MetaMask:”0x3bCeedF1098A8FE5261225224b9E402E9D3c667B”
Discord:” metinaktug#8717”
GitHub:https://github.com/metinaktug/GitHub-https-github.com-metinaktug-ZKU_Background_Assignment_MetinAktug
e-mail:”mtn.aktug@gmail.com”

A. Conceptual Knowledge

1- A smart contract, like any other contract, sets the terms of an agreement. But unlike a traditional contract, the terms of a smart contract are executed in the form of code that runs on top of a blockchain like Ethereum. Smart contracts offer versatile peer-to-peer functionality, from credit and insurance to logistics and games, allowing developers to build applications that take advantage of the security, reliability and accessibility offered by blockchain.

2- Smart contracts are not free. Fees are paid, albeit very low, for the execution of these transactions. It would not be right to say "perfect" for smart contracts yet. Although it is very difficult to hack, it is not impossible at this stage, and at the same time, some mistakes that can be made by the people who prepared the codes can cause serious problems.

3- Cryptographic Hash functions produce an output of a certain length (for example, 20 bytes) regardless of the size of the input they receive. The output produced is specific to the given input. The same output always occurs with the same input and cannot be the same with the output of two different inputs. The output of the Cryptographic Hash function can be called "message digest", "hash", "checksum" and "digital fingerprint".

4- this analysis as follows: By showing the non-color-blind person's operational reactions to colors, we prove to the color-blind person that two objects of different colors are actually different colors. The proof here is verification. A person who is not colorblind is a validator.
another way is symbol notation. Here the colors are triple coded to 6 digits. If we show it as a cluster; {red (r), green (y),blue(b)}, {b,r,g}, {g,r,b}, {r,g,b}, {k,b,g}, {g, b, r}.
Zero-knowledge proof, the basic principle of the algorithm used in the field of data security in computer science is to prove the information you know to someone else without giving the information to him/her.
The way you prove your knowledge is generally the assumptions of an ideal cryptographic function.
A zero-knowledge proof must fulfill these three characteristics:
Completeness: If the evidence you provide is accurate and complete, the recipient will be assured that you have the information.
Accuracy: If your proof is not false, you can convince the buyer without fraud.
Keeping a Secret: If the statement is true, the buyer will understand. You can prove that you know with the example you give to the buyer.
The first two features above are generally the method of interacting with the buyer, and the third can be summarized as the method of proving.

Zero knowledge protocol is not a mathematical proof method, because in this proof method, it is tried to prove the information to the buyer by manipulating the information.
The ZKP buyer can also be persuaded by deception, although there is a small possibility. In such a case, some techniques are used to neglect the accuracy factor (trying to confuse by dwelling on probabilities, etc.).
One of the most common usage areas of ZKP is Turing machines. If we give an example of the usage area of Turing machines;
If there is such an expected PTV condition for any probabilistic polynomial time validator (PTV), we can adapt an interactive proof ZKP for that language.

B. You sure you’re solid with Solidity?

Storage.sol:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;
/**
 * @title Storage
 * @dev Store & retrieve value in a variable
 * @custom:dev-run-script ./scripts/deploy_with_ethers.ts
 */
contract storage {

    uint256 public number;

    /**
     * @dev Store value in variable
     * @param num value to store
     */
    function storeNumber(uint256 num) public {
        number = num;
    }

    /**
     * @dev Return value
     * @return value of 'number'
     */
    function retrieveNumber() public view returns (uint256){
        return number;
    }
}
```

Ballot.sol

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;
/// @title Voting with delegation.
contract Ballot {
    // This declares a new complex type which will
    // be used for variables later.
    // It will represent a single voter.
    struct Voter {
        uint weight; // weight is accumulated by delegation
        bool voted;  // if true, that person already voted
        address delegate; // person delegated to
        uint vote;   // index of the voted proposal
    }

    // This is a type for a single proposal.
    struct Proposal {
        bytes32 name;   // short name (up to 32 bytes)
        uint voteCount; // number of accumulated votes
    }

    address public chairperson;
    mapping(address => Voter) public voters;
    Proposal[] public proposals;

    uint256 [] public startTime = [block.timestamp, block.timestamp + 300]; // 300 seconds is
five minutes

      /// Create a new ballot to choose one of `proposalNames`.
    constructor(bytes memory proposalNames) {      // bytes32 replaced by bytes.
                                                   // "Dynamic-Size Types" is used
        chairperson = msg.sender;
        voters[chairperson].weight = 1;

        // For each of the provided proposal names,
        // create a new proposal object and add it
        // to the end of the array.
        for (uint i = 0; i < proposalNames.length; i++) {
            // `Proposal({...})` creates a temporary
            // Proposal object and `proposals.push(...)`
            // appends it to the end of `proposals`.
            proposals.push(Proposal({
                name: proposalNames[i],
                voteCount: 0
```

```
            }));
        }
    }

    // Give `voter` the right to vote on this ballot.
    // May only be called by `chairperson`.
    function giveRightToVote(address voter) external {
        // If the first argument of `require` evaluates
        // to `false`, execution terminates and all
        // changes to the state and to Ether balances
        // are reverted.
        // This used to consume all gas in old EVM versions, but
        // not anymore.
        // It is often a good idea to use `require` to check if
        // functions are called correctly.
        // As a second argument, you can also provide an
        // explanation about what went wrong.


        require(
            msg.sender == chairperson,
            "Only chairperson can give right to vote."
        );
        require(
            !voters[voter].voted,
            "The voter already voted."
        );
        require(voters[voter].weight == 0);
        voters[voter].weight = 1;
    }

    /// Delegate your vote to the voter `to`.
    function delegate(address to) voteEnded_timeNOK external {
        // assigns reference
        Voter storage sender = voters[msg.sender];

       //require((startTime [0] = block.timestamp) <= startTime [1], "5 minutes are up");
//voteEnded_timeNOK
        require(!sender.voted, "You already voted.");
        require(to != msg.sender, "Self-delegation is disallowed.");

        // Forward the delegation as long as
        // `to` also delegated.
        // In general, such loops are very dangerous,
```

```
        // because if they run too long, they might
        // need more gas than is available in a block.
        // In this case, the delegation will not be executed,
        // but in other situations, such loops might
        // cause a contract to get "stuck" completely.
        while (voters[to].delegate != address(0)) {
            to = voters[to].delegate;

            // We found a loop in the delegation, not allowed.
            require(to != msg.sender, "Found loop in delegation.");
        }

        // Since `sender` is a reference, this
        // modifies `voters[msg.sender].voted`
        sender.voted = true;
        sender.delegate = to;
        Voter storage delegate_ = voters[to];
        if (delegate_.voted) {
            // If the delegate already voted,
            // directly add to the number of votes
            proposals[delegate_.vote].voteCount += sender.weight;
        } else {
            // If the delegate did not vote yet,
            // add to her weight.
            delegate_.weight += sender.weight;
        }
    }

    /// Give your vote (including votes delegated to you)
    /// to proposal `proposals[proposal].name`.
    function vote(uint proposal) voteEnded_timeNOK external {
        Voter storage sender = voters[msg.sender];
        require(sender.weight != 0, "Has no right to vote");
        require(!sender.voted, "Already voted.");
        sender.voted = true;
        sender.vote = proposal;

        // If `proposal` is out of the range of the array,
        // this will throw automatically and revert all
        // changes.
        proposals[proposal].voteCount += sender.weight;
    }

    /// @dev Computes the winning proposal taking all
```

```
/// previous votes into account.
function winningProposal() public view
        returns (uint winningProposal_)
{
    uint winningVoteCount = 0;
    for (uint p = 0; p < proposals.length; p++) {
        if (proposals[p].voteCount > winningVoteCount) {
            winningVoteCount = proposals[p].voteCount;
            winningProposal_ = p;
        }
    }
}

// Calls winningProposal() function to get the index
// of the winner contained in the proposals array and then
// returns the name of the winner
function winnerName() external view
        returns (bytes32 winnerName_)
{
    winnerName_ = proposals[winningProposal()].name;
}

modifier voteEnded_timeNOK {
    _;
    require((startTime [0] = block.timestamp) <= startTime [1], "5 minutes are up");
}
}
```