LAB TASK- 5(b)

AIM: Write Solidity program to send ether from a Meta-mask account to another Meta-mask account through a smart contract.

DESCRIPTION:

This guide walks through writing, compiling, and deploying a simple Solidity smart contract to send Ether using Remix IDE. Solidity is the main language for Ethereum, enabling developers to build smart contracts and dApps. The contract features a function to transfer Ether from the sender to a specified address and includes a balance checker. Users will learn basic Solidity syntax, contract structure, the payable keyword, and how to interact with the contract through Remix. The contract is compiled and deployed using Remix's JavaScript VM or a connected MetaMask account, offering a practical introduction to smart contract-based Ether transfers.

PROCEDURE:

Prerequisites

- Two MetaMask accounts (e.g., Account A and Account B)
- Some test ETH in Account A (from Goerli or Sepolia faucet)
- Remix IDE: https://remix.ethereum.org

Step 1: Setup MetaMask

1. Install and open MetaMask extension in your browser.
2. Select a test network (e.g., Sepolia or Goerli).
3. Fund your account using a faucet:
    - Sepolia faucet: https://sepoliafaucet.com/
    - Goerli faucet: https://goerlifaucet.com/

Step 2: Open Remix and Create Contract

1. Visit https://remix.ethereum.org
2. Create a new Solidity file (e.g., EtherTransfer.sol)
3. Paste the following code:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;
contract EtherTransfer {
    address public owner;
    constructor() {    157440 gas 133000 gas
        owner = msg.sender;
    }
    function sendEther(address payable _recipient) public payable {
        require(msg.value > 0, "Send some Ether");
        _recipient.transfer(msg.value);
    }
    receive() external payable {}    undefined gas
}
```

Step 3: Compile the Contract

1. Go to the **Solidity Compiler** tab in Remix.
2. Select the correct compiler version (e.g., 0.8.0 or above).
3. Click **Compile EtherTransfer.sol**

Step 4: Deploy the Contract

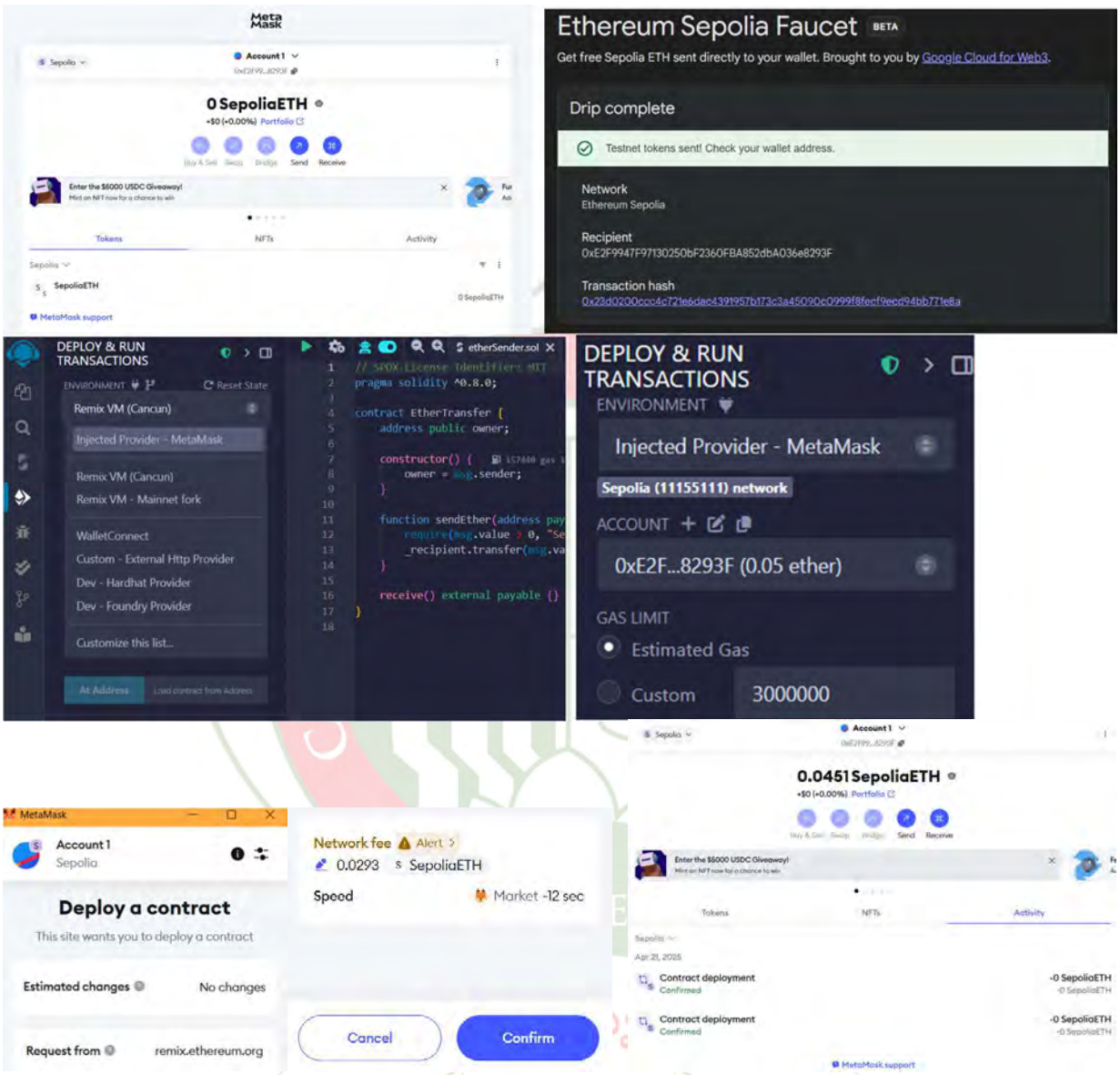
1. Go to the **Deploy & Run Transactions** tab.
2. Set **Environment** to Injected Provider - MetaMask (connects Remix to MetaMask).
3. Select your MetaMask account (Account A).
4. Click **Deploy** and approve the MetaMask transaction.

Step 5: Send Ether

1. After deployment, the contract appears under "Deployed Contracts."
2. Under sendEther, enter the recipient's address (e.g., MetaMask Account B).
3. Enter the amount in **ETH** in the "Value" field (top right corner in Remix).
4. Click **transact** and approve it in MetaMask.

OUTPUT:

The recipient (Account B) will receive the Ether. You can check using MetaMask or an Ethereum block explorer.



OUTPUT ANALYSIS:

1. Initial State: On deployment, owner is set to the deployer's address; contract balance is 0 ETH.

2. Successful Transfer: Calling sendEther() with a valid address and ETH transfers the exact amt to the recipient. MetaMask confirms the transaction.

3. Zero ETH Rejection: If no Ether is sent, the function reverts with **"Send some Ether"**, ensuring valid transfers only.

4. Fallback Enabled: Contract accepts direct ETH transfers via the receive() function.

LAB TASK- 5(c)

AIM: Write Solidity program to simulate a lottery game

DESCRIPTION:

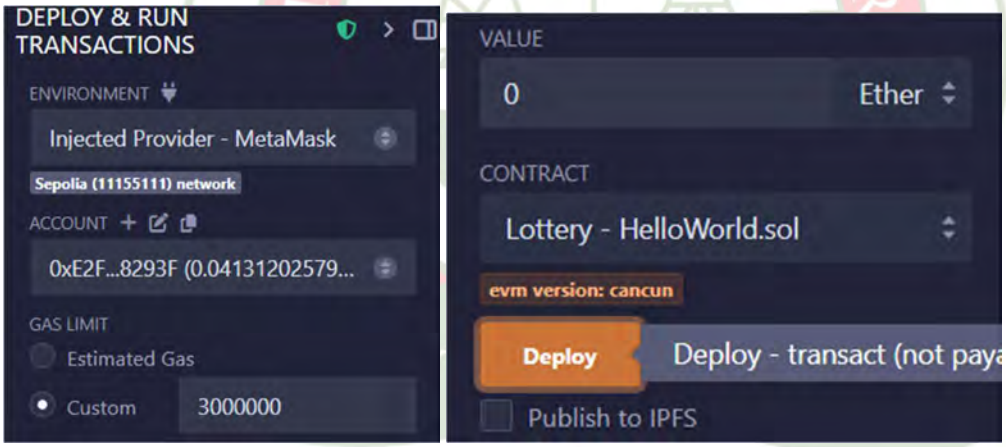
In this lottery game smart contract, participants can enter by sending Ether to the contract. Once the lottery is closed, a random winner is selected, and the prize (all the accumulated Ether in the contract) is sent to the winner. The contract owner can control when the lottery starts and ends. Additionally, the contract will handle the validation of participants and prevent issues like selecting the same person twice. The randomness in this case will be simplified for demonstration purposes (as true randomness requires an oracle or external solution).

CODE:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;
contract Lottery {
    address public owner;
    address[] public players;
    bool public isLotteryOpen;
    constructor() {  726036 gas 676800 gas
        owner = msg.sender;
        isLotteryOpen = false;
    }
    modifier onlyOwner() {
        require(msg.sender == owner, "Only the owner can perform this action");
        _;
    }
    function startLottery() public onlyOwner {  infinite gas
        require(!isLotteryOpen, "Lottery is already running");
        isLotteryOpen = true;
        delete players;
    }
    function enterLottery() public payable {  50890 gas
        require(isLotteryOpen, "Lottery is not open yet");
        require(msg.value == 0.1 ether, "Entry fee is 0.1 ETH");
        players.push(msg.sender);
    }
    function pickWinner() public onlyOwner {  infinite gas
        require(isLotteryOpen, "Lottery is not open");
        require(players.length > 0, "No players entered");
        uint randomIndex = uint(keccak256(abi.encodePacked(block.prevrandao, block.timestamp, players))) % players.length;
        address winner = players[randomIndex];
        payable(winner).transfer(address(this).balance);
        isLotteryOpen = false;
        delete players;
    }
    function getPlayers() public view returns (address[] memory) {  infinite gas
        return players;
    }
}
```

1. **Contract Owner:** The contract owner (deployer) is the only one who can start and close the lottery.
2. **Enter Lottery:** Players can enter by sending exactly 0.1 Ether, and they are added to the list of participants.
3. **Start Lottery:** The owner can start the lottery, which clears any previous participants and opens the lottery for a new round.
4. **Pick Winner:** After the lottery is closed, the owner can pick a winner randomly. The winner receives all the Ether accumulated in the contract.
5. **Randomness:** A simple random number generator is used based on block difficulty and timestamp. For more secure randomness, oracles like Chainlink VRF can be used.

OUTPUT:



Execution Steps

1. **Deployment**: Contract was deployed using Remix and MetaMask on the Sepolia testnet.
2. **Start Lottery**: startLottery() was called by the owner to begin the game.
3. **Enter Lottery**: Participants entered by calling enterLottery() with 0.1 ETH. MetaMask confirmed each transaction.
4. **Multiple Accounts**: Additional MetaMask accounts joined the lottery after receiving test ETH.
5. **Pick Winner**: Owner called pickWinner() to randomly select a winner and transfer the prize.
6. **View Players**: getPlayers() was used to view the current list of participants.

LAB TASK- 5(d)

AIM: Write Solidity program to demonstrate ERC20 tokens and to create a bank which deals with ERC20 tokens. The bank should provide lending, and borrowing of ERC20 tokens.

DESCRIPTION:

This Solidity program demonstrates the use of the ERC20 token standard by creating a custom token and integrating it into a decentralized bank. The bank enables users to **deposit**, **borrow**, and **repay** tokens. Users can lend their ERC20 tokens to the bank and earn interest, while others can borrow tokens with the condition of repayment. The contract maintains balances, enforces borrowing limits, and ensures secure token transfers using the transferFrom and approve mechanisms of the ERC20 standard. This system simulates a basic decentralized finance (DeFi) lending and borrowing model on the Ethereum blockchain.

CODE:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.18;

// Basic ERC20 Token
interface IERC20 {
    function totalSupply() external view returns (uint);   - gas
    function balanceOf(address account) external view returns (uint);   - gas
    function transfer(address recipient, uint amount) external returns (bool);   - gas
    function approve(address spender, uint amount) external returns (bool);   - gas
    function transferFrom(address sender, address recipient, uint amount) external returns (bool);
    function allowance(address owner, address spender) external view returns (uint);   - gas
    event Transfer(address indexed from, address indexed to, uint amount);
    event Approval(address indexed owner, address indexed spender, uint amount);
}

contract MyToken is IERC20 {
    string public name = "BankToken";
    string public symbol = "BTK";
    uint8 public decimals = 18;
    uint public override totalSupply;

    mapping(address => uint) public override balanceOf;
    mapping(address => mapping(address => uint)) public override allowance;

    constructor(uint _initialSupply) {   infinite gas 685000 gas
        totalSupply = _initialSupply * 10 ** uint(decimals);
        balanceOf[msg.sender] = totalSupply;
        emit Transfer(address(0), msg.sender, totalSupply);
    }

    function transfer(address recipient, uint amount) public override returns (bool) {   infinite
        require(balanceOf[msg.sender] >= amount, "Insufficient balance");
        balanceOf[msg.sender] -= amount;
        balanceOf[recipient] += amount;
        emit Transfer(msg.sender, recipient, amount);
        return true;
    }
```

```
    function approve(address spender, uint amount) public override returns (bool) {    infinite gas
        allowance[msg.sender][spender] = amount;
        emit Approval(msg.sender, spender, amount);
        return true;
    }

    function transferFrom(address sender, address recipient, uint amount) public override returns (bool) {
        require(balanceOf[sender] >= amount, "Insufficient balance");
        require(allowance[sender][msg.sender] >= amount, "Allowance exceeded");
        allowance[sender][msg.sender] -= amount;
        balanceOf[sender] -= amount;
        balanceOf[recipient] += amount;
        emit Transfer(sender, recipient, amount);
        return true;
    }
}

// Bank Contract
contract TokenBank {
    IERC20 public token;
    address public owner;

    mapping(address => uint) public deposits;
    mapping(address => uint) public loans;

    constructor(address _tokenAddress) {    infinite gas 733400 gas
        token = IERC20(_tokenAddress);
        owner = msg.sender;
    }

    function deposit(uint amount) public {    infinite gas
        require(token.transferFrom(msg.sender, address(this), amount), "Transfer failed");
        deposits[msg.sender] += amount;
    }
```

```
    function withdraw(uint amount) public {    infinite gas
        require(deposits[msg.sender] >= amount, "Insufficient deposit");
        deposits[msg.sender] -= amount;
        require(token.transfer(msg.sender, amount), "Transfer failed");
    }

    function borrow(uint amount) public {    infinite gas
        require(deposits[msg.sender] >= amount / 2, "Collateral too low (require 50%)");
        require(token.transfer(msg.sender, amount), "Borrow transfer failed");
        loans[msg.sender] += amount;
    }

    function repay(uint amount) public {    infinite gas
        require(token.transferFrom(msg.sender, address(this), amount), "Repayment failed");
        loans[msg.sender] -= amount;
    }

    function getDeposit() public view returns (uint) {    2541 gas
        return deposits[msg.sender];
    }

    function getLoan() public view returns (uint) {    2564 gas
        return loans[msg.sender];
    }
```

Steps in Remix

1. **Setup**: Open Remix IDE and paste the provided contract code into a new file.
2. **Compile**: Go to the **Solidity Compiler** tab, select version 0.8.18, and compile the contract.
3. **Deploy Token**: Deploy the MyToken contract with an initial supply (e.g., 1000 tokens). Copy the token contract address.
4. **Deploy Bank**: Deploy the TokenBank contract using the copied token address.

Interaction

1. **Approve Tokens**: Call approve(bankAddress, amount) in MyToken to allow the bank to use your tokens.
2. **Deposit**: Call deposit(amount) in TokenBank to deposit tokens.
3. **Borrow**: Call borrow(amount) in TokenBank (up to 50% of your deposit).
4. **Repay**: Approve the bank and call repay(amount) to repay the loan.
5. **Withdraw**: Call withdraw(amount) to get your tokens back.

OUTPUT ANALYSIS:

1. Approve Tokens - After calling approve(bankAddress, 500):
   - MetaMask prompts for confirmation.
   - Output: Bank is authorized to use 500 tokens.
2. Deposit Tokens - After calling deposit(100):
   - MetaMask prompts for confirmation.
   - Output: Bank deposit increases by 100 tokens (verify with getDeposit()).
3. Borrow Tokens - After calling borrow(50):
   - MetaMask prompts for confirmation.
   - Output: Loan balance increases by 50 tokens (verify with getLoan()).
4. Repay Loan - After calling repay(50):
   - MetaMask prompts for confirmation.
   - Output: Loan balance decreases to 0 (verify with getLoan()).
5. Withdraw Tokens - After calling withdraw(100):
   - MetaMask prompts for confirmation.
   - Output: Deposit decreases by 100 tokens (verify with getDeposit()).

LAB TASK- 6

AIM: Write a Solidity program to demonstrate Decentralized Autonomous Organizations (DAO) and liquidity pools.

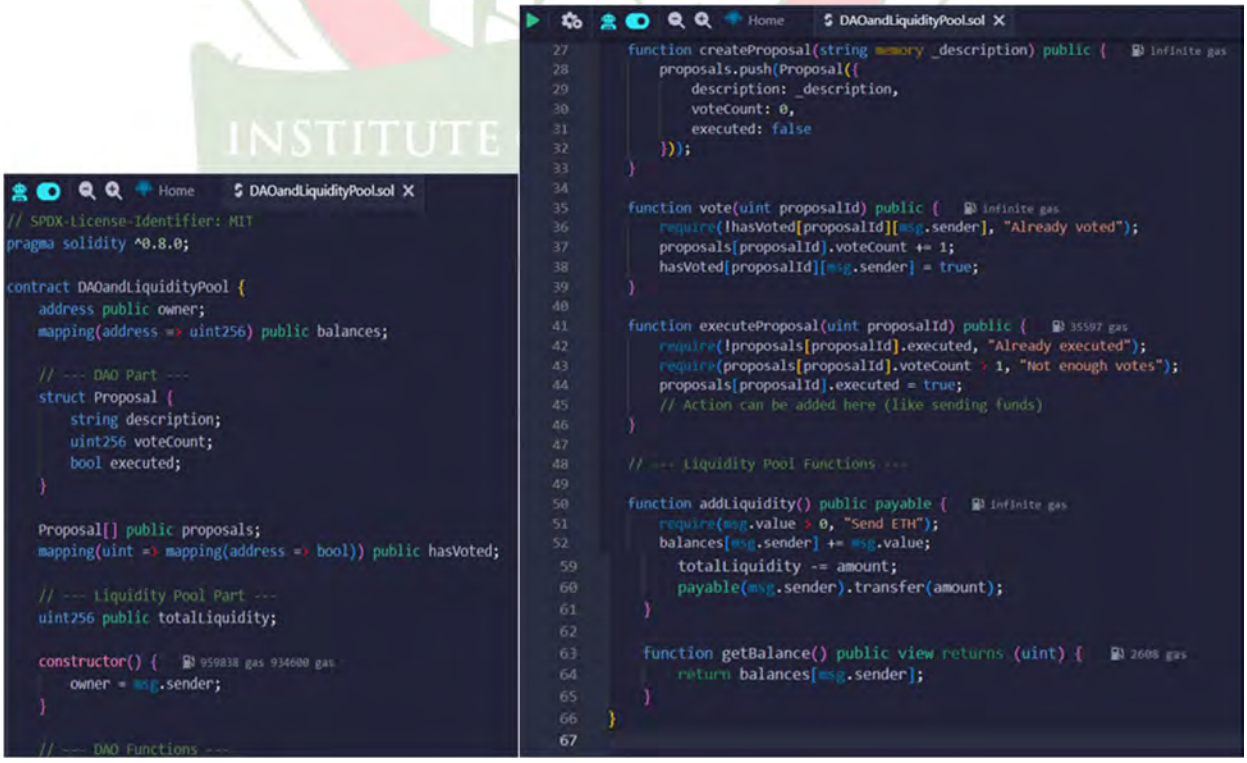
DESCRIPTION:

This experiment showcases the implementation of two key concepts in decentralized finance (DeFi):

- DAO (Decentralized Autonomous Organization): A system where users can contribute Ether to become stakeholders and vote on proposals to allocate funds.
- Liquidity Pool: A smart contract where users deposit Ether to create a pool. Liquidity providers can withdraw their share at any time.

PROCEDURE:

1. Open Remix IDE (https://remix.ethereum.org/)
2. Create a new Solidity file: DAOLiquidityPool.sol
3. Paste the smart contract code:

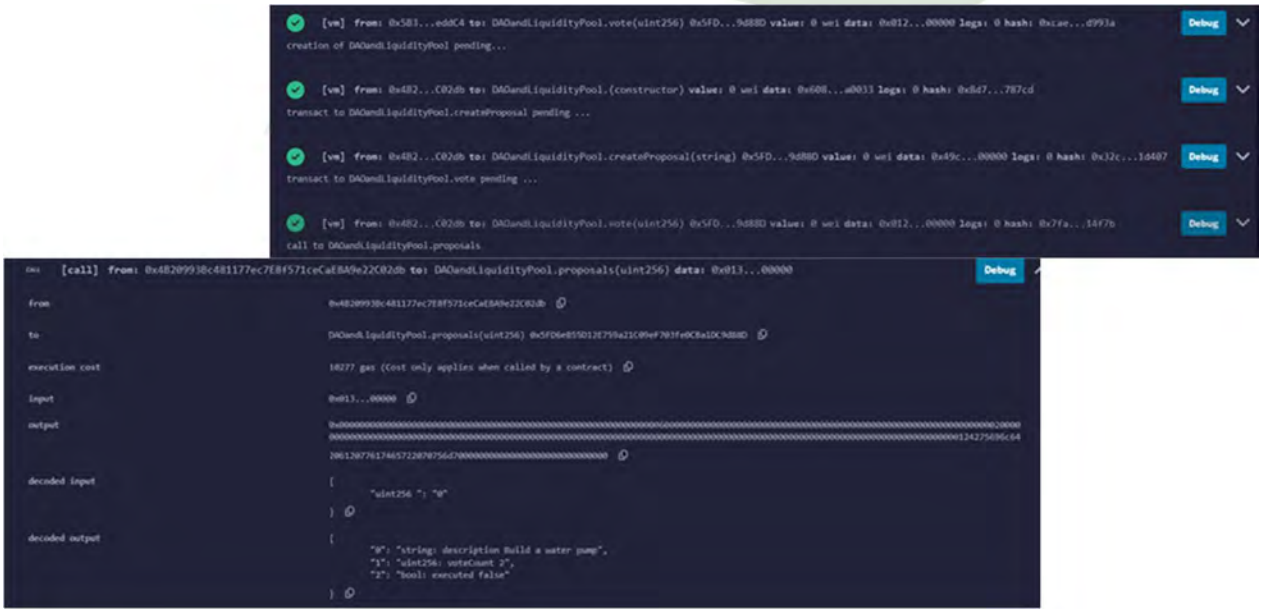


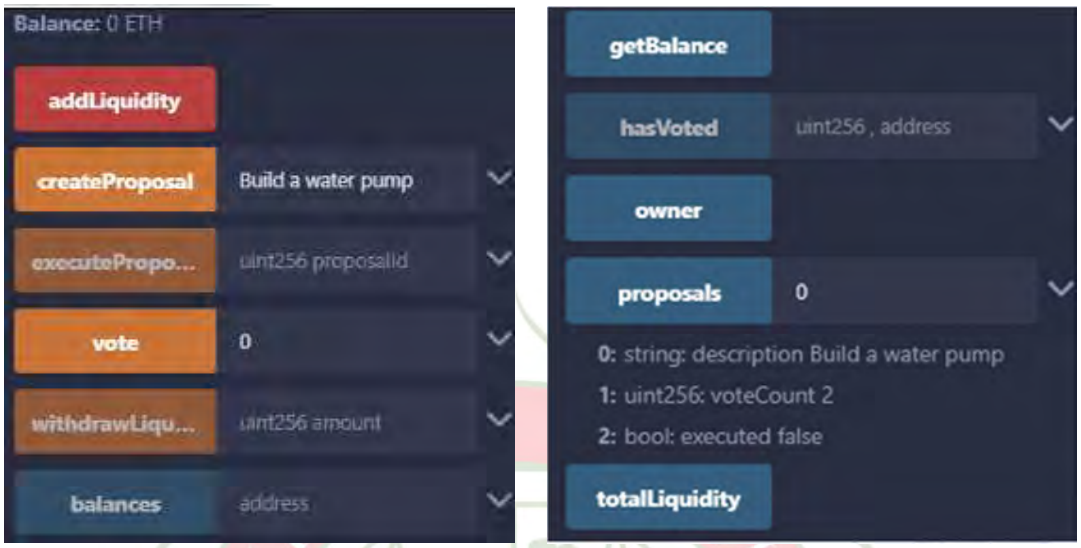
```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract DAOandLiquidityPool {
    address public owner;
    mapping(address => uint256) public balances;

    // --- DAO Part ---
    struct Proposal {
        string description;
        uint256 voteCount;
        bool executed;
    }

    Proposal[] public proposals;
    mapping(uint => mapping(address => bool)) public hasVoted;

    // --- Liquidity Pool Part ---
    uint256 public totalLiquidity;

    constructor() {
        owner = msg.sender;
    }

    // --- DAO Functions ---
    function createProposal(string memory _description) public {
        proposals.push(Proposal({
            description: _description,
            voteCount: 0,
            executed: false
        }));
    }

    function vote(uint proposalId) public {
        require(!hasVoted[proposalId][msg.sender], "Already voted");
        proposals[proposalId].voteCount += 1;
        hasVoted[proposalId][msg.sender] = true;
    }

    function executeProposal(uint proposalId) public {
        require(!proposals[proposalId].executed, "Already executed");
        require(proposals[proposalId].voteCount > 1, "Not enough votes");
        proposals[proposalId].executed = true;
        // Action can be added here (like sending funds)
    }

    // --- Liquidity Pool Functions ---
    function addLiquidity() public payable {
        require(msg.value > 0, "Send ETH");
        balances[msg.sender] += msg.value;
        totalLiquidity -= amount;
        payable(msg.sender).transfer(amount);
    }

    function getBalance() public view returns (uint) {
        return balances[msg.sender];
    }
}
```

4. Compile the contract using Solidity compiler (version 0.8.0 or above).
5. Deploy the contract using JavaScript VM environment.
6. Test DAO Functions:
   - Call contribute() with Ether using multiple accounts.
   - Create proposals with createProposal().
   - Vote using vote().
   - Execute winning proposal with executeProposal().
7. Test Liquidity Pool:
   - Call depositLiquidity() with Ether.
   - Call withdrawLiquidity() to withdraw deposited Ether.
   - Call getContractBalance() to check total pool balance.

OUTPUT:

Action	Output
Call `contribute()`	User becomes a stakeholder
Call `createProposal()`	New proposal created
Call `vote()`	Vote is registered
Call `executeProposal()`	Funds sent to recipient if majority is reached
Call `depositLiquidity()`	Ether added to pool
Call `withdrawLiquidity()`	Ether returned to user
Call `getContractBalance()`	Total balance shown





OUTPUT ANALYSIS:

DAO Functionality:

- Only stakeholders can create and vote on proposals.
- A proposal gets executed only if it receives more than 50% of the total votes.
- Funds are securely transferred to the specified recipient upon execution.

Liquidity Pool Functionality:

- Ether can be added and withdrawn by users.
- Contract maintains an internal mapping of each user's liquidity.
- The balance is updated correctly and users can withdraw any time.

CONCLUSION:

This experiment successfully demonstrated the core principles of a DAO and a Liquidity Pool using Solidity:

- Users can contribute and vote on decentralized proposals.
- Users can provide and withdraw liquidity.
- The contract enforces secure and transparent rules.

LAB TASK- 7

AIM: Write a Solidity program to demonstrate Multi-signature wallet, Time-locked wallet and Escrow contract.

DESCRIPTION:

This experiment showcases three key smart contract patterns commonly used for secure fund management in decentralized applications:

- Multi-signature Wallet: A contract requiring confirmations from multiple owners before executing a transaction, improving fund security.
- Time-locked Wallet: A wallet where funds can only be withdrawn after a specific time has passed.
- Escrow Contract: A contract involving a buyer, seller, and arbiter, ensuring safe release of funds only when both buyer and seller agree.

PROCEDURE:

1. Open Remix IDE.
2. Create a new Solidity file named WalletContracts.sol.
3. Paste the following code:

Contract 1: Multi-Signature Wallet

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract MultiSigWallet {
    address[] public owners;
    uint public approvalsNeeded;

    struct Transaction {
        address to;
        uint amount;
        uint approvalCount;
        bool executed;
    }

    mapping(address => bool) public isOwner;
    Transaction[] public transactions;
    mapping(uint => mapping(address => bool)) public approvals;

    constructor(address[] memory _owners, uint _approvalsNeeded) {
        require(_owners.length >= _approvalsNeeded, "Not enough owners");
        for (uint i = 0; i < _owners.length; i++) {
            isOwner[_owners[i]] = true;
        }
        owners = _owners;
        approvalsNeeded = _approvalsNeeded;
    }

    function approveTransaction(uint _txIndex) public {
        require(isOwner[msg.sender], "Only owners can approve");
        require(!approvals[_txIndex][msg.sender], "Already approved");

        approvals[_txIndex][msg.sender] = true;
        transactions[_txIndex].approvalCount++;

        if (transactions[_txIndex].approvalCount >= approvalsNeeded) {
            executeTransaction(_txIndex);
        }
    }

    function executeTransaction(uint _txIndex) internal {
        Transaction storage txn = transactions[_txIndex];
        require(!txn.executed, "Already executed");

        txn.executed = true;
        payable(txn.to).transfer(txn.amount);
    }

    receive() external payable {}
```

Contract 2: Time-Locked Wallet

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract TimeLockedWallet {
    address public owner;
    uint public unlockTime;

    constructor(uint _unlockTime) {    infinite gas 202200 gas
        owner = msg.sender;
        unlockTime = _unlockTime;
    }

    function deposit() public payable {}    185 gas

    function withdraw() public {    infinite gas
        require(msg.sender == owner, "Not the owner");
        require(block.timestamp >= unlockTime, "Too early!");
        payable(owner).transfer(address(this).balance);
    }

    function getBalance() public view returns (uint) {    312
        return address(this).balance;
    }
}
```

Contract 3: Escrow Contract

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract Escrow {
    address public buyer;
    address payable public seller;
    address public arbiter;
    uint public amount;

    constructor(address payable _seller, address _arbiter) payable {    infinite gas 207000 g
        buyer = msg.sender;
        seller = _seller;
        arbiter = _arbiter;
        amount = msg.value;
    }

    function releaseFunds() public {    infinite gas
        require(msg.sender == buyer || msg.sender == arbiter, "Only buyer or arbiter");
        seller.transfer(amount);
    }
}
```

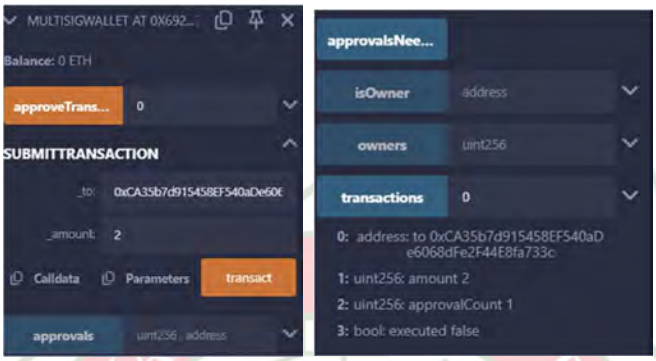
4. Compile the code using Solidity compiler version 0.8.0 or later.
5. Deploy each contract one by one:
   - MultiSigWallet – provide an array of 2-3 Ethereum addresses and required confirmations.
   - TimeLockedWallet – provide the unlock time using a Unix timestamp (e.g., block.timestamp + 60 for 1-minute lock).
   - Escrow – provide the buyer, seller, arbiter addresses and send some Ether.
6. Test the contracts:
   - For MultiSig:
     - Call submitTransaction() from one owner.

- Call confirmTransaction() from the other owners.
- Observe that transaction only executes when required confirmations are met.

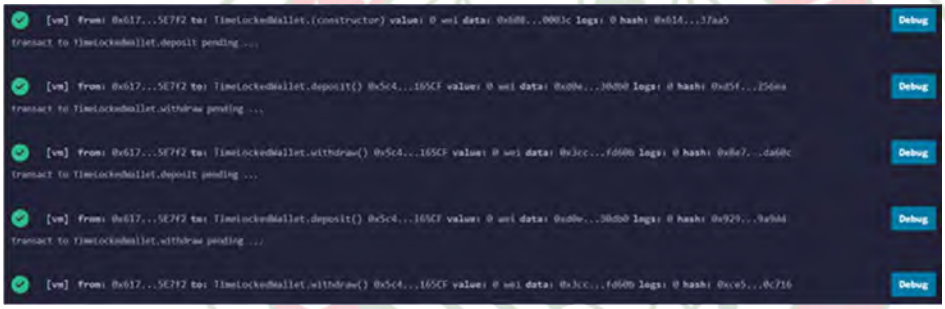
- For TimeLockedWallet:
  - Try calling withdraw() before unlock time — it should revert.
  - Wait for the time to pass, then call withdraw() to receive Ether.
- For Escrow:
  - Call approveByBuyer() and approveBySeller() — funds go to seller.
  - If dispute, call refundToBuyer() from arbiter.

OUTPUT:

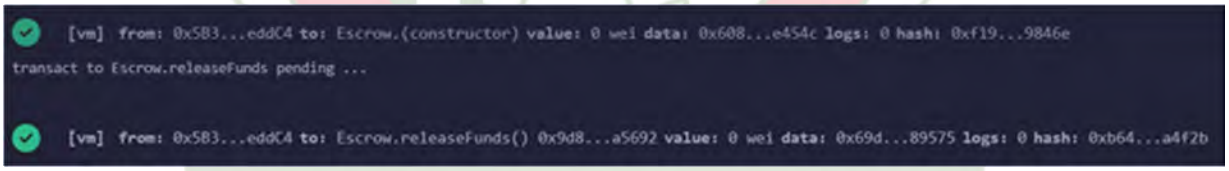
Contract	Function Tested	Result
MultiSigWallet	`submitTransaction`	Transaction added
MultiSigWallet	`confirmTransaction`	Executed only after required approvals
TimeLockedWallet	`withdraw` (before unlock)	Reverted (locked)
TimeLockedWallet	`withdraw` (after unlock)	Ether withdrawn successfully
Escrow	`approveByBuyer & Seller`	Seller received payment
Escrow	`refundToBuyer`	Buyer refunded (if only 1 approval)



Contract 2: Time-Locked Wallet



Contract 3: Escrow Contract



OUTPUT ANALYSIS:

Multi-signature wallet ensures consensus among multiple owners before spending funds. Unauthorized access is not possible. Time-locked wallet enforces temporal restrictions, preventing premature access to funds. Escrow contract maintains fairness in payments by requiring mutual approval or arbiter intervention.

CONCLUSION:

This experiment successfully demonstrated the implementation and testing of:

- A multi-signature wallet for collaborative fund management.
- A time-locked wallet to restrict access until a specific time.
- An escrow contract to facilitate secure transactions between untrusted parties.

LAB TASK- 8

AIM: Write a Solidity program to track provenance and movement of goods through the supply chain, ensuring transparency and authenticity (Supply-Chain)

DESCRIPTION:

In a supply chain, products move from manufacturers to distributors, then to retailers, and finally to customers. Tracking this journey is often opaque, vulnerable to fraud, and lacks real-time traceability. This experiment uses Solidity to develop a decentralized application (dApp) to:

- Record the origin and status of goods at each stage.
- Ensure that all updates are transparent and immutable.
- Allow any user to verify the authenticity and current status of a product.

Each product is identified by a unique ID and its details (owner, location, status) are recorded and updated as it moves through the supply chain.

PROCEDURE:

1. Open Remix IDE – Go to Remix Ethereum IDE.
2. Create a new file – Name it SupplyChain.sol and paste the above code.
3. Compile the contract – Select Solidity Compiler tab, set version to 0.8.x and click Compile SupplyChain.sol.
4. Deploy the contract – In the "Deploy & Run Transactions" tab:
   - Select "JavaScript VM" environment.
   - Click Deploy.
5. Interact with the contract:
   - createProduct: Enter ID (e.g., 1), Name (e.g., "Rice"), andLocation (e.g., "Farm").
   - getProduct: Use ID to check product details.
   - updateProduct: Use ID, new location (e.g., "Warehouse"), and new status (1 for InTransit, 2 for Delivered).
   - transferOwnership: Use ID and new owner's address (select another account from Remix dropdown).

```solidity
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract SupplyChain {
    enum State { Manufactured, Supplied, Retail, Sold }

    struct Product {
        uint id;
        string name;
        State state;
        address owner;
    }

    mapping(uint => Product) public products;
    uint public productCount;

    // Event for each movement
    event StateChanged(uint productId, State state, address owner);

    // Add new product by Manufacturer
    function manufactureProduct(string memory _name) public {  infinite gas
        productCount++;
        products[productCount] = Product(productCount, _name, State.Manufactured, msg.sender);
        emit StateChanged(productCount, State.Manufactured, msg.sender);
    }
```

```solidity
    // Move to Supplier
    function supplyProduct(uint _id) public {  infinite gas
        require(products[_id].state == State.Manufactured, "Not manufactured yet");
        products[_id].state = State.Supplied;
        products[_id].owner = msg.sender;
        emit StateChanged(_id, State.Supplied, msg.sender);
    }

    // Move to Retail
    function retailProduct(uint _id) public {  infinite gas
        require(products[_id].state == State.Supplied, "Not supplied yet");
        products[_id].state = State.Retail;
        products[_id].owner = msg.sender;
        emit StateChanged(_id, State.Retail, msg.sender);
    }

    // Sold to Consumer
    function sellProduct(uint _id) public {  infinite gas
        require(products[_id].state == State.Retail, "Not in retail yet");
        products[_id].state = State.Sold;
        products[_id].owner = msg.sender;
        emit StateChanged(_id, State.Sold, msg.sender);
    }
```

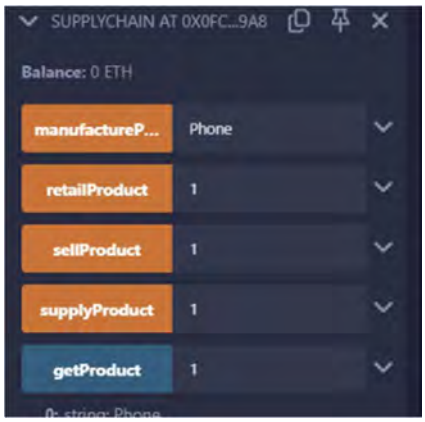
```solidity
    // View product details
    function getProduct(uint _id) public view returns(string memory, State, address) {  infinite gas
        Product memory p = products[_id];
        return (p.name, p.state, p.owner);
    }
}
```

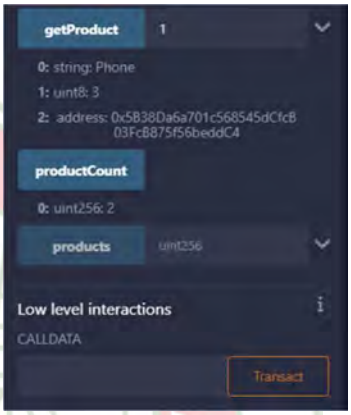
OUTPUT:

Sample interaction:

1. Call createProduct(1, "Wheat", "Farm")
2. Call getProduct(1)
   Output: ("Wheat", "Farm",<creator_address>, 0)
3. Call updateProduct(1, "Factory", 1)
4. Call getProduct(1)
   Output: ("Wheat", "Factory", , 1)
5. Call transferOwnership(1,)
6. Call getProduct(1)
   Output: owner is now

```
[vm] from: 0x5B3...eddC4 to: SupplyChain.(constructor) value: 0 wei data: 0x608...00033 logs: 0 hash: 0x514...aeb39
transact to SupplyChain.manufactureProduct pending ...

[vm] from: 0x5B3...eddC4 to: SupplyChain.manufactureProduct(string) 0x0fC...9A836 value: 0 wei data: 0xd64...00000 logs: 1 hash: 0x427...200c
transact to SupplyChain.manufactureProduct pending ...

[vm] from: 0x5B3...eddC4 to: SupplyChain.manufactureProduct(string) 0x0fC...9A836 value: 0 wei data: 0xd64...00000 logs: 1 hash: 0x0c9...d6d0
transact to SupplyChain.supplyProduct pending ...

[vm] from: 0x5B3...eddC4 to: SupplyChain.supplyProduct(uint256) 0x0fC...9A836 value: 0 wei data: 0x6b3...00001 logs: 1 hash: 0xfd2...fd778
transact to SupplyChain.retailProduct pending ...

[vm] from: 0x5B3...eddC4 to: SupplyChain.retailProduct(uint256) 0x0fC...9A836 value: 0 wei data: 0xff4...00001 logs: 1 hash: 0xe9d...87c37
transact to SupplyChain.sellProduct pending ...

[vm] from: 0x5B3...eddC4 to: SupplyChain.sellProduct(uint256) 0x0fC...9A836 value: 0 wei data: 0x492...00001 logs: 1 hash: 0xf87...21980
call to SupplyChain.getProduct
```





OUTPUT ANALYSIS:

1. The smart contract records each product's name, location, current owner, and status.
2. All updates are only allowed by the current owner, ensuring authenticity.
3. Status and ownership changes are reflected immediately and immutably.
4. Anyone can view the product data using getProduct, achieving full transparency.

CONCLUSION:

This experiment successfully demonstrates how a blockchain-based solution using Solidity can enhance transparency, trust, and traceability in the supply chain. Every change in the lifecycle of a product is securely recorded, and tampering is prevented through ownership verification and immutable logging, showcasing the real-world utility of smart contracts.

LAB TASK- 9

AIM: Write a Solidity program that automatically pays out claims based on predefined conditions eliminating the need for intermediate (Insurance)

DESCRIPTION:

This experiment demonstrates the use of Ethereum smart contracts to create a decentralized insurance model. A farmer is insured against poor rainfall. If the rainfall goes below a threshold (e.g., 50 mm), the contract automatically pays out a predefined amount (1 ether) to the farmer. The contract:

- Accepts insurance funds from any user (like an insurer or the government).
- Takes the farmer's address during deployment.
- Allows an update to rainfall (simulating an oracle).
- Automatically triggers a payout if rainfall is too low and funds are available.

This removes the need for third-party insurance companies to manually verify and process claims.

PROCEDURE:

1. Open Remix IDE – Go to Remix Ethereum IDE.
2. Create a new file – Name it Insurance.sol and paste the smart contract code.
3. Select Compiler:
   - Go to the "Solidity Compiler" tab.
   - Set the compiler version to 0.8.0 or higher.
   - Click Compile Insurance.sol.
4. Deploy the Contract:
   - Go to "Deploy & Run Transactions" tab.
   - Choose "JavaScript VM" environment.
   - In the constructor input, paste any Ethereum address (one from the dropdown list).
   - Click Deploy.

5. Fund the Contract:
   - In the "Value" field above the fundInsurance button, enter 1 and change the dropdown next to it to ether.
   - Click fundInsurance to add money to the contract.
6. Trigger the Payout:
   - In the updateRainfall function, enter a value less than 50 (e.g., 30).
   - Click the button. If funds are available, the farmer is paid automatically.
7. Check Balances:
   - Use getBalance to check the contract's balance.
   - Use the Remix console or "Account" dropdown to check the farmer's balance.

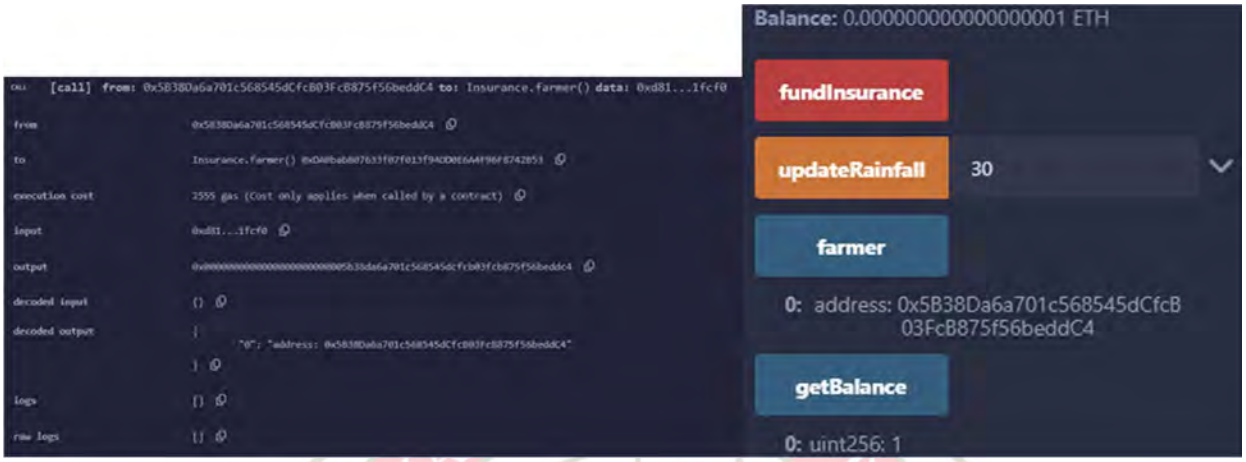
```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract Insurance {
    address public farmer;
    address public insurer;
    uint public rainfall; // in mm
    uint public payoutAmount = 1 ether;
    bool public isClaimed = false;

    constructor() {    334899 gas 252000 gas
        farmer = msg.sender;
        insurer = address(this); // Contract holds the money
    }

    // Anyone can fund the contract (e.g., government, company, etc.)
    function fundInsurance() public payable {}    165 gas

    // Oracle (external system) updates the rainfall
    function updateRainfall(uint _rainfall) public {    infinite gas
        rainfall = _rainfall;
        if (rainfall < 50 && !isClaimed) {
            payout();
        }
    }

    function payout() internal {    infinite gas
        require(address(this).balance >= payoutAmount, "Not enough funds");
        payable(farmer).transfer(payoutAmount);
        isClaimed = true;
    }

    // Check contract balance
    function getBalance() public view returns (uint) {    362 gas
        return address(this).balance;
    }
}
```

OUTPUT:

After calling updateRainfall(30) and meeting the payout condition:

- 1 ether is automatically transferred to the farmer.
- The contract's balance is reduced.
- The isClaimed variable becomes true, preventing duplicate claims.

```
[vm] from: 0x583...eddC4 to: Insurance.fundInsurance() 0xDA0...42B53 value: 1000000000000000000 wei data: 0x12e...acf4b logs: 0 hash: 0x054...2c62
transact to Insurance.updateRainfall pending ...

[vm] from: 0x583...eddC4 to: Insurance.updateRainfall(uint256) 0xDA0...42B53 value: 0 wei data: 0x183...0001e logs: 0 hash: 0x8cc...638c6
call to Insurance.getBalance

[call] from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to: Insurance.getBalance() data: 0x120...65fe0
call to Insurance.farmer

[call] from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to: Insurance.farmer() data: 0xd81...1fcf0
```



OUTPUT ANALYSIS:

1. When rainfall was above 50 mm, no payout occurred, and isClaimed remained false.
2. When rainfall was below 50 mm, a payout was made only once, showing correct conditional automation.
3. Any attempt to call updateRainfall() again with low values after payout resulted in no further payout — proving the claim was marked as complete.

CONCLUSION:

This experiment successfully demonstrates how smart contracts on the Ethereum blockchain can be used to automate insurance claim processes without intermediaries. The use of predefined conditions ensures fast, transparent, and trustless payouts. It is a practical example of how blockchain technology can improve traditional systems like agriculture insurance through decentralization and automation.

LAB TASK- 10

AIM: Write a Solidity program to conduct secure and transparent voting processes without relying on central authority (Voting Systems)

DESCRIPTION:

In traditional voting systems, there's often a central authority responsible for organizing, validating, and counting votes. This creates a single point of trust—and potential failure. Using blockchain technology and smart contracts, we can build a decentralized voting platform that:

- Records votes immutably and Prevents double voting.
- Allows anyone to verify the results transparently.
- Requires no central authority for validation.

In this system:

- A list of candidates is initialized.
- Voters can vote once.
- The contract stores and tallies all votes.
- Final results can be viewed publicly and securely.

PROCEDURE:

1. Open Remix IDE – https://remix.ethereum.org
2. Create a file – Name it VotingSystem.sol and paste the code above.
3. Compile the contract – In the "Solidity Compiler" tab, choose version 0.8.x, click Compile.
4. Deploy the contract – In the "Deploy & Run Transactions" tab:
    a. Choose JavaScript VM.
    b. Click Deploy.
5. Add Candidates:
    a. Call addCandidate("Alice")
    b. Call addCandidate("Bob")
6. Switch to other accounts in Remix and vote:
    a. Select another account from the top dropdown.
    b. Call vote(1) for Alice or vote(2) for Bob.
    c. Repeat with other accounts to simulate voting.
7. End Voting: Switch back to the admin account and Call endVoting().

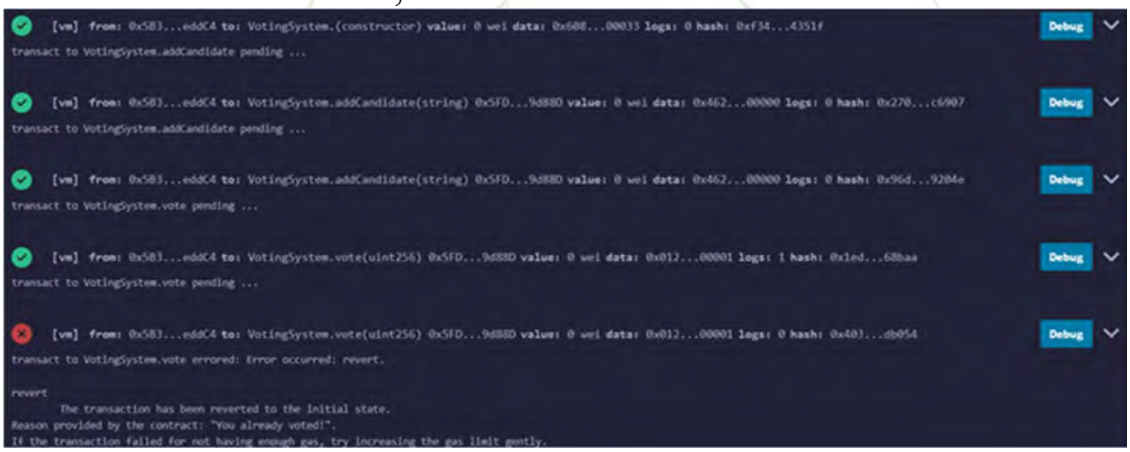
8. View Winner: • Call getWinner() to see the winner's name and number of votes.

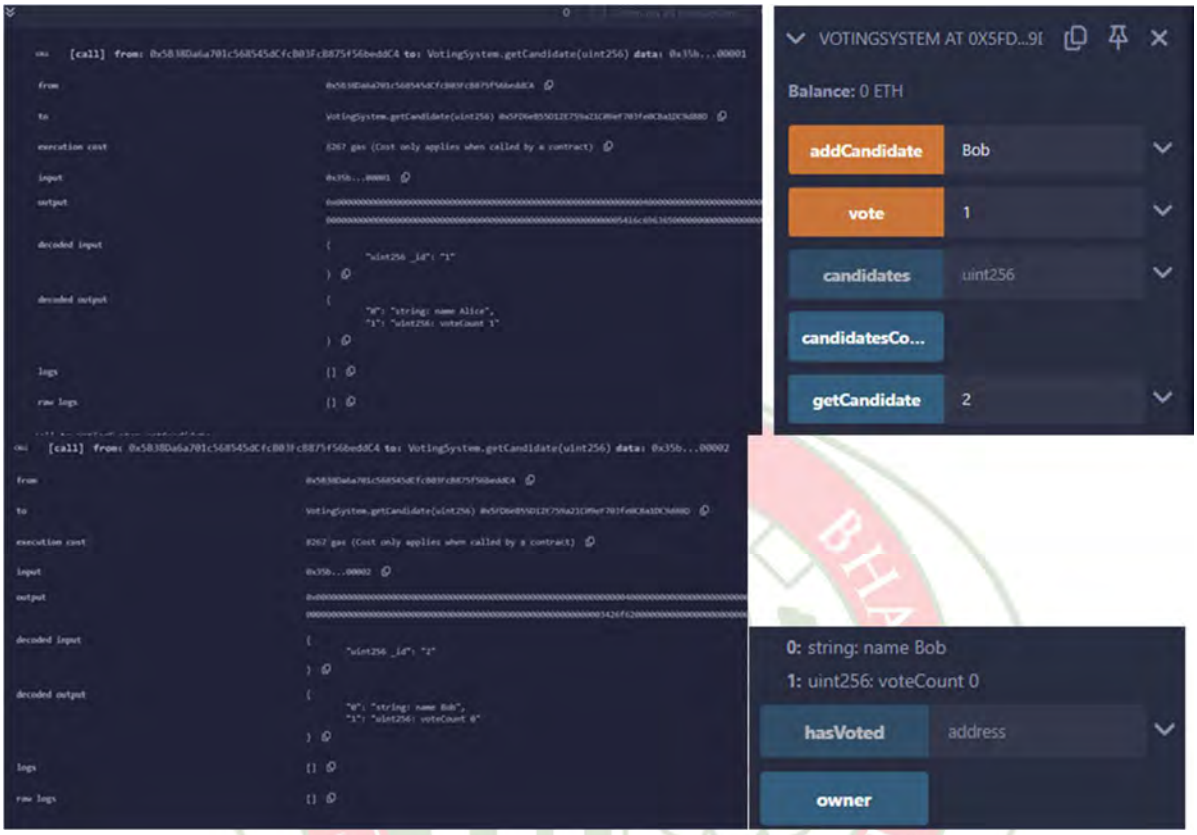
```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract VotingSystem {
    address public owner;

    struct Candidate {
        uint id;
        string name;
        uint voteCount;
    }

    uint public candidatesCount;
    mapping(uint => Candidate) public candidates;

    mapping(address => bool) public hasVoted;

    event Voted(uint candidateId);

    constructor() {
        owner = msg.sender;
    }

    // Add candidates before voting starts
    function addCandidate(string memory _name) public {
        require(msg.sender == owner, "Only owner can add candidates");
        candidatesCount++;
        candidates[candidatesCount] = Candidate(candidatesCount, _name, 0);
    }

    // Vote for a candidate
    function vote(uint _candidateId) public {
        require(!hasVoted[msg.sender], "You already voted!");
        require(_candidateId > 0 && _candidateId <= candidatesCount, "Invalid candidate");

        hasVoted[msg.sender] = true;
        candidates[_candidateId].voteCount++;

        emit Voted(_candidateId);
    }

    // View candidate details
    function getCandidate(uint _id) public view returns (string memory name, uint voteCount) {
        Candidate memory c = candidates[_id];
        return (c.name, c.voteCount);
    }
}
```

OUTPUT:

Example:

- Candidates: Alice (ID 1), Bob (ID 2)
- Votes: Voter A → Alice, Voter B → Bob, Voter C → Alice
- Winner: Name: Alice, Votes: 2





OUTPUT ANALYSIS:

- Voters are only allowed to vote once; second attempts are blocked.
- The winner is determined accurately only after voting is ended.
- Votes are immutable and transparent—anyone can verify the results.
- The use of different accounts simulates multiple voters and provides realism to the test.

CONCLUSION:

This experiment shows how a decentralized voting system can be built using smart contracts on the Ethereum blockchain. It ensures:

- Security (no unauthorized voting),
- Transparency (all votes visible),
- Fairness (no double voting),
- And trustless execution (no need for central authority).