# BLOCKCHAIN: TECHNOLOGY FOR FUTURE

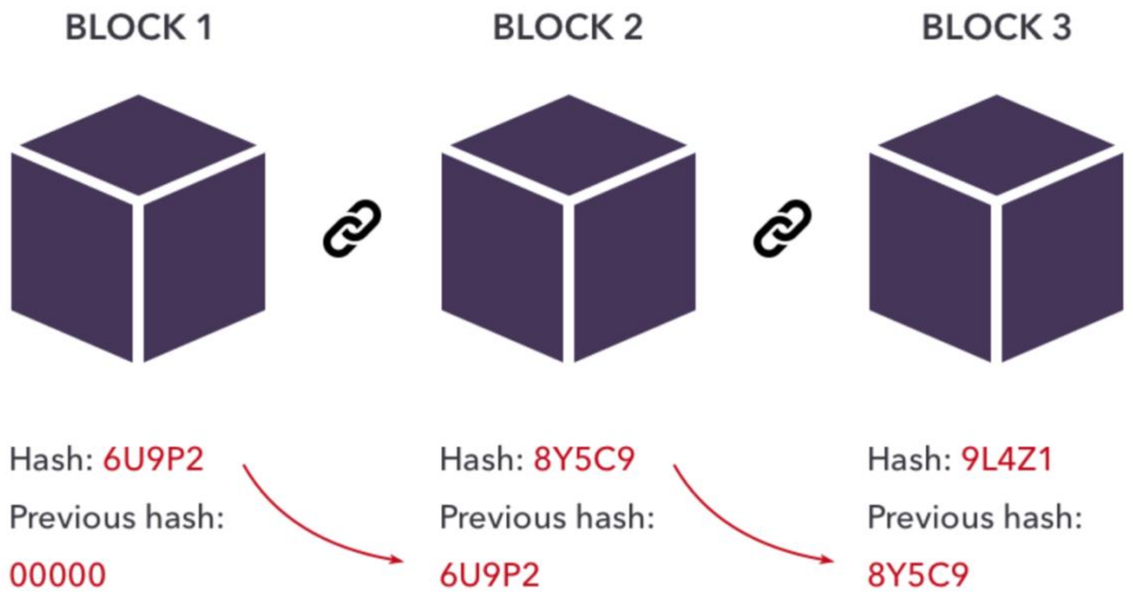
## Table of Contents

## INTRODUCTION

Blockchain is a data structure that stores data in form of a ledger and is shared among multiple independent machines in the network. The blockchain is a decentralized network such that no one needs to trust any organisation to act as an authority to validate the transactions and this is done by the peer machines in the network.

Blockchain term is composed of two words 'Block' and 'Chain', the block is like a page in a record-keeping book and chain is that each block has a link to the previous block, this ensures that the ledger cannot be tampered with.

Hash is a digital fingerprint of the data contained in the block and is calculated using the SHA-256 algorithm (Secure Hashing Algorithm -2), each block has a link to the hash of the previous hash maintains the integrity of the data structure. These huge blockchains are distributed across a network of nodes.



The new blocks in the blockchain can only be added only after a consensus has been reached among the nodes of the system, this can be done by algorithms, Proof of Work (POW) algorithm or Proof of Stake (POS) algorithm.

The blockchain can potentially have applications in different domains like finance, healthcare, IOT to maintain ledgers without a centralised authority, and along with that provide security which is a big improvement on the applications that already exist.

## SIMPLE IMPLEMENTATION OF BLOCKCHAIN IN JAVASCRIPT

The block of the blockchain can be an instance of a class represented by the attributes and properties as shown in figure (1). The blockchain can be represented as a class that has an attribute of the list of blocks and some relevant properties as shown in figure (2).

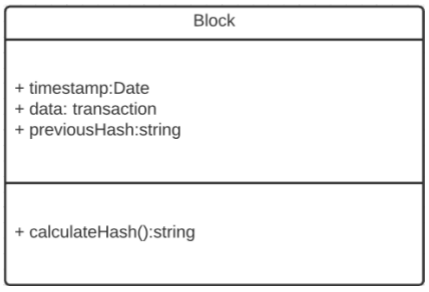


**Figure 1** `

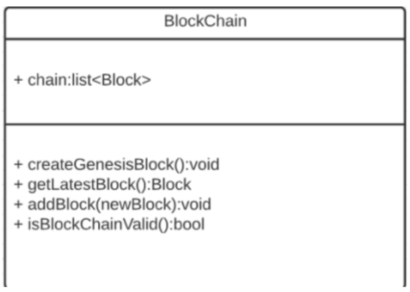
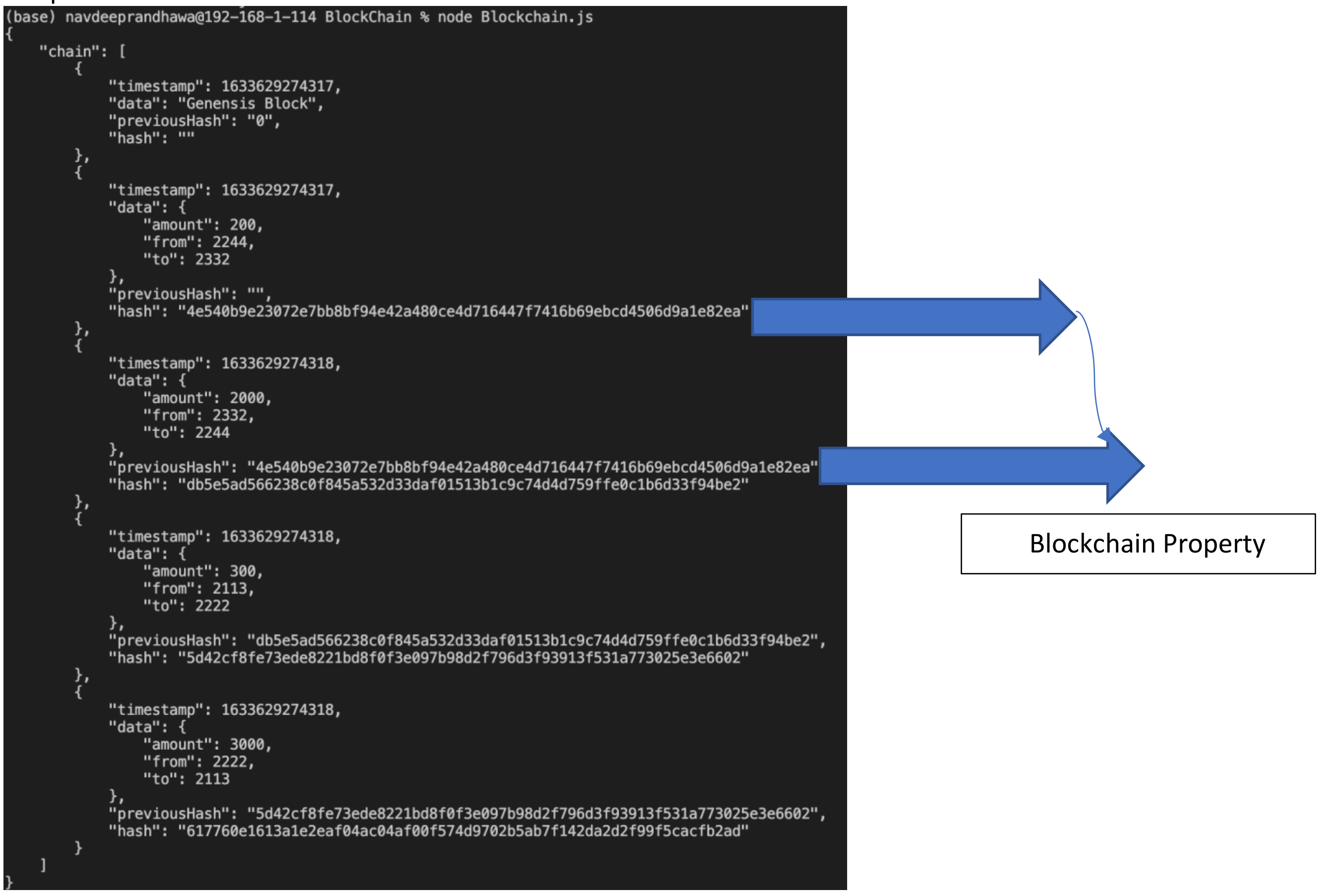

BlockChain
+ chain:list<Block>
+ createGenesisBlock():void
+ getLatestBlock():Block
+ addBlock(newBlock):void
+ isBlockChainValid():bool

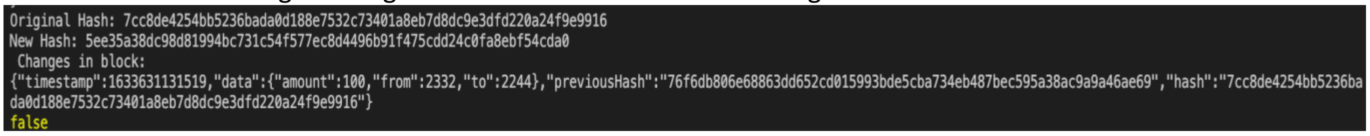
**Figure 2**

The sample output of an instance of the BlockChain class, with multiple blocks, is illustrated below. As evident, each block of data has a hash value associated with it and has information of the hash of the previous block.



```
(base) navdeeprandhawa@192-168-1-114 BlockChain % node Blockchain.js
{
    "chain": [
        {
            "timestamp": 1633629274317,
            "data": "Genensis Block",
            "previousHash": "0",
            "hash": ""
        },
        {
            "timestamp": 1633629274317,
            "data": {
                "amount": 200,
                "from": 2244,
                "to": 2332
            },
            "previousHash": "",
            "hash": "4e540b9e23072e7bb8bf94e42a480ce4d716447f7416b69ebcd4506d9a1e82ea"
        },
        {
            "timestamp": 1633629274318,
            "data": {
                "amount": 2000,
                "from": 2332,
                "to": 2244
            },
            "previousHash": "4e540b9e23072e7bb8bf94e42a480ce4d716447f7416b69ebcd4506d9a1e82ea"
            "hash": "db5e5ad566238c0f845a532d33daf01513b1c9c74d4d759ffe0c1b6d33f94be2"
        },
        {
            "timestamp": 1633629274318,
            "data": {
                "amount": 300,
                "from": 2113,
                "to": 2222
            },
            "previousHash": "db5e5ad566238c0f845a532d33daf01513b1c9c74d4d759ffe0c1b6d33f94be2",
            "hash": "5d42cf8fe73ede8221bd8f0f3e097b98d2f796d3f93913f531a773025e3e6602"
        },
        {
            "timestamp": 1633629274318,
            "data": {
                "amount": 3000,
                "from": 2222,
                "to": 2113
            },
            "previousHash": "5d42cf8fe73ede8221bd8f0f3e097b98d2f796d3f93913f531a773025e3e6602",
            "hash": "617760e1613a1e2eaf04ac04af00f574d9702b5ab7f142da2d2f99f5cacfb2ad"
        }
    ]
}
```

Now the question arises, Can the data stored in the blocks be easily changed? and the answer is No, it is one of the properties of the blockchain implemented using blockchain. In the Blockchain above, let's change the data of one of the blocks and check whether the blockchain is safe anymore or not. In figure (3), the data in the block in the output was tinkered and is was identified because the hash of the data with a changed a single bit is even different from the original data.



```
Original Hash: 7cc8de4254bb5236bada0d188e7532c73401a8eb7d8dc9e3dfd220a24f9e9916
New Hash: 5ee35a38dc98d81994bc731c54f577ec8d4496b91f475cdd24c0fa8ebf54cda0
 Changes in block:
{"timestamp":1633631131519,"data":{"amount":100,"from":2332,"to":2244},"previousHash":"76f6db806e68863dd652cd015993bde5cba734eb487bec595a38ac9a9a46ae69","hash":"7cc8de4254bb5236ba
da0d188e7532c73401a8eb7d8dc9e3dfd220a24f9e9916"}
false
```

**Figure 3**

## CONSENSUS ALGORITHMS IN BLOCKCHAIN NETWORK

A consensus algorithm in general is the mechanism in which all the nodes in the system reach an agreement over a single data value in the distributed systems of nodes. It is used to achieve reliability in the network by verifying the new data being added to the system.

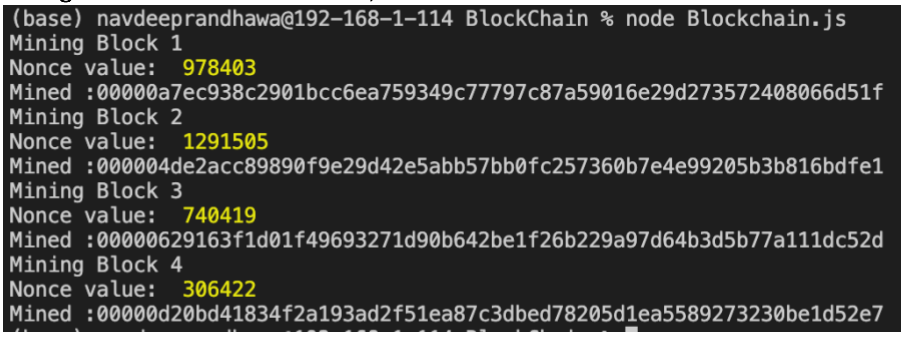
Consensus algorithms have a range of applications varying from load balancing and PageRank to Blockchain. The blockchain is distributed network of nodes and thus there must be an agreement among the nodes when a new block is added to the chain and it must be ensured that the data in the blockchain is not tinkered, this is where the consensus algorithms can be used. Although there are many consensus algorithms, most of the blockchains depend on namely two algorithms i.e., Proof of Work and Proof of State.

## PROOF OF WORK (PoW)

Proof of Work is a mechanism to ensure blockchain is safe from malicious attacks that aim to change the data or the structure of blockchain. This algorithm consumes a lot of computational resources as it is required to solve a very complicated mathematics puzzle. The puzzle is nothing but to come up with a hash value which has a prefix set to few numbers of 0's because it is a trivial job for computers to come up with any hash, the number of 0's required is termed as *difficulty* in the mining process, this leads to a race among the nodes in the system to come with the required hash value, the node which does it the quickest is rewarded and the block is thus added to the blockchain but the setting of the blockchain is set to such difficulty that a valid hash is generated in approximately 10 minutes.

*Nonce,* is another property of the Block class declared above, is a random number at which the valid hash is generated, this can be understood in the example below.

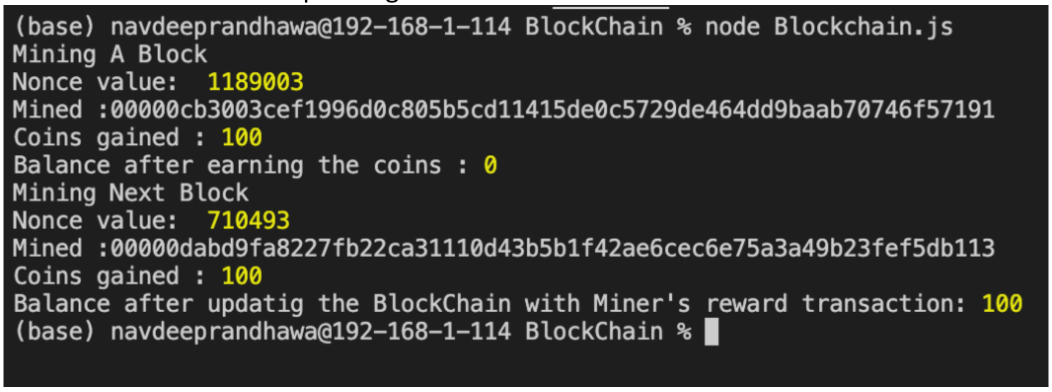
**Example of Proof of Work:**
There is the function within the BlockChain class that loops until a hash is generated that has the number of 0's in the prefix equivalent to difficulty.
In figure (4), the nonce value for the first block is 977403 which implies that 977402 hashes were calculated to get the hash required, there is a randomness in the number of nonces for each block, this would depend upon the difficulty, if the difficulty is lowered, nonces could be lower. This is because of how the Proof Of Work algorithm works and is heavily dependent on SHA-256 algorithm that generates the hash values, which we will delve more into.



```
(base) navdeeprandhawa@192-168-1-114 BlockChain % node Blockchain.js
Mining Block 1
Nonce value:  978403
Mined :00000a7ec938c2901bcc6ea759349c77797c87a59016e29d273572408066d51f
Mining Block 2
Nonce value:  1291505
Mined :000004de2acc89890f9e29d42e5abb57bb0fc257360b7e4e99205b3b816bdfe1
Mining Block 3
Nonce value:  740419
Mined :00000629163f1d01f49693271d90b642be1f26b229a97d64b3d5b77a111dc52d
Mining Block 4
Nonce value:  306422
Mined :00000d20bd41834f2a193ad2f51ea87c3dbed78205d1ea5589273230be1d52e7
```

**The reward for Proof of Work:**
The node that calculates the nonce value for a block is rewarded in form of coins that has monetary values, one such example is Bitcoin, if a node comes up with the nonce value for a block, the owner of the node is rewarded 6.25 Bitcoins and the transaction is further stored in Bitcoin Blockchain.
The block is mined and the miner is rewarded with the coins, to ensure that miner cannot spam the blockchain, the transaction has to be added to the blockchain before the miner gets the actual coins

in the account as shown below, the first block is mined and the transaction is stored in the blockchain which has a nonce value of 1189003, the miner is rewarded 100 coins, and this transaction for reward is pending and once finished the miner's balance increases with several coins.



```
(base) navdeeprandhawa@192-168-1-114 BlockChain % node Blockchain.js
Mining A Block
Nonce value:  1189003
Mined :00000cb3003cef1996d0c805b5cd11415de0c5729de464dd9baab70746f57191
Coins gained : 100
Balance after earning the coins : 0
Mining Next Block
Nonce value:  710493
Mined :00000dabd9fa8227fb22ca31110d43b5b1f42ae6cec6e75a3a49b23fef5db113
Coins gained : 100
Balance after updatig the BlockChain with Miner's reward transaction: 100
(base) navdeeprandhawa@192-168-1-114 BlockChain % 
```

## PROOF OF STATE (PoS)

The idea of Proof of Work is flawed when it comes to the wastage of computational resources of all the participants participating in the race to update the blockchain and find the nonce value since only one node can come up with a nonce value. This can be solved using the Proof of State mechanism to perform the consensus among nodes. In contrary to PoW where all the nodes in the system are performing the same computations to solve the puzzle explained in the earlier section, why not allot that work to just one node at a time. This is what exactly is done in the proof of state mechanism, a node is elected as a *validator* and that perform the work necessary to reach consensus among nodes and update blockchain and *mints* new coin. *The validator* has to deposit a certain number of coins in the network termed as *stake*, higher the stake, higher the chance to be appointed validator. *Validator* has the role of verifying all the transactions in a block and adds it to the blockchain, the coins minted as a result are stored in the *validator's* account along with a security deposit and if the validator verifies any fraudulent transaction, a penalty can be issued on *validator's* account.

The setback that PoS can have is the 51% rule, if a group of validators acquire 51% of the stakes of a blockchain then fraudulent activity could occur. Ethereum, a blockchain-based cryptocurrency is an example of a blockchain that uses PoS to update the blockchain.

## Cryptography in Blockchains

Cryptography is essentially the backbone of any blockchain, the algorithms to hash the data is where it is executed in the blockchains, there could be no secure blockchain system if not for cryptography. In the next few sections, this module will cover the hashing in blockchains specifically SHA-256 algorithm and use of the Merkle tree.
Cryptographic hash functions provide few benefits in the blockchain, the first benefit being avalanche effect, even if a bit of original data of the block changed, the hash function would produce different output, the second benefit being that the hash function used is deterministic, i.e., an input if passed through the hash function will always have the same output, another benefits are speed of algorithms and the fact that reverse engineering of output of hash function back to original data is impossible.

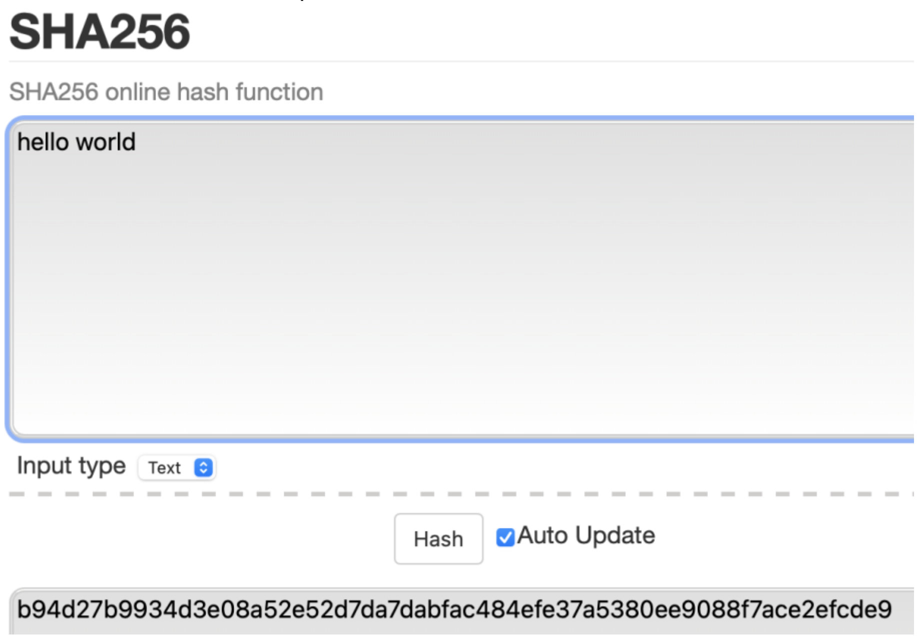
## HASHING ALGORITHMS

Hashing algorithms are the algorithms that transform data of any size to a fixed-length character series, the result is termed as *Hash*, in blockchain each data block is hashed and has a reference to the hash of the previous block.

These algorithms can be used for many applications like password hashes, the password of a user is converted into a hash before being stored in the database, as we know the reverse engineering is impossible, therefore the authority storing the password cannot identify the password. The other major application is integrity verification, as mentioned earlier even a change in a bit of data could change the value of hash, therefore it can be used to check the integrity of the data.

## SECURE HASHING ALGORITHMS (SHA-256): DEEP DIVE

SHA-256 is a hashing algorithm that was published in 2001 by NSA US and NIST US, the property being that the input data size will always be a multiple of 512 bits and output hash would be 256 bits long.
The example being considered for this section is of converting the "hello world" string into its hash value, in this section a step wise calculation is considered.



```
hello world
```

```
b94d27b9934d3e08a52e52d7da7dabfac484efe37a5380ee9088f7ace2efcde9
```

**Algorithm:**

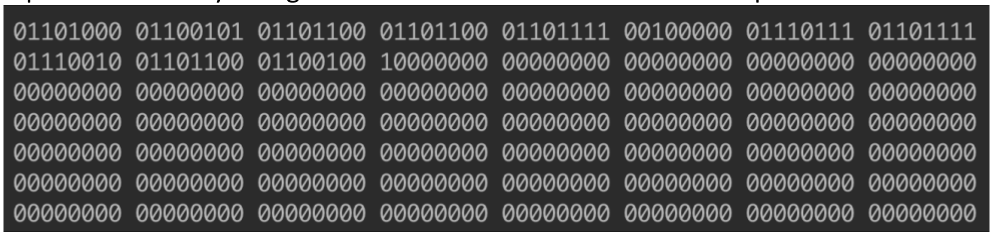
1) **Pre-processing of Data**
   - Pre-process the input data to convert it into binary. (initial string is “hello world”).

```
01101000 01100101 01101100 01101100 01101111 00100000 01110111 01101111
01110010 01101100 01100100
```

   - Append a single 1 to the binary value of the data.

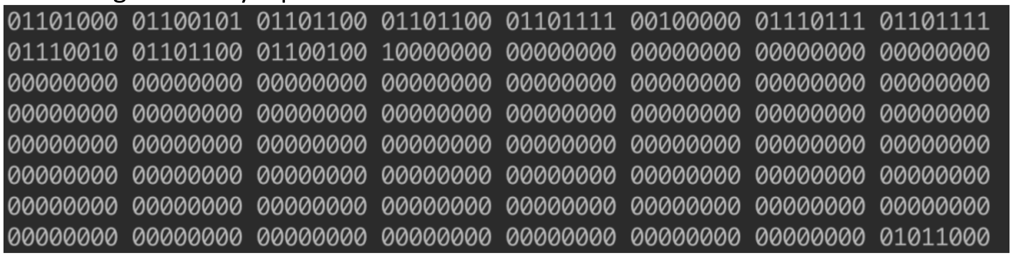
```
01101000 01100101 01101100 01101100 01101111 00100000 01110111 01101111
01110010 01101100 01100100 1
```

- As discussed earlier, the size of binary of data needs to be a multiple of 512, pad the input data's binary string with 0's until data is 64 less of a multiple of 512.



```
01101000 01100101 01101100 01101100 01101111 00100000 01110111 01101111
01110010 01101100 01100100 10000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
```

- The last 64 bits are big-endian integer, representing binary representation of the length of the original binary input of the data.

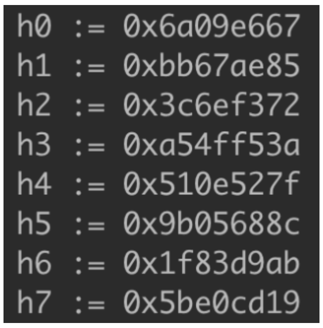


```
01101000 01100101 01101100 01101100 01101111 00100000 01110111 01101111
01110010 01101100 01100100 10000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 01011000
```

**2) The initialisation of Hash Value**

- Create 8 hash values, these are the constant values that are the first 32 bits of the fractional parts of the square roots of the first 8 prime numbers, these could be $n\sqrt{2}, n\sqrt{3}, n\sqrt{5}, n\sqrt{7}, n\sqrt{11}, n\sqrt{13}, n\sqrt{17}, n\sqrt{19}$, and n could be any positive integer number.

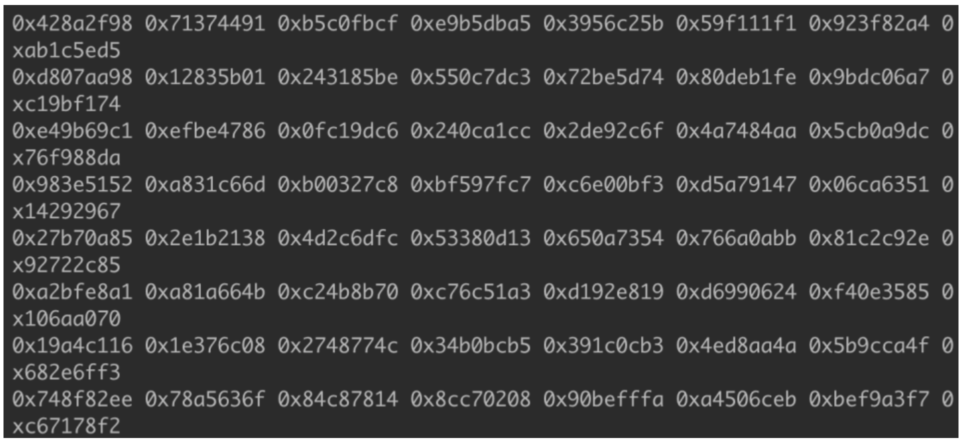
  This is a collection of hashes with n being equal to 3.



```
h0 := 0x6a09e667
h1 := 0xbb67ae85
h2 := 0x3c6ef372
h3 := 0xa54ff53a
h4 := 0x510e527f
h5 := 0x9b05688c
h6 := 0x1f83d9ab
h7 := 0x5be0cd19
```

**3) The initialisation of Round Constants (k)**

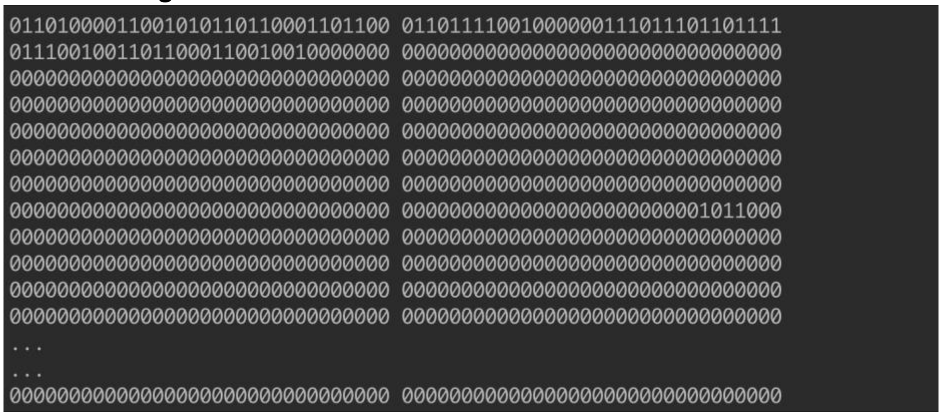
- Create 64 constants, like the previous step, the value of these constants is 32 bits of the fractional parts of the cube roots of the first 64 prime numbers, the value of n here is 3 as well.



```
0x428a2f98 0x71374491 0xb5c0fbcf 0xe9b5dba5 0x3956c25b 0x59f111f1 0x923f82a4 0
xab1c5ed5
0xd807aa98 0x12835b01 0x243185be 0x550c7dc3 0x72be5d74 0x80deb1fe 0x9bdc06a7 0
xc19bf174
0xe49b69c1 0xefbe4786 0x0fc19dc6 0x240ca1cc 0x2de92c6f 0x4a7484aa 0x5cb0a9dc 0
x76f988da
0x983e5152 0xa831c66d 0xb00327c8 0xbf597fc7 0xc6e00bf3 0xd5a79147 0x06ca6351 0
x14292967
0x27b70a85 0x2e1b2138 0x4d2c6dfc 0x53380d13 0x650a7354 0x766a0abb 0x81c2c92e 0
x92722c85
0xa2bfe8a1 0xa81a664b 0xc24b8b70 0xc76c51a3 0xd192e819 0xd6990624 0xf40e3585 0
x106aa070
0x19a4c116 0x1e376c08 0x2748774c 0x34b0bcb5 0x391c0cb3 0x4ed8aa4a 0x5b9cca4f 0
x682e6ff3
0x748f82ee 0x78a5636f 0x84c87814 0x8cc70208 0x90befffa 0xa4506ceb 0xbef9a3f7 0
xc67178f2
```

4) **The following steps will loop over for each 512-bit chunk of the input data, within the loop, create a message scheduler(w)**

- Store the binary input data from the first step into an array of 32 bits each.
- After the first step, the array would have 16 values, 512/32, make these 64 values by adding more 48 values initialised to zeroes, such take scheduler w has 64 items in it.
  **Initial Message Scheduler:**

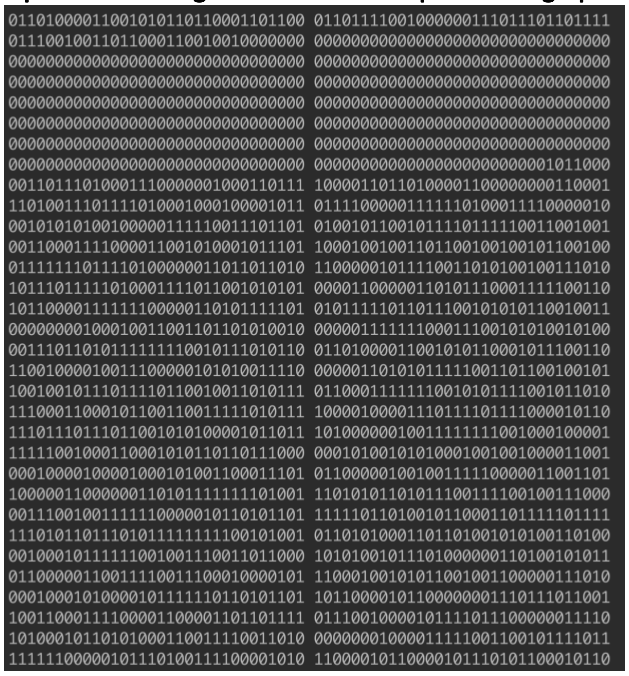


```
01101000011001010110110001101100 01101111001000000111011101101111
01110010011011000110010010000000 00000000000000000000000000000000
00000000000000000000000000000000 00000000000000000000000000000000
00000000000000000000000000000000 00000000000000000000000000000000
00000000000000000000000000000000 00000000000000000000000000000000
00000000000000000000000000000000 00000000000000000000000000000000
00000000000000000000000000000000 00000000000000000000000000000000
00000000000000000000000000000000 00000000000000000000000001011000
00000000000000000000000000000000 00000000000000000000000000000000
00000000000000000000000000000000 00000000000000000000000000000000
00000000000000000000000000000000 00000000000000000000000000000000
00000000000000000000000000000000 00000000000000000000000000000000
...
...
00000000000000000000000000000000 00000000000000000000000000000000
```

- In the next step, the message scheduler's values from index 13 to 63, last 48 items will be recalculated based on the following calculations:

  For **i** from w[16…63]:
  - s0 = (w[i-15] rightrotate 7) xor (w[i-15] rightrotate 18) xor (w[i-15] rightshift 3)
  - s1 = (w[i- 2] rightrotate 17) xor (w[i- 2] rightrotate 19) xor (w[i- 2] rightshift 10)
  - w[i] = w[i-16] + s0 + w[i-7] + s1

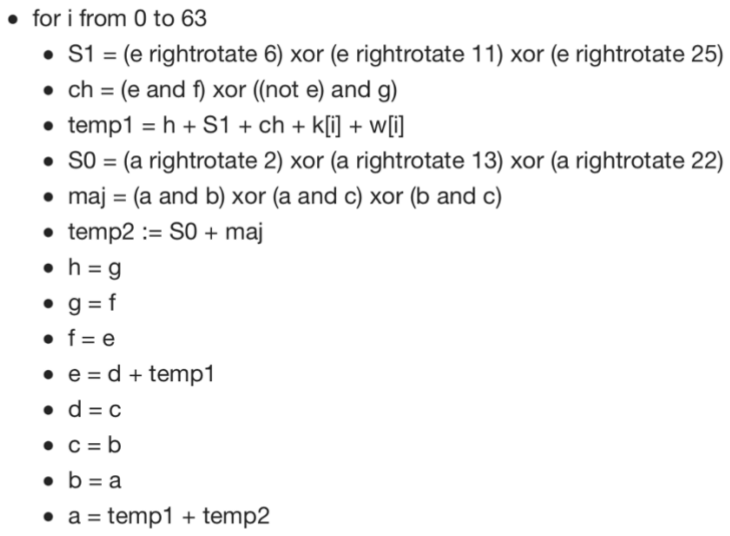
  Rightshift: Right shit of n =1100101 by 3 makes it 00011100, three bits from the last are removed and zeroes are added in the front to accommodate for the bits shifted out.
  Right rotate: Right rotation of n= 11100101 by 3 makes it 10111100, the bits are right-shifted by 3 and the last three bits are put in front.
  Xor: Xor function takes two bits and the result is 1 if only one of the bits is 1, and is 0 if both of the bits are either 1 or 0. 0 XOR 1=1, 0 XOR 0=0, 1 XOR 1=0.
- This would update the values of each index in the message scheduler array.
  **Updated message scheduler after performing operations mentioned above:**



```
01101000011001010110110001101100 01101111001000000111011101101111
01110010011011000110010010000000 00000000000000000000000000000000
00000000000000000000000000000000 00000000000000000000000000000000
00000000000000000000000000000000 00000000000000000000000000000000
00000000000000000000000000000000 00000000000000000000000000000000
00000000000000000000000000000000 00000000000000000000000000000000
00000000000000000000000000000000 00000000000000000000000000000000
00000000000000000000000000000000 00000000000000000000000001011000
00110111010001110000001000110111 10000110110100001100000000110001
11010011101111010001000100001011 01111000001111101000111100000010
00101010100100000111110011101101 01001011001011110111110011001001
00110001111000011001010001011101 10001001001101100100100101100100
01111111011110100000011011011010 11000001011110011010100100111010
10111011111010001111011001010101 00001100000110101110001111100110
10110000111111100000110101111101 01011111011011100101010110010011
00000000100010011001101101010010 00000111111100011100101010010100
00111011010111111110010111010110 01101000011001010110001011100110
11001000010011100000010101001110 00000110101011111001101100100101
10010010111011110110010011010111 01100011111110010101111001011010
11100011000101100110011111010111 10000100001110111101111000010110
11101110111011001010100001011011 10100000010011111111001000100001
11111001000110001010110110111000 00010100101010001001001000011001
00010000100001000101001100011101 01100000010010011111000001100101
10000011000000110101111111101001 11010101101011100111100100111000
00111001001111110000010110101101 11111011010010110001101111101111
11101011011101011111111100101001 01101010001101101001010100110100
00100010111111001001110011011000 10101001011101000000110100101011
01100000110011110011100010000101 11000100101011001001100000111010
00010001010000101111110110101101 10110000101100000001110111011001
10011000111100001100001101101111 01110010000101111011100000011110
10100010110101000110011110011010 00000001000011111001100101111011
11111100000101110100111100001010 11000010110000101110101100010110
```

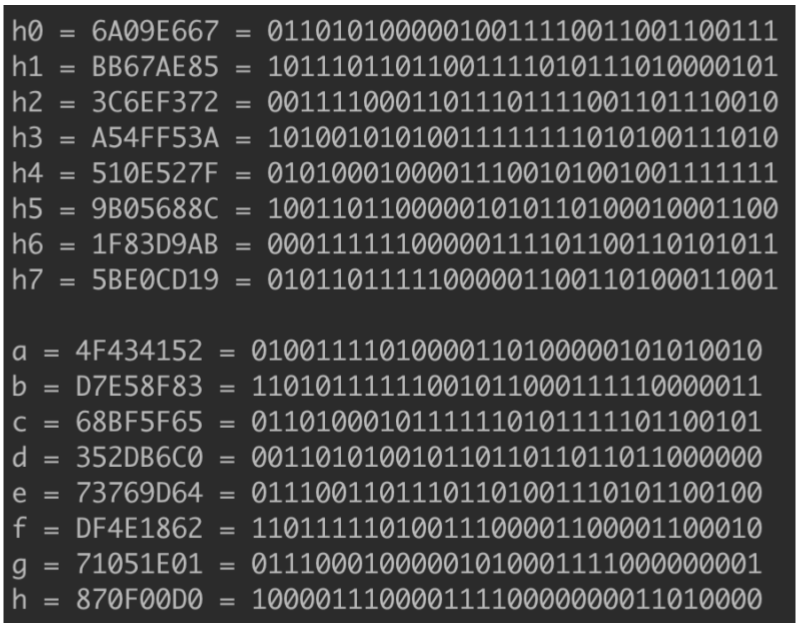
**5) The next step is that of compression, which would yield the final output hash of 32 bits.**

- The first step is to copy the values of all hash values calculated in step (2) into new variables, a,b,c,d,e,f,g,h.
- After the first step, run a loop that loops over the message scheduler array and perform the calculations as shown:



```
• for i from 0 to 63
    • S1 = (e rightrotate 6) xor (e rightrotate 11) xor (e rightrotate 25)
    • ch = (e and f) xor ((not e) and g)
    • temp1 = h + S1 + ch + k[i] + w[i]
    • S0 = (a rightrotate 2) xor (a rightrotate 13) xor (a rightrotate 22)
    • maj = (a and b) xor (a and c) xor (b and c)
    • temp2 := S0 + maj
    • h = g
    • g = f
    • f = e
    • e = d + temp1
    • d = c
    • c = b
    • b = a
    • a = temp1 + temp2
```

**Result of Compression:**



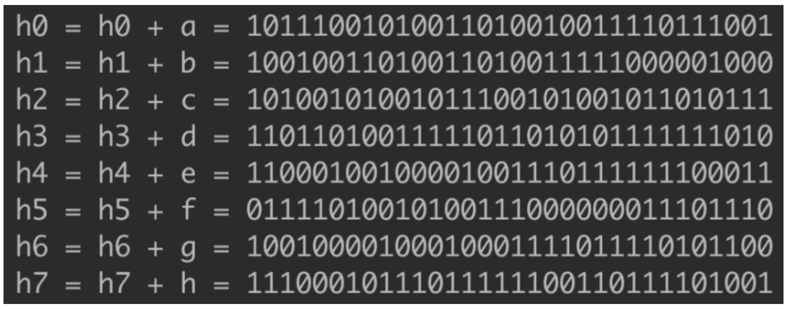
```
h0 = 6A09E667 = 01101010000010011110011001100111
h1 = BB67AE85 = 10111011011001111010111010000101
h2 = 3C6EF372 = 00111100011011101111001101110010
h3 = A54FF53A = 10100101010011111111010100111010
h4 = 510E527F = 01010001000011100101001001111111
h5 = 9B05688C = 10011011000001010110100010001100
h6 = 1F83D9AB = 00011111100000111101100110101011
h7 = 5BE0CD19 = 01011011111000001100110100011001

a = 4F434152 = 01001111010000110100000101010010
b = D7E58F83 = 11010111111001011000111110000011
c = 68BF5F65 = 01101000101111110101111101100101
d = 352DB6C0 = 00110101001011011011011011000000
e = 73769D64 = 01110011011101101001110101100100
f = DF4E1862 = 11011111010011100001100001100010
g = 71051E01 = 01110001000001010001111000000001
h = 870F00D0 = 10000111000011110000000011010000
```

**6) The final two steps are:**

- Update the values of hashes calculated in step (2) as following and each value is modulo of 2^32.

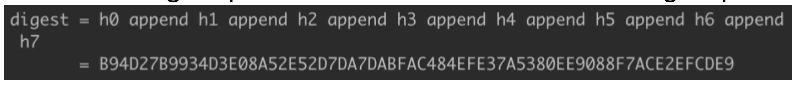
$$h0 = h0 + a$$
$$h1 = h1 + b$$
$$h2 = h2 + c$$
$$h3 = h3 + d$$
$$h4 = h4 + e$$
$$h5 = h5 + f$$
$$h6 = h6 + g$$

$$h7 = h7 + h$$



```
h0 = h0 + a = 10111001010011010010011110111001
h1 = h1 + b = 10010011010011010011111000001000
h2 = h2 + c = 10100101001011100101001011010111
h3 = h3 + d = 11011010011111011010101111111010
h4 = h4 + e = 11000100100001001110111111100011
h5 = h5 + f = 01111010010100111000000011101110
h6 = h6 + g = 10010000100010001111011110101100
h7 = h7 + h = 11100010111011111100110111101001
```

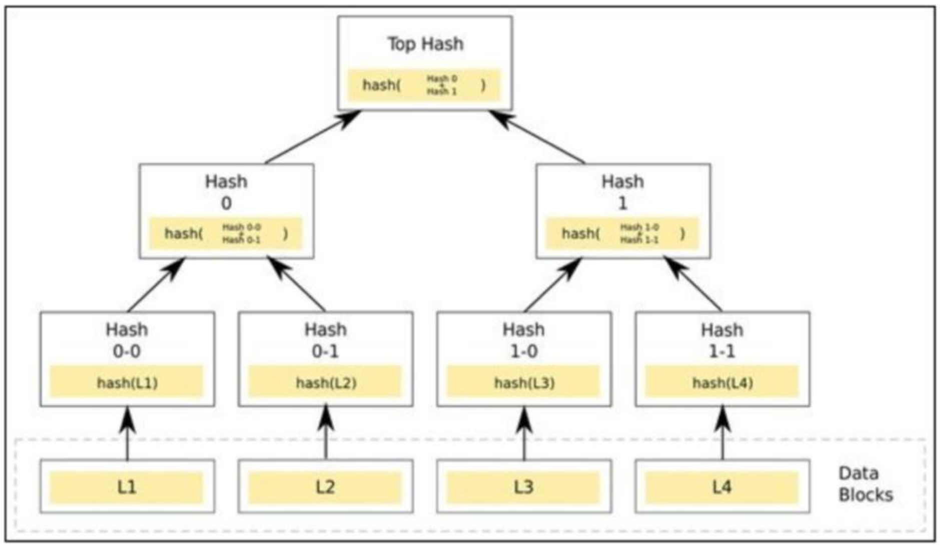
- The final string output is a concatenation of all the strings representing hash values .



```
digest = h0 append h1 append h2 append h3 append h4 append h5 append h6 append
 h7
       = B94D27B9934D3E08A52E52D7DA7DABFAC484EFE37A5380EE9088F7ACE2EFCDE9
```

## MERKLE TREE IN BLOCKCHAIN

Merkle tree is a binary hash tree, and each leaf node is the hash of a data block, and non-leaf nodes are labelled with the hashes of their child nodes.
In the figure below, each leaf node is the hash value of the data blocks, with L1, L2, L3 and L4 being data blocks and leaf nodes are their respective hash values. The non-leaf nodes are hashes of the concatenation of their child nodes hash values. The top hash is hash (Hash 0+ Hash 1) and similarly, Hash 0 is (Hash 0-0 + Hash 0-1) and Hash 1 is (Hash 1-0 + Hash 1-1) and Hash 0-0, Hash 0-1, Hash 1-0, Hash 1-1 are hash values of data blocks. The top hash is also known as *Merkle Root*, which has a significant role in the security of blockchains. This data structure has a range of applications like DynamoDB, Git and Blockchain.



If a block within a blockchain implements more than one transaction, then the Merkle tree can be used within the block to ensure that the block stores only one hash value, but also secure all the transactions implemented in the block. Let's consider there are four transactions for one node, A, B, C, D, the hash of each transaction would make the leaf nodes of as shown in the figure above and the top hash would be the hash value stored in the block. This ensures that the storage resources are used efficiently, and the data is secured.

## APPLICATION OF BLOCKCHAIN: BITCOIN

Bitcoin is an online currency developed by Satoshi Nakamoto in 2008 and he highlighted his work in the paper Bitcoin: A Peer-to-Peer Electronic Cash System. The motivation behind the development was to create a faster way to send money from one place to another without going through a financial institution and still maintain the trust among the parties involved in the transaction. This was solved using a blockchain where nodes could add transactions to the blockchain and get rewarded in form of the digital currency known as Bitcoin. The Bitcoin blockchain inherently solves Byzantine Generals problems in the world of information, i.e., How to trust where the information is coming from? This is solved using distributed ledger which is available to all the nodes in the system, as all the transactions taking place are stored in the ledger in form of hashes which are impossible to be changed because of the avalanche effect a change in the data of a block can cause throughout the distributed system.

## APPLICATION OF BLOCKCHAIN: ETHEREUM

Ethereum is a blockchain that creates and shares business, developers can build and publish smart contracts and DAPPs that are decentralized in nature and cannot be tampered with, the developers can monetize the applications developed can Ethereum's cryptocurrency Ether can be used as a mode of payment. Ethereum is the blockchain exploring the use cases of technology beyond digital currency and providing the accessibility to develop applications that are decentralised, distributed and almost impossible to hack, this could have breakthroughs in fields like healthcare, trading of digital assets, and financial services.

## APPLICATION OF BLOCKCHAIN: HYPERLEDGER FABRIC

Ethereum and Bitcoin both are public permissionless blockchains, as they are open to anyone to participate anonymously, whereas some enterprise-level applications cannot be open to the public and the participants must have an identity within the network. Following are the requirements of the enterprise-level applications, firstly participants of the network must have an identity, the network must be permissioned in contrast to Bitcoin and Ethereum, performance must be greater, low latency while confirming these transactions and privacy of the transactions in the blockchain is the utmost priority.

Hyperledger Fabric, a project of Linux Foundations specifically targets enterprise-level applications. The Hyperledger Fabric has a modular architecture with the following modular components:

- Ordering Service: This creates consensus on the transactions and broadcast the blocks of transaction to all the nodes in the network.
- Membership Service Provider: This service provider is responsible for providing cryptographic identities to the participants and embed them within the network.
- Smart Contracts: The Smart Contracts are deterministic business logic behind the blockchain application to reach consensus among the nodes of the network. The Smart Contracts are the code functions that validates and orders transactions and broadcast them to all the nodes in the network and each node in the network then can implement the transaction in a sequence.
- In Fabric network, a transaction is executed and is endorsed, then ordered via a protocol and is validated as per the endorsement policy of the specific application.

## REFERENCES

Laurence, T., n.d. *Blockchain for dummies*. Wiley's.

ethereum.org. 2021. *Intro to Ethereum | ethereum.org*. [online] Available at: <https://ethereum.org/en/developers/docs/intro-to-ethereum/> [Accessed 27 September 2021].

Nakamoto, S., 2008. *Bitcoin: A Peer-to-Peer Electronic Cash System*. [ebook] pp.1-4. Available at: <https://bitcoin.org/bitcoin.pdf> [Accessed 4 October 2021].

Capgemini Australia. 2021. *What is the blockchain Solving the Generals' Problem*. [online] Available at: <https://www.capgemini.com/au-en/2018/07/what-is-the-blockchain-solving-the-generals-problem/> [Accessed 10 October 2021].

Veness, C., 2021. *SHA-256 Cryptographic Hash Algorithm implemented in JavaScript | Movable Type Scripts*. [online] Movable-type.co.uk. Available at: <https://www.movable-type.co.uk/scripts/sha256.html> [Accessed 10 October 2021].

Hyperledger.org. 2021. [online] Available at: <https://www.hyperledger.org/wp-content/uploads/2018/08/HL_Whitepaper_IntroductiontoHyperledger.pdf> [Accessed 9 October 2021].

Wagner, L., 2021. *How SHA-256 Works Step-By-Step*. [online] https://qvault.io/. Available at: <https://qvault.io/cryptography/how-sha-2-works-step-by-step-sha-256/> [Accessed 10 October 2021].

## LINK FOR MODULE PRESENTATION

https://www.youtube.com/watch?v=j8WXBTqm084