# q1: Question 1

## 1.1: Basics

1. All apart from supernodes

## 1.2: Cryptography

1. Nothing in Bitcoin, network communication in Ethereum
2. Transactions
3. The hash of the transaction is concatenated with its sibling node then compared with the parent node’s hash. This process is done recursively until the Merkle Tree Root Node
4. Log (number of transactions)
5. 8,4,2
6. Some sort of file system?  
     
   Hash trees are also used in the IPFS, Btrfs and ZFS file systems (to counter data degradation); Dat protocol; Apache Wave protocol; Git and Mercurial distributed revision control systems; the Tahoe-LAFS backup system; Zeronet; the Bitcoin and Ethereum peer-to-peer networks; the Certificate Transparency framework; and a number of NoSQL systems such as Apache Cassandra, Riak, and Dynamo. (from <https://en.wikipedia.org/wiki/Merkle_tree>)

Git version control

## 1.3: Hash Functions

* 17476 Hashes/second
* 32 minutes
* 1024 minutes
* (2,3,5,5,7,2,5,0) - or anything else that outputs a digital root of 2 . (If you can’t be bothered computing the value, any permutation, or removing the 0 is easier..)

## 1.4: Privacy

### 1.4.1: CoinJoin

1. No linkability
2. Need to trust a third party to administer the transactions
3. Maybe use the system of pools to send coins from A to B. **A** send *x* coins to the pool and **B** is given a ZNP that he/she owes *x* coins in the pool

## 1.5: Mining and Proof-of-Work

1. Uncle block Ethereum <=> Orphan block bitcoin. Ethereum block generation times are much smaller than btc ones, so incentivize bc: increase bandwidth + decrease centralisation due to mining pools
2. Btc: increase difficulty so time to mine a block stays constant (measure mean of last n blocks). Ethereum other way around, increase/decrease based on computational power (not trying to keep time a constant), also update difficulty in each block.
3. “Smaller” difficulty than btc. I’m not sure the really do, but since blocks depend on the chosen transactions to include and their order n the tree, highly unlikely that the puzzle is the same across pools. The coinbase transaction (first tx of the block to reward the miner) is not the same across the pools because they don't have the same addresses for example. It means that the Merkle trees are different, so the headers are different. So the mined blocks are not exactly the same.
4. Can use the same nonce more than once because the entire block header is being hashed and here the merkle root might change as more transactions are being added to the memory pool, where this means that hashing with the same nonce at different times will result in different hashes.

# 

# 2: Wallets, Addresses and Transactions

1. In case it is mistyped/copied incorrectly it can be **fast** checked to see if its correct
2. Doesn’t include easily confused characters like (i, l, 0, O, + and /) (compared to Base64) [ <https://en.wikipedia.org/wiki/Base58> ] & human readable
3. Use real words (like an email-address) so easier to remember
4. It should also be added that most web wallets (one example is blockchain.info) encrypts the private key with your password. As the web wallet service does not store your password, your encrypted private key is secure, even if the web wallet service is hacked.

## 2.1: Transactions

### 2.1.2: Bitcoin Script

1. See course slides
2. double-spend? Blocks are not allowed to contain a transaction whose identifier matches that of an earlier, not-fully-spent transaction in the same chain. This rule is to be applied to all blocks whose timestamp is after a point in time
3. 250 bytes
4. Use smaller hashes?
5. Yes, if the second transaction gets mined before the first one

Or if you own 51% of hash power, simply rewrite the transaction and proposal a new block and make sure it longer enough to be the main chain.

1. Using spent addresses enables someone to easily trace transactions made and follow the flow of funds. Ideally, a different address is used for each transaction to minimize traceability.

### 2.1.3: Ethereum Transactions

1. See slides
2. \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_
3. \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

## 2.2: Blockchain Layer

1. 10 minutes
2. 15 seconds
3. Scalability vs security : faster block interval= faster transactions, but less security, slower= better security but slower transactions
4. The chain with the most PoW
5. PoS ?
6. Hard, depends (if it can be ignored by non-upgraded nodes, then soft, otherwise hard; see discussion in lecture on 19/02), hard, soft

* Increase of the blocksize from 1MB to 8MB. - hard?
* Introduction of a chainID field to differentiate between two forked version of the same chain for replay protection, e.g., Ethereum and Ethereum Classic. - HARD (<https://capital.com/ethereum-vs-ethereum-classic>)
* Redefinition of the OP VERIFY opcode to mark a transaction invalid if the top stack value is true - HARD
* Removal of the OP RETURN opcode, previously used to attach extra data to transactions - SOFT

1. Hardfork = backwards incompatible, all nodes should upgrade to the new consensus rules. Softfork = backwards compatible, no nodes need to upgrade thus old nodes can act as if nothing ever happened.

# 

# 3: Smart Contracts

## 3.1: Visibility

1. Only the creator of the contract as the owner variable is set when the contract is created and the function getting the secret requires the person calling the function to be the owner. (To add to this, anybody can read the actual value of “mySecret” since its stored on a public blockchain. So even if you have no assosiation with the contract, you can still read the value.)

## 3.2: Gas

1. During the transaction
2. Returned to sender
3. To make sure that no contracts take too long to execute, to prevent DOS attacks (i.e. someone making spam transactions bloating the blockchain and denying service to honest users)

## 3.3: Security

1. The balances map is adjusted after calling into the sender. The sender can call the contract again and empty out the entire contract using reentrancy as *value* does not restrict the amount of gas (assuming sufficient gas). The assertion amount <= balances[msg.sender] won't work because the internal state is updated after. (AFAIK the caller of the function has to be a contract that is purposly and maliciously written, overwritting their fallback function in order to recall the refund function of the honest contract. This cannot be done when a EOA calls the contract as the balances[ ] variable will be updated.)
2. function exploit() public {  
    v.refund(100);  
   }  
     
   function () public payable {  
    v.refund(100);  
   }
3. // ...  
   balances[msg.sender] -= amount;  
   msg.sender.call.value(amount)(""); // better msg.sender.transfer(amount); -- throws on error  
   // ...
4. transfer. Gas limit of 2300 prevents reentrancy and throws on error.
5. uint8(0) — unsigned addition has wraparound semantics