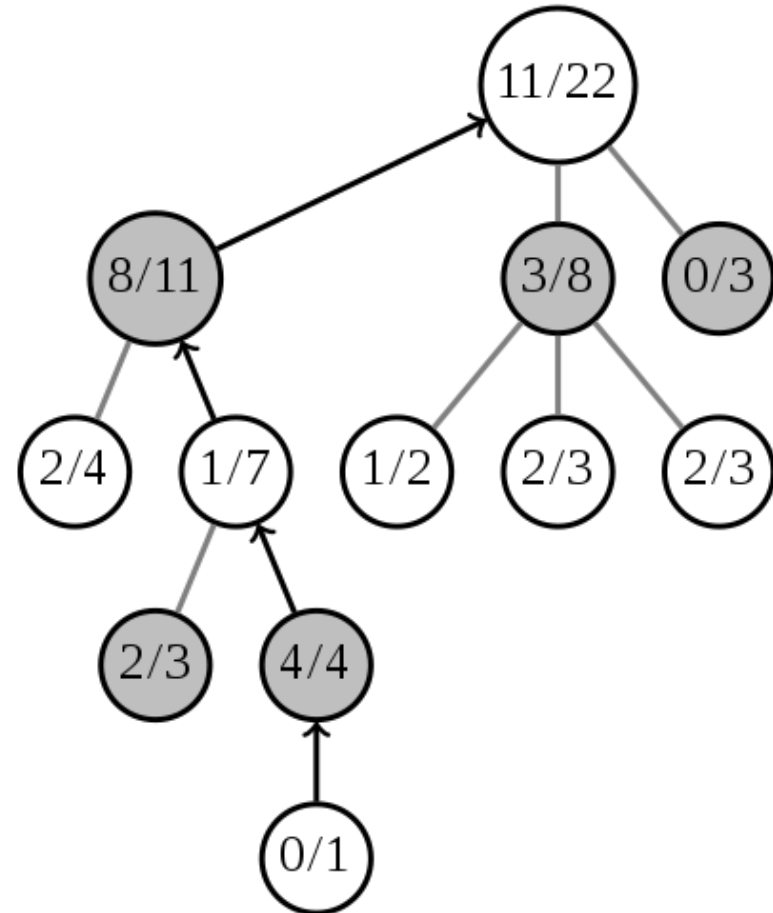


Game tree with winning probability



n: number of win
m: number of play

Every status could be taken as a node which can be assigned with some winning probability. And different statuses (nodes) are connected by different moves.



Monte Carlo tree search (MCTS)

MCTS is a **heuristic(启发式的)** search algorithm for some kinds of **decision processes**, most notably those employed in game play. MCTS was introduced in 2006 for computer Go. It has been used in other board games like chess and shogi, games with incomplete information such as bridge and poker, as well as in real-time video games (such as Total War: Rome II's implementation in the high level campaign AI).

The focus of MCTS is on the analysis of the most promising moves, expanding the search tree based on **random sampling of the search space**. The application of Monte Carlo tree search in games is based on many playouts also called roll-outs. In each playout, the game is played out to the very end by selecting moves at random. The final game result of each playout is then used to weight the nodes in the game tree so that better nodes are more likely to be chosen in future playouts.

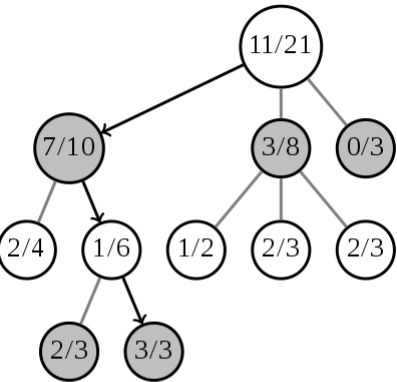
Pure Monte Carlo Game Search (PMCGS)

PMCGS is the most basic way to use playouts, where one applies the same number of playouts after each legal move, then chooses the move leading to the most victories. The more playouts, the more accurate. Each round of PMCGS consists of four steps:

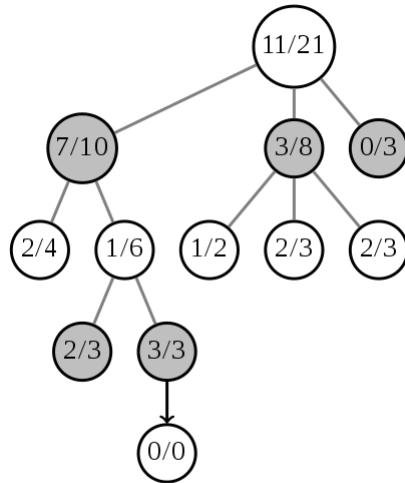
1. Selection: start from root R and select successive child nodes until a leaf node L is reached. The root is the current game state and a leaf is any node from which no simulation (playout) has yet been initiated.
2. Expansion: unless L ends the game decisively (e.g. win / loss / draw) for either player, create one (or more) child nodes and choose node C from one of them.
3. Simulation: complete a playout from node C . A playout may be as simple as choosing uniform random moves until the game ends
4. Backpropagation: use the result of the playout to update information in the nodes on the path from C to R .

Steps of Monte Carlo tree search

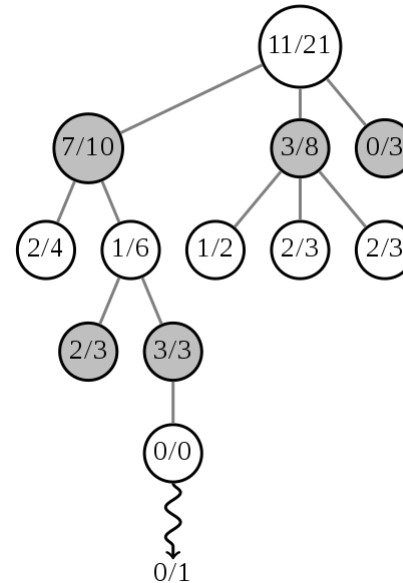
Selection



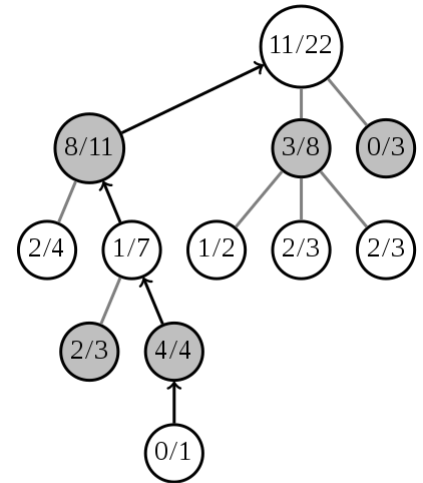
Expansion



Simulation



Backpropagation



Select one node using random number based on the probability of winning.

Expand the node by randomly choosing a legal move.

Randomly choose the following moves until the game ends

Use the result of the playout to update information for all the previous nodes

More discussion

- The main difficulty in selecting child nodes is maintaining some balance between the exploitation of deep variants after moves with high average win rate and the exploration of moves with few simulations.
- Pure Monte Carlo tree search does not need an explicit evaluation function. Simply implementing the game's mechanics is sufficient to explore the search space (i.e. the generating of allowed moves in a given position and the game-end conditions). As such, Monte Carlo tree search can be employed in games without a developed theory or in general game playing.
- A disadvantage is that, in a critical position against an expert player, there may be a single branch which leads to a loss. Because this is not easily found at random, the search may not "see" it and will not take it into account

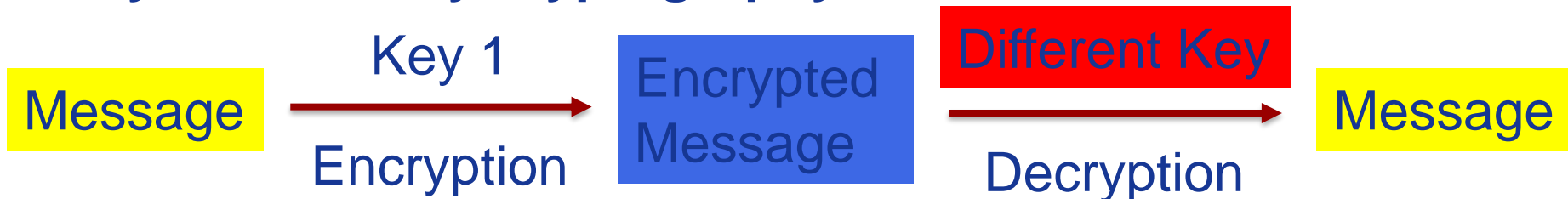
The other example: encryption

- In many cases, we want our information or message to be safe, then we need encryption method.

- **Symmetric-key cryptography**



- **Asymmetric-key cryptography**



Simplest symmetric-key algorithm

- A simple example is the ***substitution cipher***: take the letters of the alphabet and add a constant to them, e.g. if the constant is three, then you change A to D, B to E, C to F and so forth. Decoding the message just changes D to A, E to B, F to C and so forth.

Example codes

```
from numpy import mod
shift=1
all_characters='abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ`1234567890-=[]\;.,/~!@$%^&*()_+{}|:<>? '
num_char=len(all_characters)
codes={}; decodes={}
for ni in range(num_char):
    codes[all_characters[ni]]=all_characters[(ni+shift)%num_char]
    decodes[all_characters[mod(ni+shift,num_char)]] = all_characters[ni]

str1='we are learning encryption now. Do you understand it? Easy!!!'
str2=[];str3=[]
for ni in range(len(str1)):
    str2.append(codes[str1[ni]])

for ni in range(len(str2)):
    str3.append(decodes[str2[ni]])

print("The original message is  --->\n","".join(str1))
print("After coding, message is  --->\n","".join(str2))
print("After decoding, message is --->\n", "".join(str3))
```

It is very easy to break. You only need to try less than 100 times. In modern cryptography, the rule is to assume the worst: even the thief knows the principles behind the cipher you are using, they will not be able to read your message.

Better substitution cipher

- Instead of adding the same constant to each letter no matter where it is in your message, let us use different constants for letters in different positions of your message.

```
from numpy import mod
import numpy as np
```

```
def char_table(str,shift):
```

```
all_characters='abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ`1234567890-=[\],./~!@$%^&*()_+{}|:<>? '
```

Example codes-part2

```
def coding_decoding(str, codes, coding=1):  
    str2=[]  
    for ni in range(len(str)):  
        str2.append(char_table(str[ni], coding*codes[ni]))  
    return str2
```

```
np.random.seed(10)
```

```
str1='much better methods'  
codes=[];  
for ni in range(len(str1)):  
    codes.append(ni)
```

```
str2=coding_decoding(str1, codes, 1)  
str3=coding_decoding(str2, codes, -1)
```

```
print("The original message is --->\n", "".join(str1))  
print("After coding, message is --->\n", "".join(str2))  
print("After decoding, message is --->\n", "".join(str3))
```

It is still not very safe. You have to send “codes” to your friends. Once others have the codes, they can decode your message.

Further improvement: Random cipher

- The best way to create a varying cipher is to use random number!!! If all the numbers in the constant sequence is random, it is very hard to guess what the next number in the sequence will be. They need to try all possible combinations. For example, the message has M characters and the number of characters in the character table is N , then they need to try around N^M times!
- You can make one-time pad using real random numbers, while this method has a big disadvantage: you have to create the pad and get it to your friend without anyone else seeing it, also both parties have to guard it very carefully against theft, and it can only be used once.
- A much better choice is to use pseudorandom numbers. You do not need the entire pad to use it, and you just need to know the seed for the random number generator, which is called a **key**.
- Making a good secret code and making a good random number generator are basically the same problem.

Example codes: part1

```
from numpy import mod
from random import random,seed,randrange

def char_table(str,shift):

all_characters='abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ`1234567890-=[\],./~!@#$%^&*()_+{}|:<>? '
    num_char=len(all_characters)
    pos=all_characters.find(str)
    return all_characters[(pos+shift)%num_char]

def coding_decoding(str,codes,coding=1):
    str2=[]
    full_codes=codes
    if len(str)>len(codes):
        N_codes=len(codes)
        for ni in range(N_codes,len(str)):
            full_codes.append(codes[(ni-N_codes)%N_codes])
    for ni in range(len(str)):
        str2.append(char_table(str[ni],coding*full_codes[ni]))
    return str2
```


Example codes: part2

```
str1='I love hkust when i was a 7-year old child!'
```

```
key=1234567
seed(key)
codes=[];
codes_len=len(str1);
codes_range=1000;
for ni in range(codes_len):
    codes.append(randrange(codes_range))
```

```
#codes=[1]
str2=coding_decoding(str1, codes,1)
str3=coding_decoding(str2, codes,-1)
```

```
print("The original message is --->\n","".join(str1))
print("After coding, message is --->\n","".join(str2))
print("After decoding, message is --->\n", "".join(str3))
```

It is still not good enough. In many cases, we need to use asymmetric cryptography

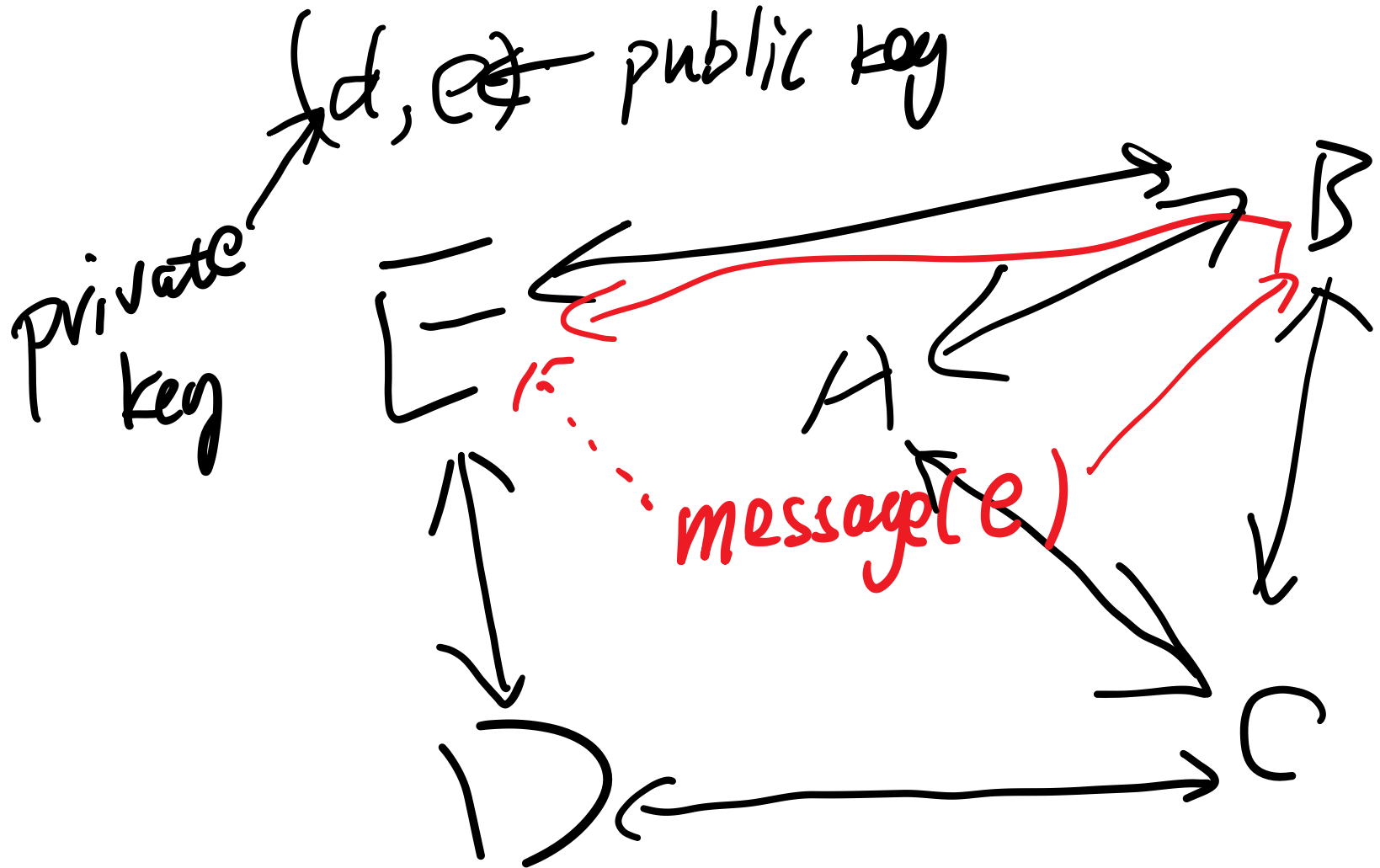
RSA—asymmetric cryptography

Key generation (n,e,d)

1. Choose two distinct prime numbers p, q . **Both are kept secret.**
2. Compute $n = p * q$. n is used as the modulus for both the public and private keys. n is released as part of the **public key**.
3. Compute $\phi(n) = (p - 1) * (q - 1)$. **$\phi(n)$ is kept secret.**
4. Choose an integer e such that $1 < e < \phi(n)$ and e and $\phi(n)$ should be are coprime.
 e having a short bit-length and small Hamming weight results in more efficient encryption – the most chosen value is $e = 2^{216} + 1 = 65537$. e is released as part of the **public key**.
5. Determine d as $d * e \bmod \phi(n) = 1$. d can be computed efficiently by using the Extended Euclidean algorithm. d is kept secret as the **private key** exponent.

Encryption and decryption

- The public key is (n, e) . The message m can be encrypted as $c = m^e \bmod n$.
- The private key is (n, d) . The encrypted message c can be decrypted as $m' = c^d \bmod n$.
- You can prove that as long as $m < n$, then $m' = m$
- RSA is very simple and useful. It has been generally used in **encrypted communication** and **digital signature**.
- The disadvantage of RSA is that it is not as fast as symmetric cryptography. So in cases of communicating large amount of messages, we combine both methods. The messages are encrypted using symmetric methods, while the key used in the symmetric cryptography is further encrypted using RSA.



$$A \xrightarrow{25} B \xrightarrow{25} E$$

$$A: 25 \xrightarrow{(n,e)} 16 \rightarrow B(n,e)$$

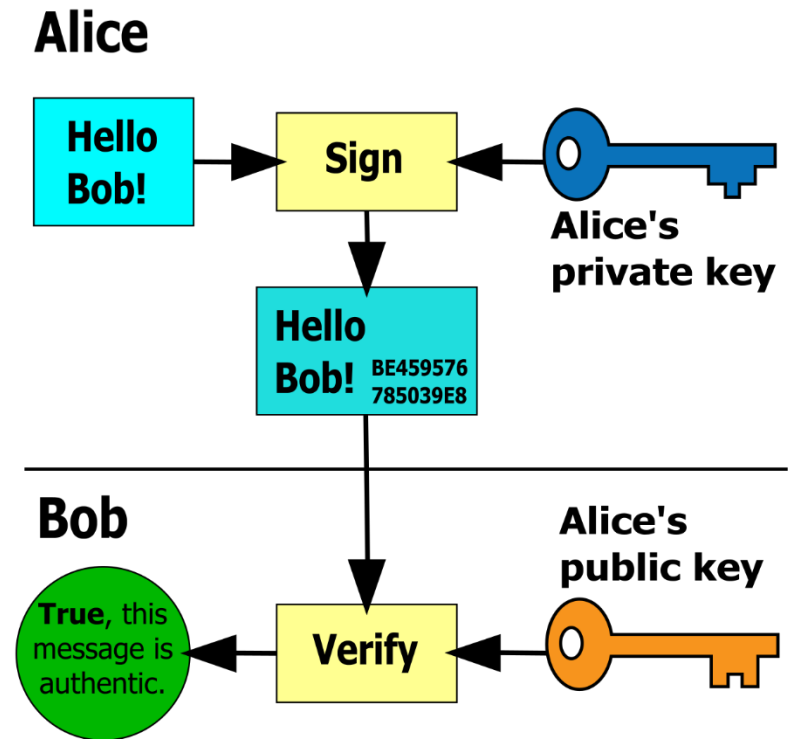
(n,e)

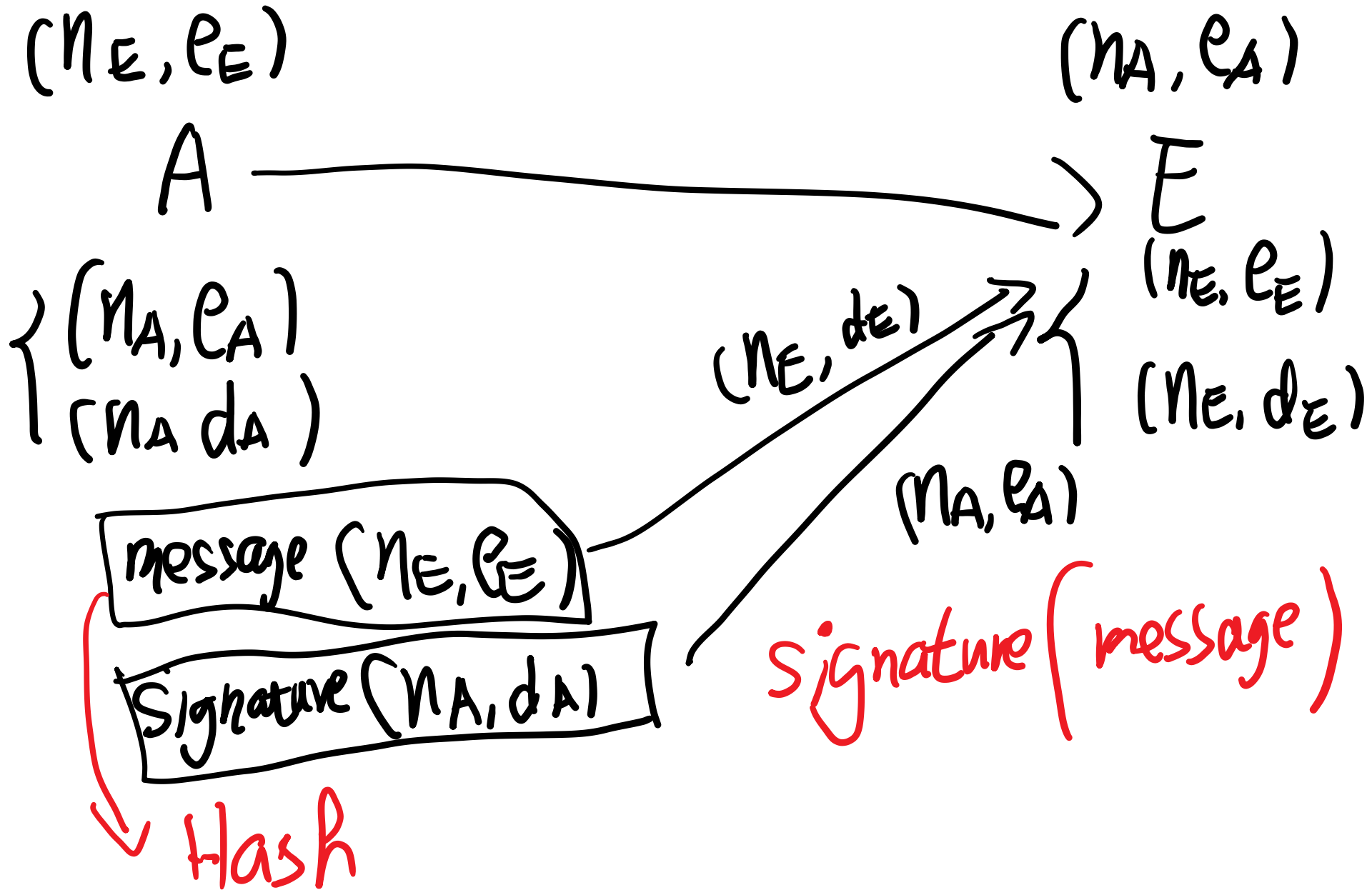
$$25 \leftarrow \underline{(n,d)} 16 \leftarrow E \xrightarrow{\text{generate } (n,e,d)}$$

$\downarrow 16, 17$

Digital signature

- To **encrypt message**, the message is **encrypted** using **public key** by others and then the encrypted message is sent to me. I can further use my **private key** to **decrypt** the encrypted message to get the original message.
- In **digital signature**, the process is opposite. I **encrypted** my signature using my **private key**, and others could use my **public key** to **verify** whether the signature is from me.





Hash Algorithm

A hash function is any function that can map data of arbitrary size to fixed-size values. The values returned by a hash function are called hash values, hash codes, digests, or hashes. The values are used to index a fixed-size table called a hash table.

The ideal cryptographic hash function has five main properties:

- It is deterministic so the same message always results in the same hash
- It is quick to compute the hash value for any given message
- It is infeasible to generate a message from its hash value except by trying all possible messages
- A small change to a message should change the hash value so extensively that the new hash value appears uncorrelated with the old hash value
- It is infeasible to find two different messages with the same hash value

Secure Hash Algorithms (SHAs)

- SHAs are a family of cryptographic hash functions published by the National Institute of Standards and Technology (NIST) as a U.S. Federal Information Processing Standard (FIPS). There are many different classes of functions. We will take SHA-256 or SHA-512 as an example.
- In SHA-256 or SHA-512, all the input string will be converted to be an binary number with 256 or 512 bits. You can easily use SHA-256 or SHA-512 in Python.

Using SHA in python

```
from hashlib import sha256
#from hashlib import sha512

data = input('Enter plaintext data: ')
output = sha256(data.encode('utf-8'))

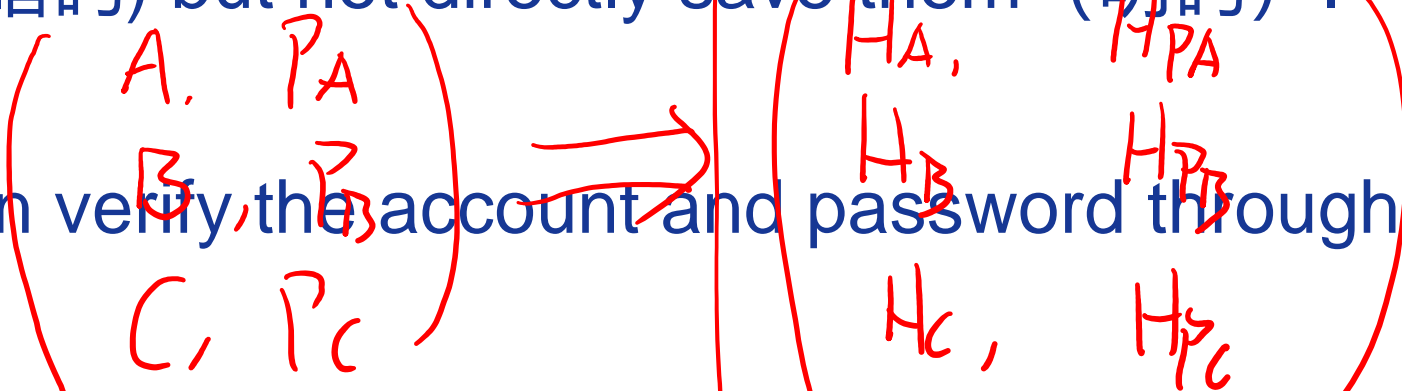
#show the results in different format
val=output.hexdigest();
int_val=int(val,16);
binary_val=bin(int_val)
```

You can also compare your results with the results from the following website <https://www.movable-type.co.uk/scripts/sha256.html>

An simple example: save password safely

The idea is to save both account and password in SHA value (暗码) but not directly save them (明码) .

One can verify the account and password through SHA.



A, P_A	$H_A,$	H_{P_A}
B, P_B	$H_B,$	H_{P_B}
C, P_C	$H_C,$	H_{P_C}

Advantage: one cannot access even he gets the full password table. It is much more safe and all the private information is well protected.

One minor disadvantage: the total number of account is only 2^{256} or 2^{512} . Usually, it should be large enough except you have a really big company (^o^)

The example code

```
from hashlib import sha256

def func_register(acnt,pwd,pwd_table={}):
    sha_acnt = sha256(acnt.encode('utf-8'));    sha_pwd = sha256(pwd.encode('utf-8'))
    if sha_acnt.hexdigest() in pwd_table:
        print(acnt,'has been used, please try another one')
    else:
        pwd_table[sha_acnt.hexdigest()]=sha_pwd.hexdigest()
    return pwd_table

pwd_table={}; pwd_table=func_register('student','Excellent',pwd_table)

acnt=input('Enter your acnt:\n'); pwd=input('Enter your password:\n')
sha_acnt = sha256(acnt.encode('utf-8')); sha_pwd = sha256(pwd.encode('utf-8'))
if sha_acnt.hexdigest() in pwd_table:
    if pwd_table[sha_acnt.hexdigest()]==sha_pwd.hexdigest():
        print('your password is right')
    else:
        print('your password is wrong')
else:
    print(acnt,'does not exist!')
```

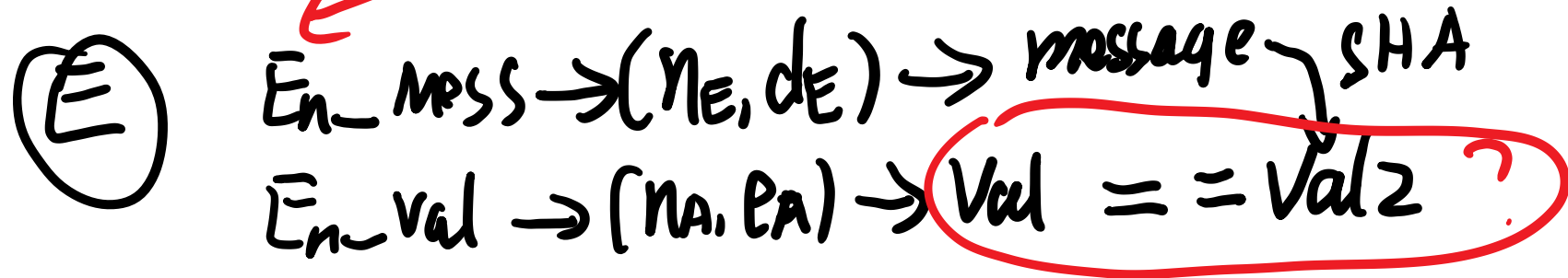
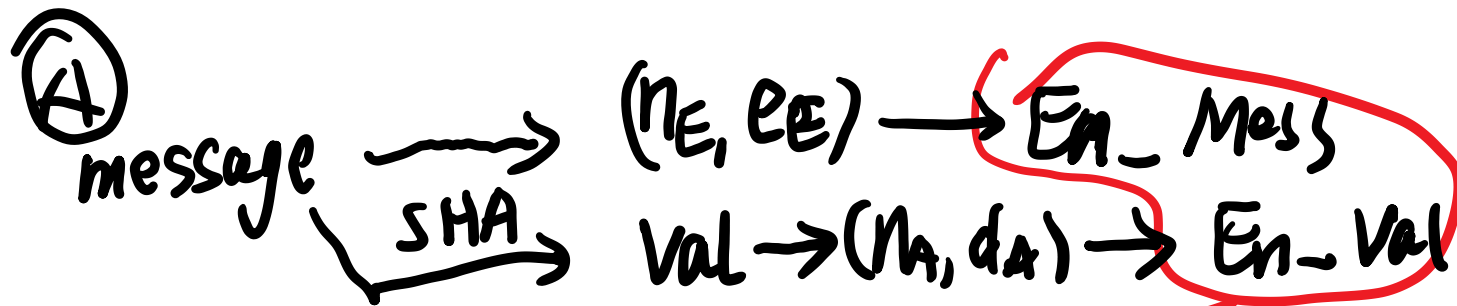
Why is it very safe?

- To get the text (明码), which has the same hash value. You need to try infinite number of times.
- To get the right hash value (暗码) . You need to try $2^{256} = 115792089237316195423570985008687907853269984665640564039457584007913129639936$ or $2^{512} = 13407807929942597099574024998205846127479365820592393377723561443721764030073546976801874298166903427690031858186486050853753882811946569946433649006084096$ times

Encrypted communication

- To build a really safe communication, we need two pairs of keys from both communicating sides, and use HASH methods.
- For example, A wants to send a message to B. A will use B's public key to encrypt the message and use A's private key to encrypt A's signature, and A send both encrypted message and signature to the internet.
- After B receives both encrypted message and signature from the internet, B could check whether it is from A through A's public key to verify A's signature. If so, B can use B's private key to decrypt the encrypted message to get the original message.

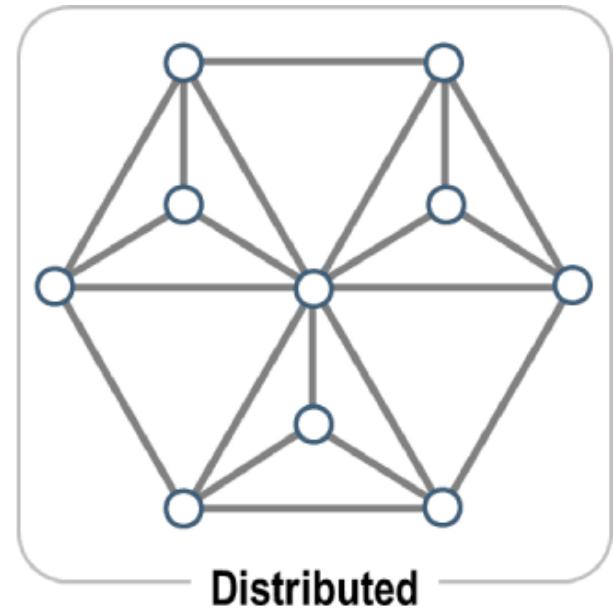
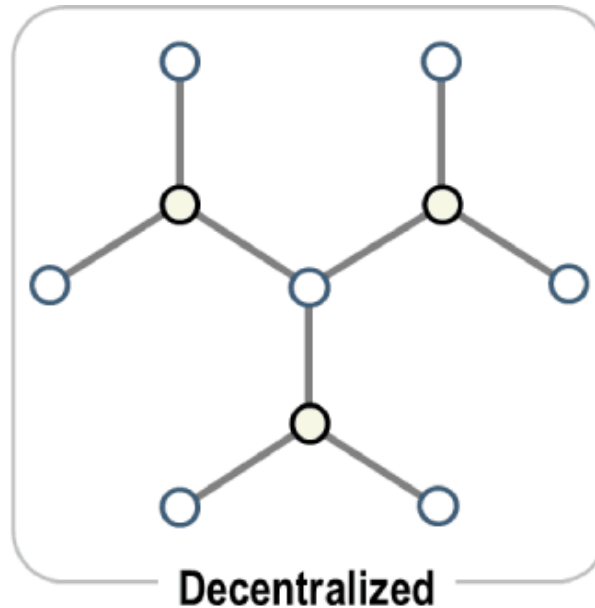
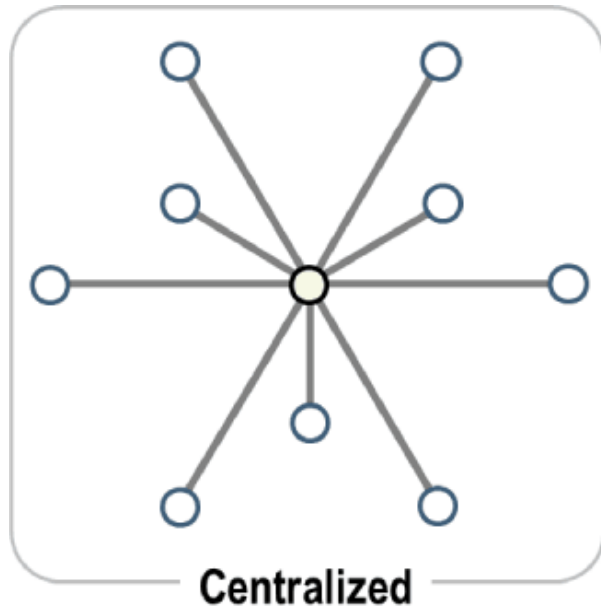
Completed Encrypted communication



Block Chain and Bitcoin

- A block chain is a growing list of **records**, called **blocks**, that are linked using cryptography. Each block contains a cryptographic **hash of the previous block**, a **timestamp**, and **transaction data** (generally represented as a Merkle tree).
- **Bitcoin** (฿) is a cryptocurrency. It is a **decentralized** digital currency without a central bank or single administrator that can be sent from user to user on the **peer-to-peer** bitcoin network without the need for **intermediaries**. Transactions are verified by **network nodes** through cryptography and recorded in a public distributed ledger called a **block chain**.

Structure of network

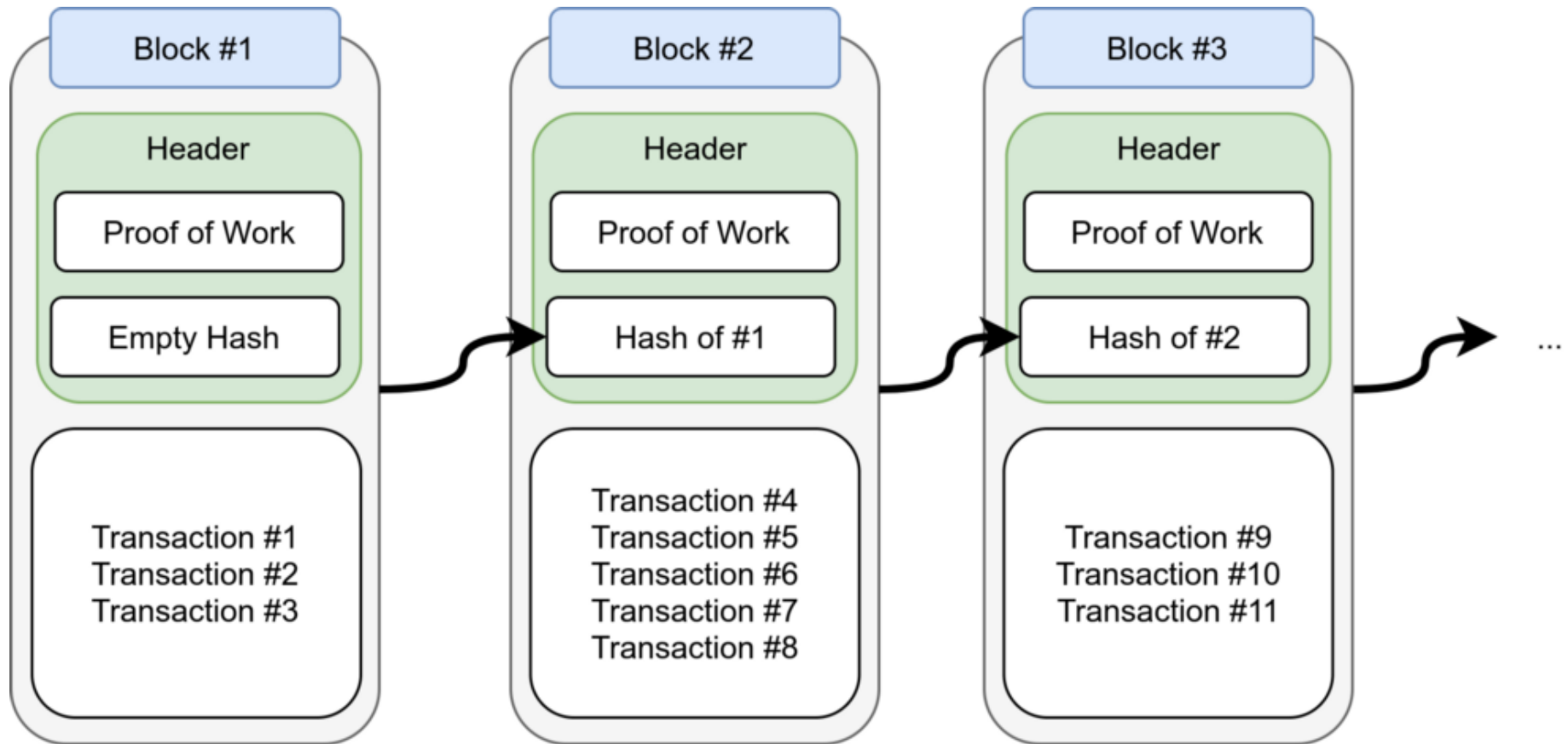


Centralized. One center has high accessibility and thus represents the dominant element of the network and the spatial structure it supports.

Decentralized. Although the center is still the point of highest accessibility, the network is structured so that sub-centers have also significant levels of accessibility.

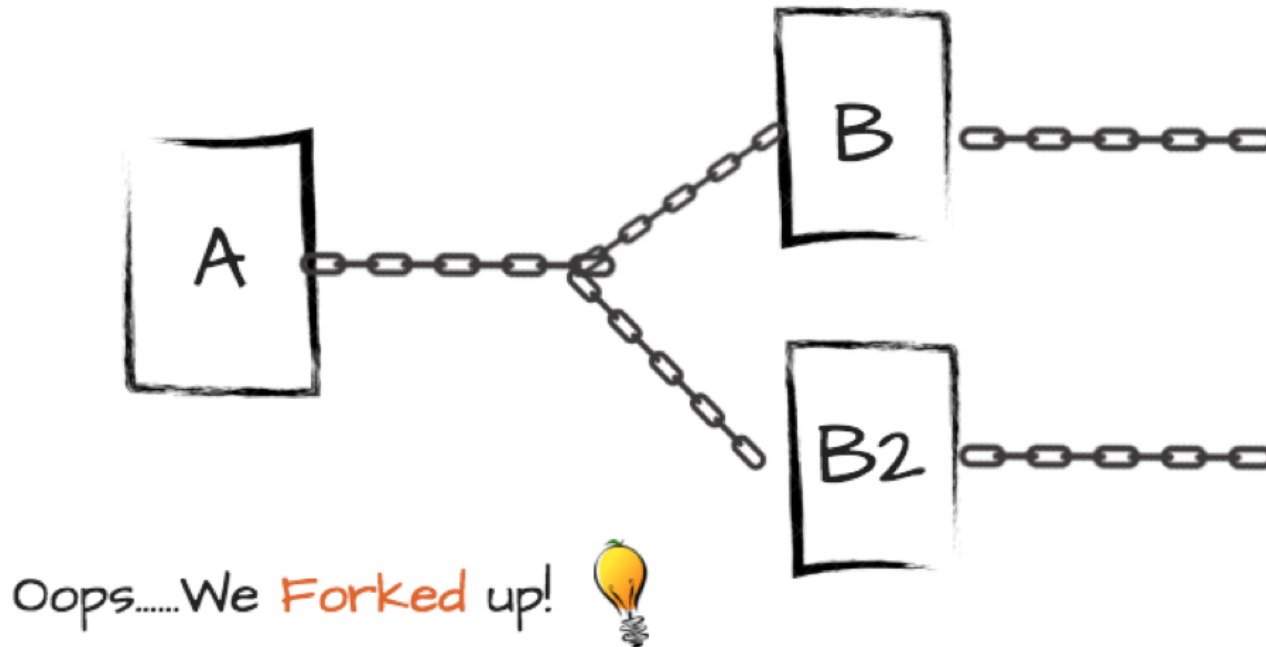
Distributed. No center has a level of accessibility significantly different from the others, which implies a high connectivity levels and redundancy.

Basic structure of blockchain



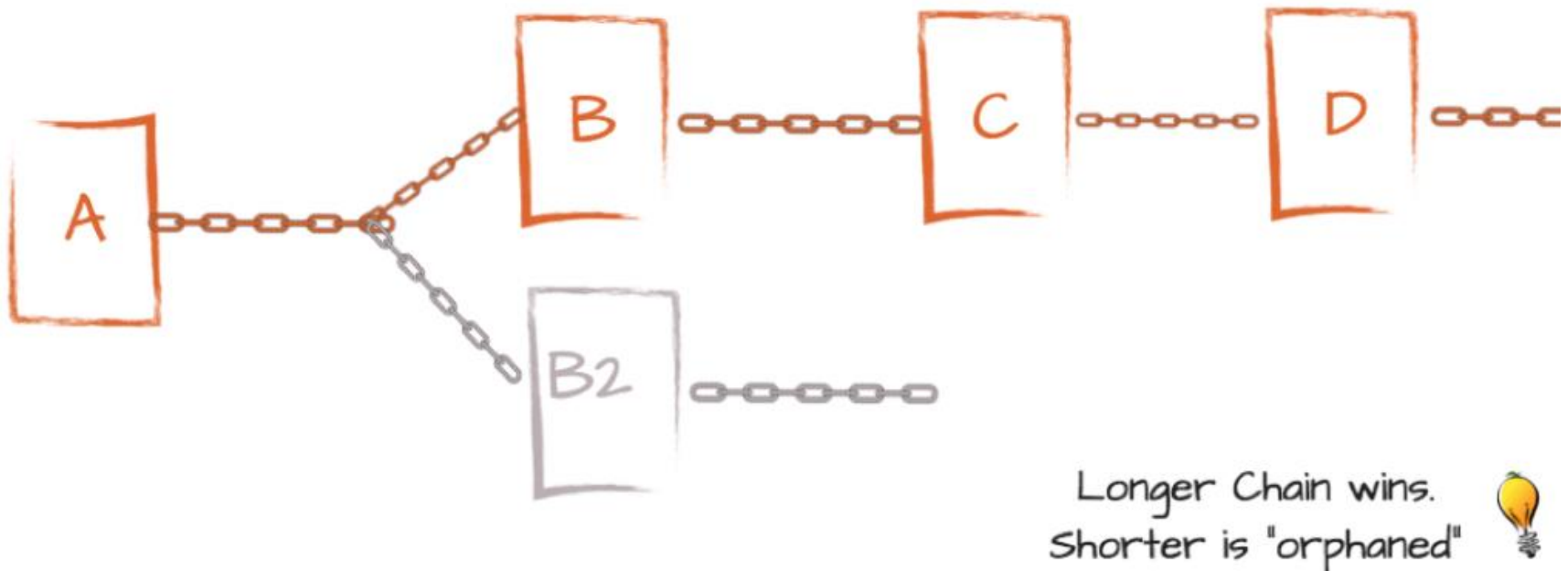
- We can use data to save different records or transactions
- These data cannot be changed
- Perfect ledger in the sense of security

Proof of work



- The prevent adding two blocks at the same time, we **artificially** make it difficult
- Proof of work: by adding a nonce (number only used once) to the block to ensure the Hash values smaller than a certain number (e.g., the first few digits to be zero).

Long chain is the right chain



- The Longest Chain Rule ensures that network will recognise the “chain with most work” as the main chain. The chain with the most work is typically (not always) the longest of the forks.
- All the blocks & transactions on other shorter chains will be considered orphaned and will be ignored.

Why does people do it?

- Reward 1: get a certain amount of bitcoin after I added a new block. This is called mining. This reward is cut in half every 210,000 blocks mined, or, about every four years. Eventually all the bitcoins will be mined around 2040.
- Reward 2: some one may pay you to add his transaction in the block chain.

How do you think?