Exercise 6

1. (a) Name and describe the four use case examples of smart contracts from the lecture.

Solution:

- **Token systems** – Tokens are a sub-currency of Ethereum and represent a certain asset. These systems can for example be used for initial coin offerings (ICOs).
- **Identity and reputation systems** – In order to manage identities without a trusted third party, a smart contract can be used.
- **Decentralized Autonomous Organization** – A DAO is an organization that is controlled entirely by multiple smart contracts. It makes all decisions in a transparent way via its code. Its stakeholders can vote on the decisions to make.
- **Election and voting systems** – Smart contracts can record votes in a non-modifiable way and can make sure that wallet owners do not vote twice.

(b) There are many use cases for smart contracts. Think of three more and explain them as well.

Solution:

- **Gambling** – Smart contracts can be used to create lotteries, collectibles, pyramid schemes, and other games.
- **Token exchanges** – Some exchanges create a new smart contract for every user that registers for their service instead of having just one large account that holds all assets. That way, there is no single point-of-failure and this approach facilitates user token management.
- **Name registrars** – Name registrars provide human-readable names for addresses, similar to the Domain Name System (DNS) on the Internet.

(c) Find concrete addresses of contracts for five of the use cases from the previous two subtasks on the Ethereum mainnet. Explain briefly why a contract falls into a specific category. Hint: `https://etherscan.io/labelcloud` might help.

Solution:

- **Token system** – this is the ICO for the cryptocurrency EOS:
  `0x86Fa049857E0209aa7D9e616F7eb3b3B78ECfdb0`
- **DAO** – this is the contract of the original DAO:
  `0xBB9bc244D798123fDe783fCc1C72d3Bb8C189413`
- **Gambling** – this is the CryptoKitties smart contract:
  `0x06012c8cf97BEaD5deAe237070F9587f8E7A266d`
- **Cryptocurrency exchange** – this is a smart contract of the cryptocurrency exchange Bittrex:
  `0xa3C1E324CA1ce40db73eD6026c4A177F099B5770`
- **Name registrar** – this is the contract of the Ethereum Name System (ENS):
  `0x6090A6e47849629b7245Dfa1Ca21D94cd15878Ef`

2. (a) What is gas in the context of Ethereum?

Solution:
Every instruction of the EVM has a certain cost assigned, called gas. In general, more complex instructions cost more gas because they consume more computing power of the nodes in the network. A sender of a transaction sets a price in Ether that they are willing to pay per gas unit and a maximum amount of gas they want to spend. The transaction fee that the miner of the transaction gets to keep is the product of used gas units and gas price.

(b) What happens with spare gas if too much gas was provided for a transaction?

Solution:
If a transaction uses less gas than the sender provided, the excess amount gets refunded to the sender.

(c) Rank the following EVM instructions by their gas cost. Note: the `CREATE` instruction creates a new smart contract.

- `SHA3`
- `PUSH4 0x00000000`
- `SELFDESTRUCT` (if no new account is created)
- `JUMP`
- `CREATE`
- `CALL`
- `PUSH1 0xff`
- `STOP`

Solution:
gas(`SELFDESTRUCT`) < gas(`STOP`) < gas(`PUSH1 0xff`) = gas(`PUSH4 0x00000000`) < gas(`JUMP`) < gas(`SHA3`) < gas(`CALL`) < gas(`CREATE`)

3. In contrast to the transaction based ledger of Bitcoin, Ethereum uses an account based ledger to maintain the world state. In Bitcoin double spend attacks are prevented by UTXOs: Once a UTXO is spent, it cannot be spent again which creates a chain of coin ownership. How does Ethereum prevent double spends?

Solution:

Ethereum, being an account based ledger, cannot use UTXOs. Instead each Ethereum account has a number associated called a ”nonce” (Not to be confused with nonce, the random value in a block for mining). The account nonce is simply the number of transactions of an account and is included in transactions as a field.

A typical double spending attempt initiates a second transaction with a higher gas price after the original payment. The double spender hopes that miners will prefer mining the second transaction as it has a higher gas price. This is not possible as the second transaction must have a higher nonce and will be rejected by miners. If both transactions have the same nonce only one of them (most likely the second) will be mined and hence no double spending will take place.

Say the double spender has 10 ETH, a nonce of 5 and makes a 10 ETH payment with nonce 6. If they try to send 10ETH to another address they own using a nonce 7, miners will reject this transaction as the account nonce is 5 and transaction nonce is 7. If they publish another transaction with the nonce 6 sending 10 ETH to their account with a higher gas price, this will probably be mined instead of the initial payment. Hence only one of the 10 ETH spending transactions become mined.

4. In Ethereum there are two types of transactions. Whenever a wallet calls a method on a contract, a transaction is sent. Whenever a contract calls a method on another contract, a virtual message is sent (Sometimes referred as internal transactions). Messages exist "virtually" and are not written to the blockchain.

   Why are messages not published to blockchain? How could the blockchain be in the same state among all nodes without writing messages to the blockchain?

   Solution:

   The Ethereum Yellow Paper describes transactions as "Transaction: A piece of data, signed by an External Actor." Think of the Ethereum VM as a closed system and wallets interacting with the system and therefore changing the state.

   Ethereum is based on strict determinism. The state of the VM must be same across all nodes at a certain point. To reach the most recent state all nodes run the Ethereum VM and process mined blocks, which include transactions.

   Take a new node that just enters the system. First thing the node has to do is to sync with the current state of the blockchain. To achive this the node has to start from the genesis block and go through all blocks until the most recent one. During these steps the node will execute all transactions and also the messages resulting from transactions. In other words the node simulates all interactions of other wallets and all messages these interactions trigger. So each and every message will eventually be executed by the syncing node. Therefore there is no need to write the messages to the blockchain as the information is inherently contained in the transactions.

5. (a) Name four reasons why a transaction that is sent to the Ethereum network might not get mined.

   Solution:

   - The transaction has a wrong signature.
   - Sending account does not have enough funds.
   - Sender set a too low gas price to be added by a miner.
   - Transaction consumes more gas than the block gas limit.

   (b) Name two different ways how a mined transaction can fail.

   Solution:

- The transaction runs out of gas.
- The transaction gets reverted (e.g. by executing an illegal instruction, the `REVERT` instruction, or a `JUMP` to an illegal destination).

(c) Why does a mined transaction that fails still cost gas?

Solution:
In order to determine that a transaction fails, it has to be executed. At the point of failure, the transaction has already consumed computing power. Therefore, the sender should pay for it. This measure prevents spam. If a failed transaction would not cost any gas, a spammer could send many transactions with complex instructions that fail in the end. The spammer would not be charged but the transactions consume a lot of computing power from the members of the network.

6. In this exercise, we take a closer look at storing data on the Ethereum blockchain.

(a) What is the difference between memory and storage in Ethereum?

Solution:
Memory is part of the EVM and storage is part of a smart contract. The former is a non-persistent byte array used as temporal storage during execution of a single transaction. The latter is the data storage used by smart contracts which is persistent between multiple transactions.

(b) At a very low level, storing data to storage is performed by the EVM instruction `SSTORE`. According to the yellow paper, the gas cost of an `SSTORE` operation is 20,000 "when the storage value is set to non-zero from zero" and 5,000 "when the storage value's zeroness remains unchanged or is set to zero." Why is it cheaper to set a value to 0 than to set it from 0 to any other value?

Solution:
All storage items of a contract are initially empty. Setting a storage value from 0 to non-zero means that this value is newly initialized and all full nodes in the network have to keep track of the new data item. Every client has to create a new database entry for this item, which increases the size of the world state. As every node in the network has to allocate physical hard disk space, this operation is quite expensive. Once that space is allocated, changing the value is not increasing the size of the state, which is why this operation is cheaper.

(c) Calculate an estimation of how much it approximately costs (in USD) to store 1 KiB of data on the Ethereum blockchain at the moment. For simplicity, consider only the cost of `SSTORE` operations and disregard everything else. Use current usual values from Etherscan for your calculation.

Solution:

A word in Ethereum is 32 bytes long. For 1024 bytes, there are $\frac{1024}{32} = 32$ `SSTORE` operations necessary as every operation stores one word on the blockchain. We multiply the gas cost by the gas price in ETH and by the price of ETH in USD:
$32 \cdot 20000 \cdot 0.000000013689484729 \cdot 187.31 = 1.64$USD
Source: `https://etherscan.io/chart/gasprice` and `https://etherscan.io/chart/etherprice`

(d) Remix is an emulator to run and test Solidity code in browsers. Run the following test program in the Remix Emulator (`https://remix.ethereum.org/`). Empirically determine the real transaction gas cost of storing 1 KiB of data on the blockchain.

To do that copy and paste the code to Remix editor. Compile the code and click to Run section. Click "Deploy" button to deploy the smart contract to the blockchain. In the console below (in gray background) you will see the receipt of the deploy operation.

On below right you will notice deployed contracts. If you expand the section you can see the methods you can call in those contracts.

Now send a transaction using "put" method writing 1 KiB of data in the contract. The put method accepts an array of 32bytes. Therefore you will write 32 items. Go ahead and fill the following array to the parameter field of the `put()` method in Remix:
`["0x01", "0x02", "0x03", "0x04", "0x05", "0x06", "0x07", "0x08", "0x09", "0x0a", "0x0b", "0x0c", "0x0d", "0x0e", "0x0f", "0x10", "0x11", "0x12", "0x13", "0x14", "0x15", "0x16", "0x17", "0x18", "0x19", "0x1a", "0x1b", "0x1c", "0x1d", "0x1e", "0x1f", "0x20"]`.

Press put and find the transaction cost. Determine the price in USD as well.

```
pragma solidity ^0.5.1;

contract StorageTest {

    bytes32[] b;
    function put(bytes32[] memory _b) public {
        b = _b;
    }
}
```

Solution:
Approach:The transaction cost: 691,193 gas. The price in USD is $691193 \cdot 0.000000013689484729 \cdot 187.31 = 1.77$USD. The values may differ slightly

(e) What additional cost factors are responsible that the real gas price determined in subtask (d) is higher than the estimation from task (c)? Name three factors.

Solution:

- Instructions other than `SSTORE`
- Base cost of the transaction (21,000 gas)
- Fix cost of data input (68 gas for every non-zero input byte and 4 gas for every zero byte input)

Still, the `SSTORE` instructions make up more than 90% of the gas cost in this case.

7. In this exercise, we investigate one of the biggest hacks in the history of Ethereum, the first Parity multi-sig wallet hack from July 2017.

   (a) What is a multi-sig wallet? What is Parity?

   Solution:
   A multi-sig wallet is a contract that allows methods to be protected by requiring the approval of a fixed number of owners. Parity is an Ethereum client that came with some smart contract templates. Users were able to directly deploy some contracts to the blockchain through Parity's UI. Among these was a multi-sig wallet which provided a Solidity modifier `onlymanyowners(hash)` to protect methods. While the number of required signatures was not modifiable after it was initially set in the constructor, the set of owners was intended to be modifiable.

   (b) The hackers exploited a bug in the `initWallet()` function of the multi-sig wallet in order to wrongfully transfer funds from the affected contracts to their own account. What is the problem with this Solidity code? How could the bug have been prevented?

```
// constructor - just pass on the owner array to the multiowned and
// the limit to daylimit
function initWallet(address[] _owners, uint _required, uint _daylimit) {
  initDaylimit(_daylimit);
  initMultiowned(_owners, _required);
}
```

   Solution:
   Anyone can call the `initWallet()` function and set themselves as owner. The function does not have a visibility type and therefore defaults to `public`. The function should only be callable once, i.e. when the wallet has not been initialized yet. Also, `initDaylimit()` and `initMultiowned()` should have been declared `internal`. This is the fix: `https://github.com/paritytech/parity-ethereum/pull/6102`

   (c) The address of the hackers is `0xB3764761E297D6f121e79C32A65829Cd1dDb4D32`. Go to `https://etherscan.io` and investigate the oldest three transactions that this account was involved in. What steps did the hackers perform to steal Ether? Which smart contract was exploited first? Which functions were called? How much Ether did they obtain in this first step?

   Solution:
   At first, the hackers obtained 0.0298 Ether in a (probably) legit way in order to be able to pay for the transaction fees. Then, they called the `initWallet()` function of the smart contract at address `0x91EFffB9C6cd3A66474688D0a48AA6ECfe515AA5` and set themselves as owner of the contract. At the same time, they set the number of required owners to one and set the daily limit to 0x5ac7391989a21040000 Wei = 26,793 Ether. Finally, they called the `execute()` function of the same contract and transferred 26,793 Ether to their account.

   (d) How much ETH did the hackers steal in total? How many contracts were involved? Use Etherscan to determine how much it was worth in USD at that time.

   Solution:

On the internal txns page on Etherscan, there are three large incoming transactions. The hackers stole 26,793 + 44,055 + 82,189 = 153,037 ETH from these three different wallets. On July 19, 2017, this was worth 29.75 million USD. Source: `https://etherscan.io/chart/etherprice`