LAB TASK- 1

AIM: Develop a Java/Python/Go program to create Elliptic curve public and private keys and demonstrate working of hash functions like SHA256 and ECC digital signatures.

DESCRIPTION:

This program demonstrates **Elliptic Curve Cryptography (ECC)** by generating public and private key pairs, hashing data using **SHA-256**, and implementing **Elliptic Curve Digital Signature Algorithm (ECDSA)** for message signing and verification. ECC is widely used for secure communications due to its strong security with smaller key sizes compared to RSA.

Key Features:

1. **Elliptic Curve Key Pair Generation**
   - Uses the secp256r1 curve to create a secure key pair.
   - Ensures strong encryption with minimal computational overhead.
2. **SHA-256 Hashing**
   - Converts input data into a fixed 256-bit hash for integrity verification.
   - Essential for digital signatures and secure password storage.
3. **ECDSA Digital Signature**
   - Signs a message using the private key to ensure authenticity.
   - Verifies the signature using the public key, preventing tampering.

Libraries Used:

- **Java** – Bouncy Castle, java.security
- **Python** – ecdsa, hashlib
- **Go** – crypto/ecdsa, crypto/sha256

Applications:

- **Blockchain Transactions**
- **Secure Authentication & Digital Certificates**
- **IoT Secure Communications**

This implementation provides a strong foundation for cryptographic security, ensuring data integrity, authenticity, and non-repudiation in secure systems.

CODE:

ECC.py

C: > Users > HP > Downloads > ECC.py > ...

```python
import hashlib
from ecdsa import SECP256k1, SigningKey, VerifyingKey
def generate_keys():
    signing_key = SigningKey.generate(curve=SECP256k1)
    verifying_key = signing_key.get_verifying_key()
    return signing_key, verifying_key
def create_sha256_hash(message):
    return hashlib.sha256(message.encode()).hexdigest()
def sign_message(signing_key, message):
    message_hash = create_sha256_hash(message)
    signature = signing_key.sign(message_hash.encode())
    return signature
def verify_signature(verifying_key, message, signature):
    message_hash = create_sha256_hash(message)
    try:
        verifying_key.verify(signature, message_hash.encode())
        print("Signature is valid.")
    except:
        print("Signature is invalid.")
private_key, public_key = generate_keys()
print("Private Key:", private_key.to_string().hex())
print("Public Key:", public_key.to_string().hex())
message = "This is a secret message."
signature = sign_message(private_key, message)
print("Signature:", signature.hex())
verify_signature(public_key, message, signature)
```


OUTPUT:

```
PS C:\Users\HP> & C:/Users/HP/AppData/Local/Programs/Python/Python312/python.exe c:/Users/HP/Downloads/ECC.py
Private Key: 1de7958ecc3bf87a6bffd371471ba6ce81e4e0e15769751b5d433fec26851e7c
Public Key: c65e656210406e2a36018e69a4c617e9e7acadfb54d63cb78b588808db80063b6565082ff61d507a20ba537a7f0c42993
Signature: 9c5d84e0dd0a635f65e3eefd16975582aad160d0a9a45e140cadb80ed414eae0e7c40499f164926b346acc78a182bac485
Signature is valid.
```

OUTPUT ANALYSIS:

The output includes the **EC key pair**, a **SHA-256 hash**, and an **ECDSA signature**. The signature verification step confirms authenticity, displaying **"Valid Signature"** if unchanged or **"Invalid Signature"** if altered. This ensures **data integrity, security, and efficiency**, making ECC ideal for **secure communications and blockchain applications**.

LAB TASK- 2

AIM: Setup a Bitcoin wallet like Electrum and demonstrate sending and receiving Bitcoins on a testnet. Use Blockchain explorer to observe the transaction details.

DESCRIPTION:

This guide demonstrates setting up an Electrum Bitcoin wallet on a **testnet**, allowing users to **send and receive testnet Bitcoins** for practice without real monetary risk. The testnet is a parallel Bitcoin network designed for development and testing, where transactions function just like on the mainnet but use worthless coins.

By configuring Electrum for the testnet, users can **generate Bitcoin addresses, receive testnet BTC from faucets, and send transactions** to other testnet users. Once a transaction is made, a **blockchain explorer** is used to track and verify details like transaction ID, sender, receiver, fees, and confirmations.

This demonstration helps understand **Bitcoin wallet operations, address generation, transaction fees, confirmations, and blockchain transparency**. It is useful for developers and cryptocurrency enthusiasts to practice securely before handling real Bitcoin.

PROCEDURE:

1. Download and Install Electrum

- Visit electrum.org and download the latest version for your operating system.
- Install and launch Electrum.

2. Configure Testnet Mode

- Open a **terminal** or **command prompt** and run:

```
bash
electrum --testnet
```

- If using the **GUI**, enable testnet mode in **Tools → Network → Testnet**.

3. Create a New Wallet

- Select **Standard Wallet** and choose **Create a New Seed**.
- Save the **12-word seed phrase** securely.
- Set a strong password and confirm wallet creation.

4. Receive Testnet Bitcoins

- Click **Receive** and copy your **testnet Bitcoin address**.
- Visit a **testnet faucet** (e.g., https://testnet-faucet.mempool.co) to request free BTC.
- Wait for confirmation (may take minutes).

5. Send Testnet Bitcoin

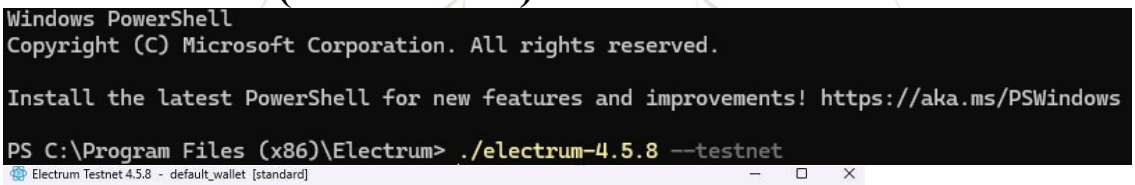
- Click **Send**, enter the recipient's testnet Bitcoin address, and specify the amount.
- Set a transaction fee (higher fees confirm faster).
- Click **Send** and confirm the transaction.

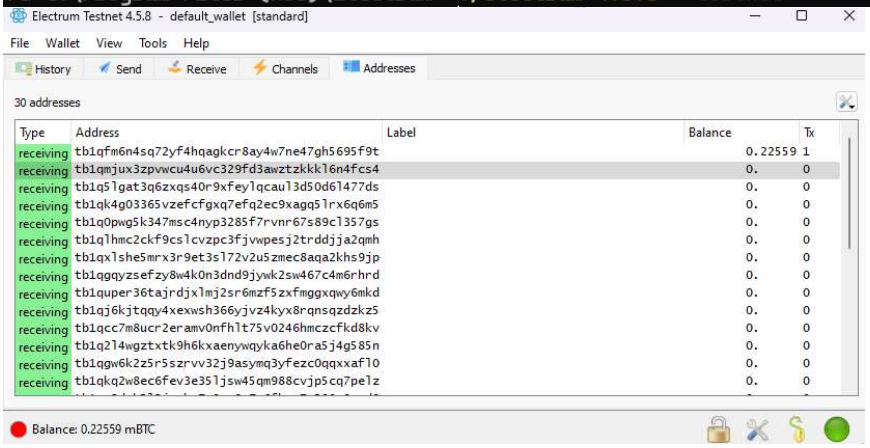
6. Verify Transaction on Blockchain Explorer

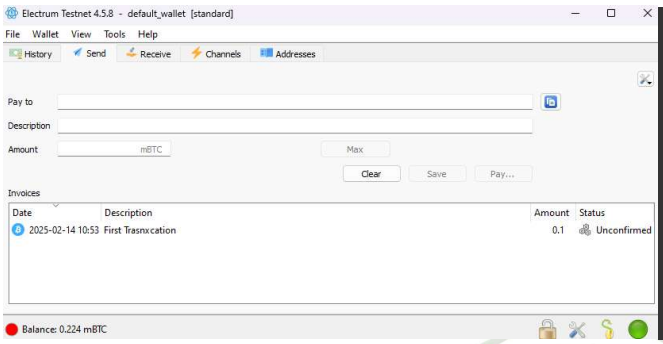
- Copy the **transaction ID (TXID)** from Electrum.
- Visit a **testnet blockchain explorer** like https://blockstream.info/testnet/ and search for the TXID.
- View transaction details like sender, receiver, fees, and confirmations.

This process helps understand **Bitcoin transactions, network confirmations, and blockchain transparency** in a risk-free environment.

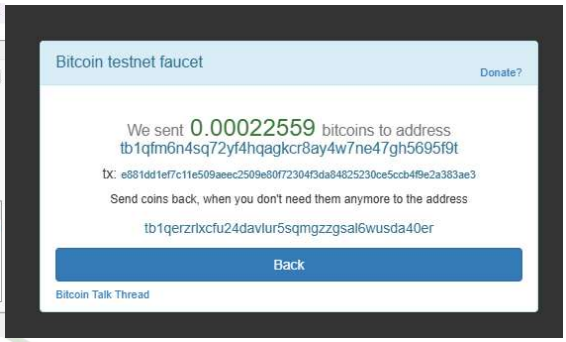
EXECUTION(Screenshots):



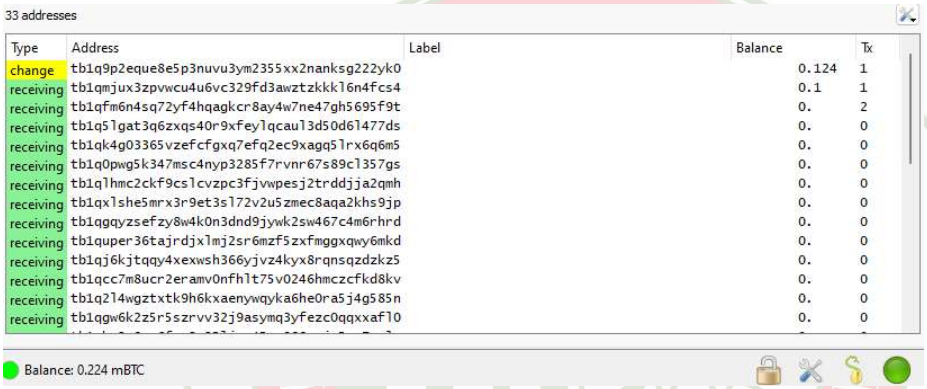




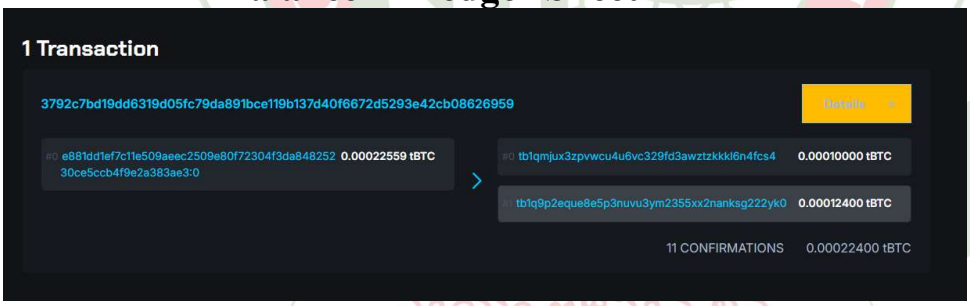
First Transaction



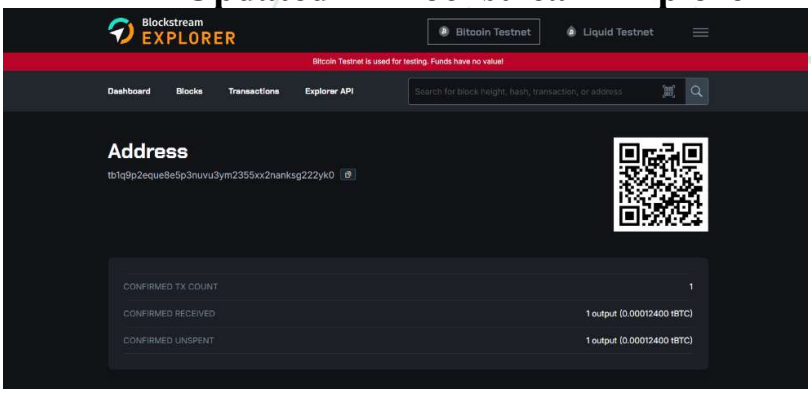
Amount Transfer (Faucet)



Balance in Ledger Sheet



Updated in Blockstream Explorer



Blockstream Explorer (Verification)

OUTPUT ANALYSIS:

1. Wallet Setup: Electrum successfully generates a testnet Bitcoin wallet with a unique public address and a 12-word seed phrase for recovery.

2. Receiving Testnet BTC: The wallet balance updates after receiving funds from a testnet faucet, with transaction details visible in the history tab.

3. Sending Bitcoin: After initiating a transaction, the wallet shows a pending status, updating to confirmed once included in a testnet block.

4. Transaction Verification: The blockchain explorer displays details such as transaction ID (TXID), sender, recipient, fee, and confirmations, ensuring transparency.

5. Conclusion: This demonstrates secure Bitcoin transactions, fee adjustments, and blockchain verification, essential for real-world cryptocurrency usage.

LAB TASK- 3

AIM: Setup metamask wallet in a web browser and create wallet and user accounts. Demonstrate sending and receiving ethers on a testnet (Sepolia). Use Block explorers like etherscan to observe the transaction details.

DESCRIPTION:

This guide explains how to set up a **MetaMask wallet** in a web browser, create a wallet and user accounts, and demonstrate **sending and receiving Ether (ETH) on the Sepolia testnet**. MetaMask is a widely used browser extension for managing Ethereum-based assets and interacting with decentralized applications (dApps). The **Sepolia testnet** is a simulated Ethereum network for developers and testers, allowing transactions without real funds. Users can obtain free ETH from **testnet faucets**, send ETH to other accounts, and verify transactions on **block explorers like Etherscan**. By following this process, users will understand **Ethereum wallet management, address generation, gas fees, transaction confirmations, and blockchain transparency** in a risk-free environment before handling real assets.

PROCEDURE:

1. Install MetaMask Extension

- Open **Google Chrome, Firefox, Brave, or Edge.**
- Visit https://metamask.io/ and click **Download**.
- Install the extension and pin it to your browser for easy access.

2. Create a New Wallet

- Click **"Get Started"** and select **"Create a Wallet".**
- Set a **strong password** and confirm.
- Save the **12-word Secret Recovery Phrase** securely (do not share it).
- Confirm the recovery phrase to complete the wallet setup.

3. Configure Sepolia Testnet

- Open MetaMask and click on the **network dropdown** at the top.
- Select **"Show/Hide test networks"** in settings and enable it.
- Choose **Sepolia Testnet** from the list.

4. Create Additional User Accounts

- Click the **account icon** and select **"Create Account".**
- Name the new account and repeat for multiple accounts if needed.

5. Receive Sepolia Testnet ETH

- Copy your **Sepolia wallet address** from MetaMask.
- Visit a **Sepolia testnet faucet** (e.g., https://sepoliafaucet.com).
- Paste your wallet address and request free ETH.
- Wait for the transaction confirmation.

6. Send ETH on Sepolia Testnet

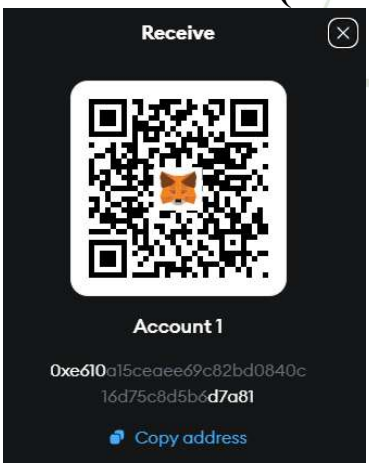
- Open MetaMask and click **Send**.
- Enter the **recipient's Sepolia** testnet address.
- Specify the amount of ETH to send.
- Adjust the **gas fee** (higher fee = faster confirmation).
- Click **Confirm** to broadcast the transaction.

7. Verify Transactions on Etherscan

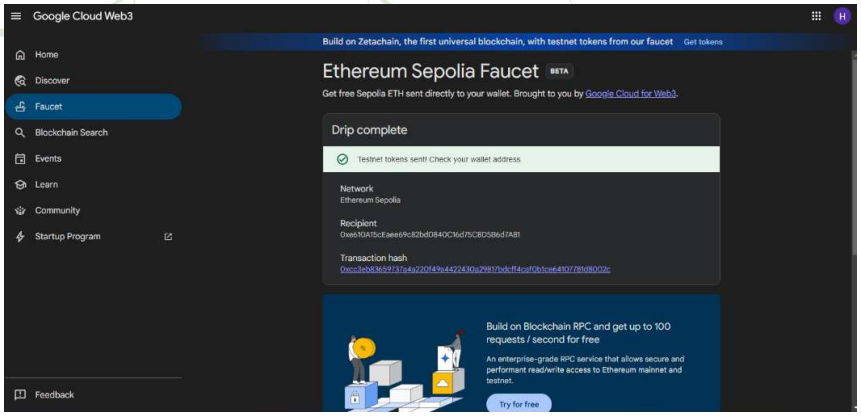
- Copy the **Transaction Hash (TXID)** from MetaMask.
- Visit https://sepolia.etherscan.io and paste the TXID.
- View transaction details, including sender, receiver, gas fees, and confirmations.

This process demonstrates **Ethereum wallet operations, gas fees, transaction speeds, and blockchain verification** in a risk-free testnet environment.

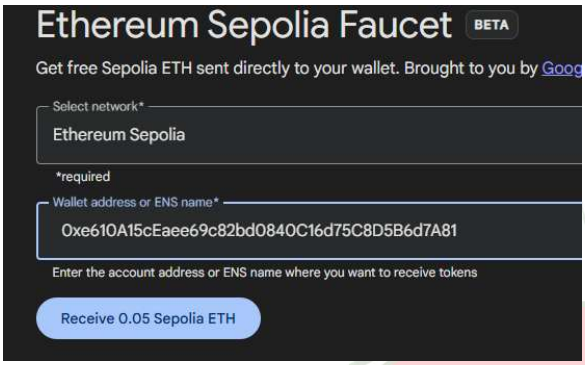
EXECUTION(Screenshots):



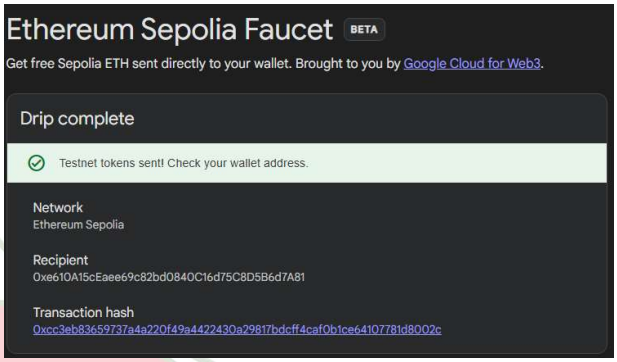
Metamask



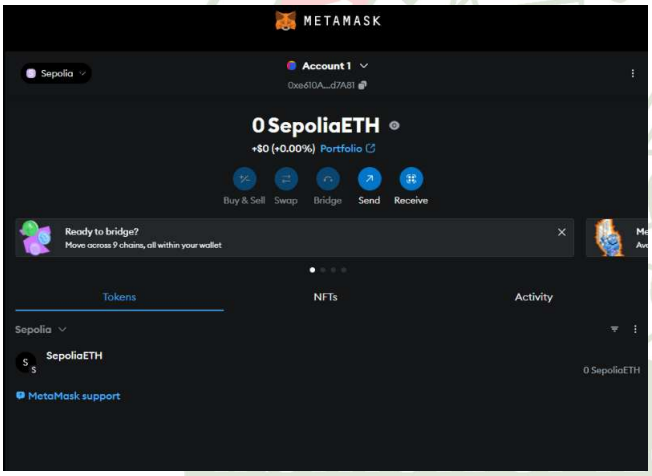
Ethereum Sepolia Faucet



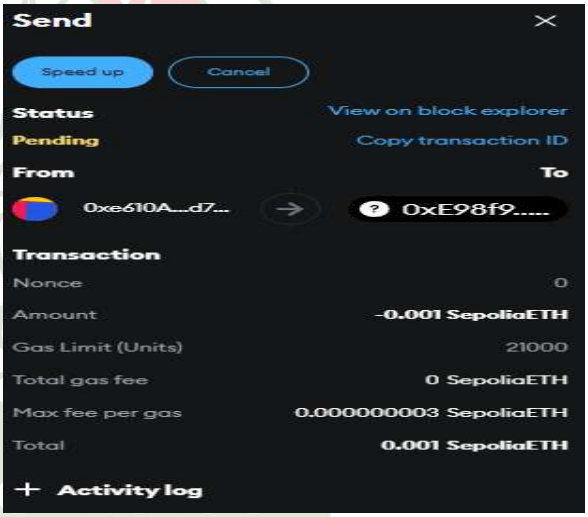
Faucet Address Transfer



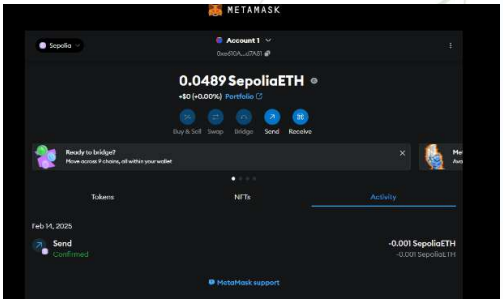
Amount Transfer (Faucet)



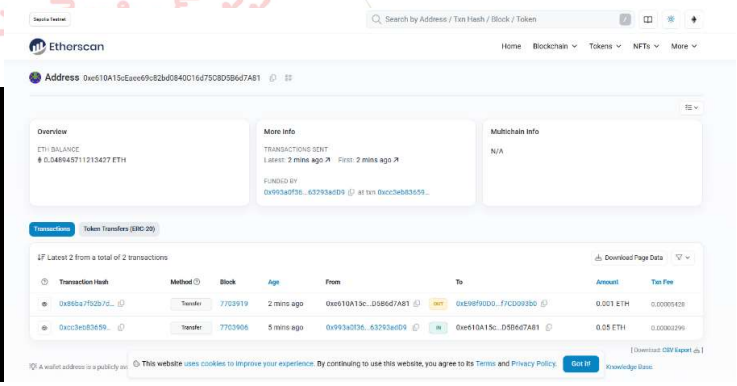
Sepolia Account



Transaction



Updated Sepolia Account



Etherscan Explorer (Verification)

OUTPUT ANALYSIS:

1. Wallet Creation: MetaMask successfully generates a **new Ethereum wallet** with a unique address and a **Secret Recovery Phrase** for backup.
2. Testnet Configuration: The wallet is connected to the **Sepolia testnet**, enabling test transactions.
3. Receiving ETH: After requesting from a faucet, the **wallet balance updates**, confirming receipt of test ETH.
4. Sending ETH: Upon sending ETH, the transaction appears as **"Pending"**, then updates to **"Confirmed"** once included in a block.
5. Transaction Verification: The **block explorer (Etherscan)** displays details like **transaction hash, sender, receiver, gas fees, and block confirmations**, ensuring transparency.
6. Conclusion: This demonstrates **Ethereum transactions, gas fees, and blockchain verification**, essential for learning before using real ETH.

LAB TASK- 4

AIM: Launch Remix web browser and write a smart contract using the solidity language for the “Hello World program”.

DESCRIPTION:

This guide explains how to **write, compile, and deploy a simple "Hello World" smart contract** using **Solidity** in **Remix IDE**, an online development environment for Ethereum smart contracts. Solidity is the primary programming language for Ethereum, enabling the creation of **dApps and smart contracts**. The **Hello World smart contract** contains a function that returns the string **"Hello, World!"** when called. This program helps understand **basic Solidity syntax, smart contract structure, function visibility, and deployment processes**. Users will also learn how to interact with a deployed contract in Remix. First, the contract is written in Solidity, defining a **public function** that returns the message. It is then compiled using the **Solidity Compiler** in Remix. After successful compilation, the contract is **deployed on a local EVM** using Remix's built-in **JavaScript VM**. Once deployed, users can call the contract’s function through Remix’s interface, which displays **"Hello, World!"** as output. This process provides an **introductory hands-on experience in smart contract development**, helping users grasp fundamental **Ethereum blockchain concepts, Solidity programming, and contract execution** before working on more advanced decentralized applications.

PROCEDURE:

Step 1: Open Remix IDE

- Go to **https://remix.ethereum.org** in your web browser.
- Select **Solidity** as the development environment.

Step 2: Create a New Solidity File

- Click on the **File Explorer (left panel)** and create a new file named **HelloWorld.sol.**

Step 3: Write the Solidity Smart Contract

- Open the newly created file and write the following Solidity code:

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

contract HelloWorld {
    function getMessage() public pure returns (string memory) {
        return "Hello, World!";
    }
}

```

Step 4: Compile the Smart Contract

- Go to the **Solidity Compiler** tab on the left panel.
- Select **Compiler Version 0.8.x** (Ensure it matches pragma solidity ^0.8.0;).
- Click **Compile HelloWorld.sol** (Check for errors in the console).

Step 5: Deploy the Smart Contract

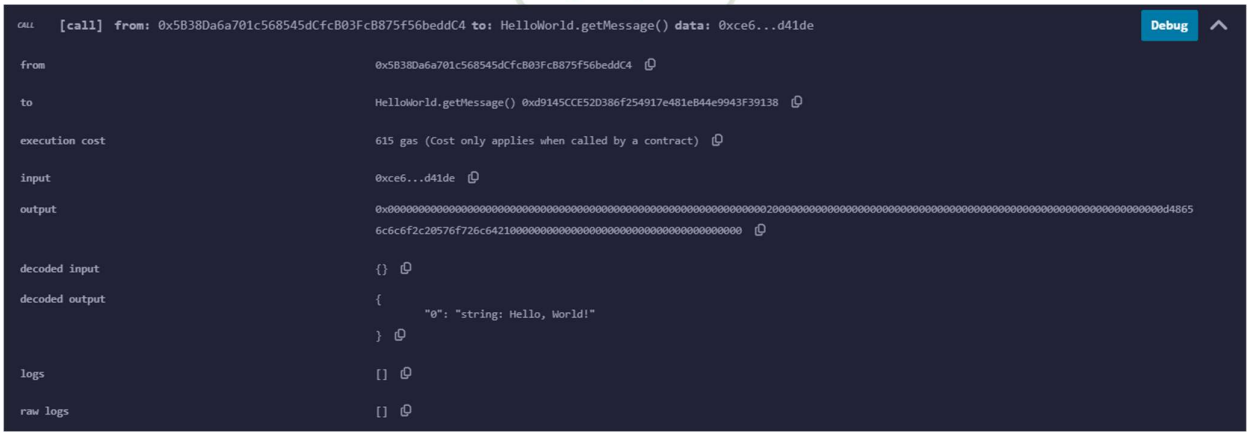
- Navigate to the **Deploy & Run Transactions** tab.
- Under **Environment**, select **"JavaScript VM (London)"** for local deployment.
- Click **Deploy** and check the **Deployed Contracts** section.

Step 6: Interact with the Smart Contract

- Click on the **getMessage()** button in the deployed contract.
- The output will display **"Hello, World!",** confirming successful execution.

This process provides a basic understanding of Solidity, contract compilation, and deployment using Remix IDE.

OUTPUT:



OUTPUT ANALYSIS:

1. Compilation Output:
   - If the contract compiles successfully, no errors appear in the Remix compiler console.
   - Warnings, if any, will indicate best practices or optimizations.
2. Deployment Output:
   - After clicking Deploy, the Deployed Contracts section lists the deployed instance with its contract address.
   - The JavaScript VM (London) provides a simulated blockchain environment, meaning the deployment does not cost real ETH.
3. Function Execution Output:
   - Clicking getMessage() calls the function and returns the expected output: "Hello, World!".
   - The function call is executed instantly because it is a pure function that does not modify the blockchain state.
4. Transaction Details:
   - If deployed on a testnet, the transaction details (gas fee, block confirmation) can be verified on Etherscan.
   - The transaction hash and gas usage appear in the Remix console when executing blockchain-modifying functions.
5. Conclusion:
   - The contract runs as expected, demonstrating basic Solidity syntax, function execution, and Remix IDE usage.
   - This foundational example helps understand smart contract interaction before deploying real-world applications.

LAB TASK- 5(a)

AIM: Write Solidity program for incrementing/decrementing a counter variable in a smart contract.

DESCRIPTION:

This guide explains how to write, compile, and deploy a simple counter smart contract using Solidity in Remix IDE, an online development environment for Ethereum smart contracts. Solidity is the primary language for Ethereum, allowing developers to build dApps and smart contracts. This program helps understand Solidity syntax, smart contract structure, function visibility, and deployment procedures. Users will also learn how to interact with a deployed contract in Remix. First, the contract is written in Solidity, defining two public functions one to increase the counter and another to decrease it while ensuring the value never becomes negative. It is then compiled using the Solidity Compiler in Remix. After successful compilation, the contract is deployed on a local EVM using Remix's built-in JavaScript VM. Once deployed, users can interact with the contract by calling the increment and decrement functions, modifying the counter's value. The getCount function allows users to retrieve the current counter value. This process provides an introductory hands-on experience in Solidity development, helping users understand Ethereum blockchain concepts, smart contract execution, and state-changing operations before building more complex dApps.

PROCEDURE:

Step 1: Open Remix IDE

- Go to **https://remix.ethereum.org** in your web browser.
- Select **Solidity** as the development environment.

Step 2: Create a New Solidity File

- Click on the **File Explorer (left panel)** and create a new file named **counter.sol.**

Step 3: Write the Solidity Smart Contract

- Open the newly created file and write the following Solidity code:

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

contract Counter {
    uint private count;

    function increment() public { count += 1; }    infinite gas
    function decrement() public { require(count > 0, "Counter cannot be negative"); count -= 1; }
    function getCount() public view returns (uint) { return count; }    2432 gas
}

```

Step 4: Compile the Smart Contract

- Go to the **Solidity Compiler** tab on the left panel.
- Select **Compiler Version 0.8.x** and Click **Compile counter.sol**

Step 5: Deploy the Smart Contract

- Navigate to the **Deploy & Run Transactions** tab.
- Under **Environment**, select **"JavaScript VM (London)"** for local deployment. Click **Deploy** and check the **Deployed Contracts** section.

Step 6: Interact with the Smart Contract

- Click **increment()** or **decrement()** to modify the counter.
- Click **getCount()** to retrieve the current value.

This process provides a basic understanding of Solidity, contract compilation, and deployment using Remix IDE.

OUTPUT:

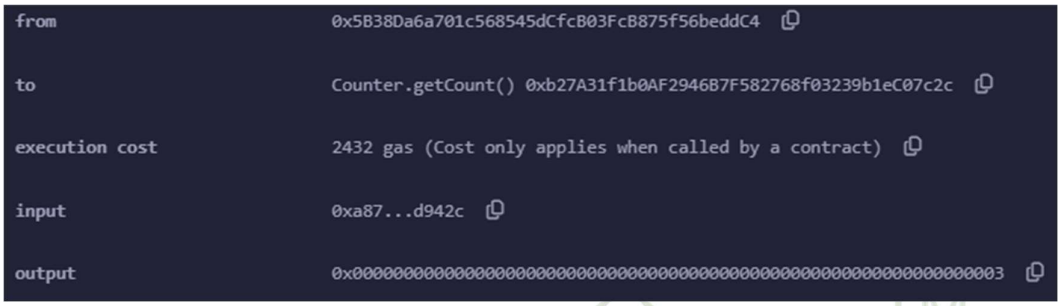

```
CALL  [call] from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to: Counter.getCount() data: 0xa87...d942c

from                    0x5B38Da6a701c568545dCfcB03FcB875f56beddC4
to                      Counter.getCount() 0xb27A31f1b0AF2946B7F582768f03239b1eC07c2c
execution cost          2432 gas (Cost only applies when called by a contract)
input                   0xa87...d942c
output                  0x0000000000000000000000000000000000000000000000000000000000000002
```

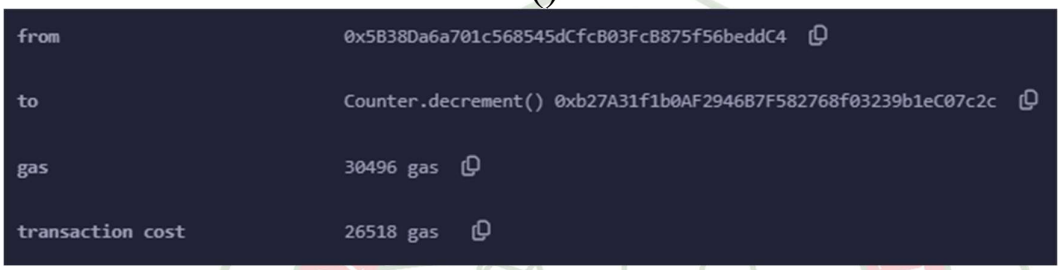
Getcount()

```
block hash              0xbfbbb62032807b381a3b0957aac7291b939fe98265816fb4c987eec7edf476bf
block number            19
from                    0x5B38Da6a701c568545dCfcB03FcB875f56beddC4
to                      Counter.increment() 0xb27A31f1b0AF2946B7F582768f03239b1eC07c2c
gas                     30406 gas
```

Counter.increment()



Getcount()



Counter.decrement()

OUTPUT ANALYSIS:

1. **Initial Counter Value:** After deployment, getCount() returns 0.
2. **Increment Operation:** Each call to increment() increases the counter by 1 (e.g., 0 → 1 → 2 → ...).
3. **Decrement Operation:** Each call to decrement() reduces the counter by 1, provided it is greater than 0 (e.g., 2 → 1 → 0).
4. **Underflow Prevention:** If the counter is 0, calling decrement() triggers an error: "Counter cannot be negative", ensuring it never becomes negative.

This confirms the smart contract's correct functionality, maintaining state integrity and preventing errors.

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

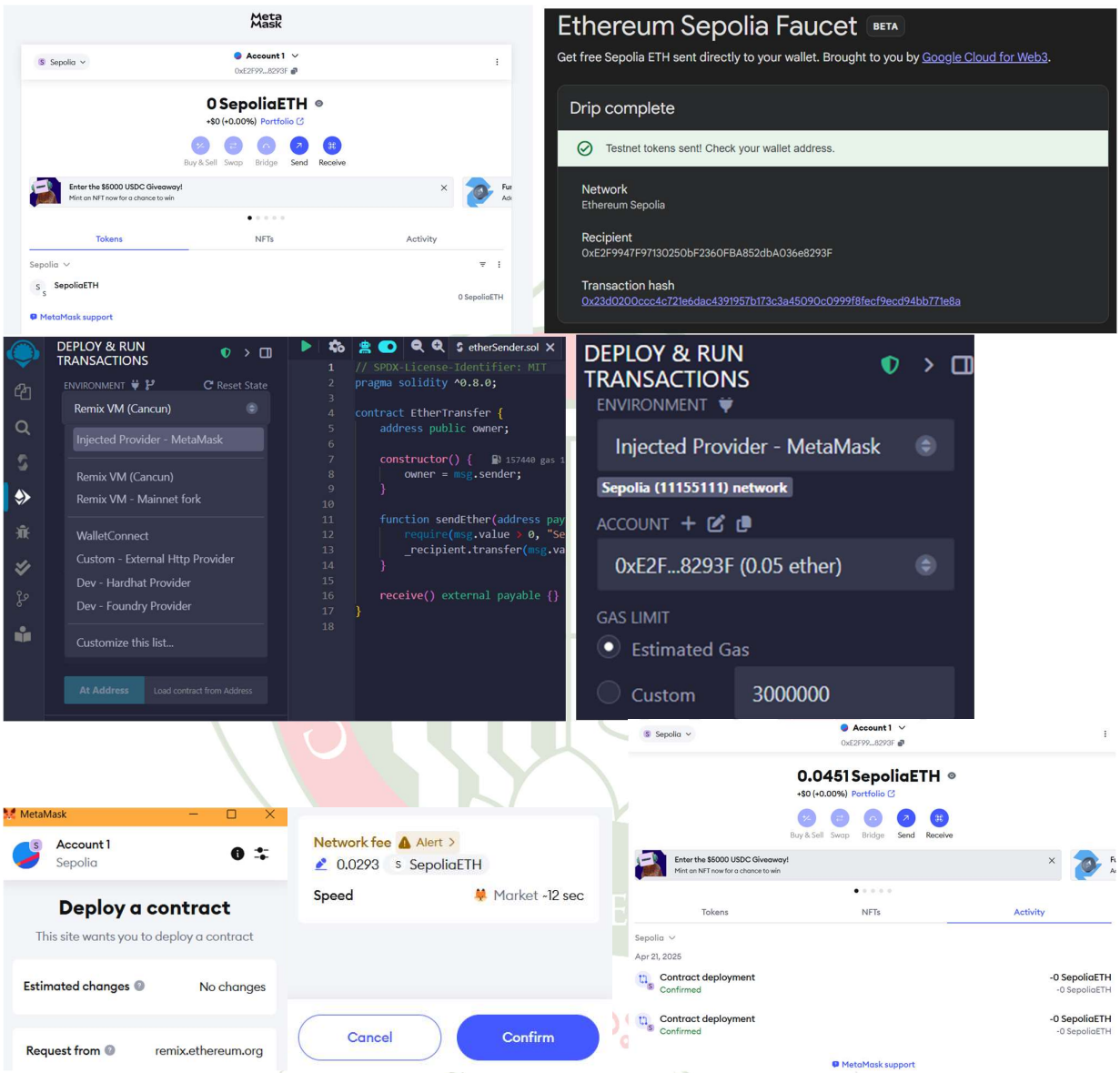
1. Go to the **Deploy & Run Transactions** tab.
2. Set **Environment** to Injected Provider - MetaMask (connects Remix to MetaMask).
3. Select your MetaMask account (Account A).
4. Click **Deploy** and approve the MetaMask transaction.

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.

DDBCA Lab Expts – 8,9,10

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:

1. **Record the origin and status of goods at each stage.**
2. **Ensure that all updates are transparent and immutable.**
3. **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. This experiment implements a blockchain-based supply chain management system using Solidity. The smart contract records and monitors the lifecycle of a product from its creation by the manufacturer to its final delivery to the consumer. Each participant in the supply chain — Manufacturer, Distributor, Retailer, and Consumer — plays a defined role in the product's journey. All interactions and state transitions are securely stored on the Ethereum blockchain, guaranteeing:

1. **Transparency**: Every action and transaction is publicly recorded and traceable.
2. **Authenticity**: Only authorized roles can perform designated actions.
3. **Immutability**: Once recorded, data cannot be altered or tampered with.

This system enhances supply chain accountability, minimizes fraud, and ensures a verified and auditable product flow.

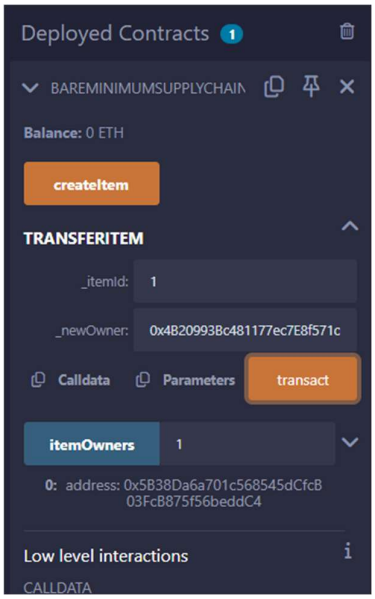
CODE:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;
contract BareMinimumSupplyChain {
    uint256 private nextItemId = 1;
    mapping(uint256 => address) public itemOwners;
    function createItem() public returns (uint256 newItemId) {    infinite gas
        newItemId = nextItemId;
        itemOwners[newItemId] = msg.sender;
        nextItemId++;
    }
    function transferItem(uint256 _itemId, address _newOwner) public {    27286 gas
        address currentOwner = itemOwners[_itemId];
        require(currentOwner != address(0), "Item does not exist");
        require(currentOwner == msg.sender, "Caller is not the owner");
        require(_newOwner != address(0), "Invalid new owner address");
        itemOwners[_itemId] = _newOwner;
    }
}
```

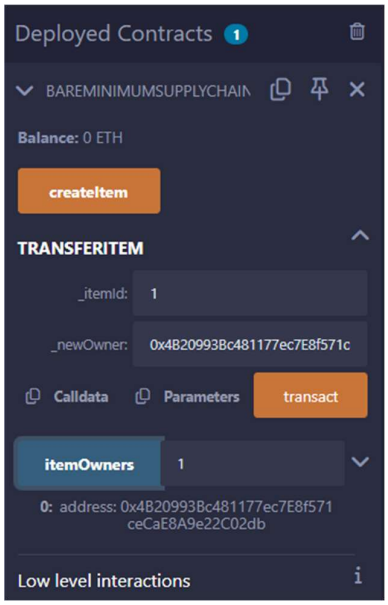
DDBCA Lab Expts – 8,9,10

OUTPUT:

Before Transact see address of 1



After Transact see address of 1



DDBCA Lab Expts – 8,9,10

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:
1. Accepts insurance funds from any user (like an insurer or the government).
2. Takes the farmer's address during deployment.
3. Allows an update to rainfall (simulating an oracle).
4. 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.

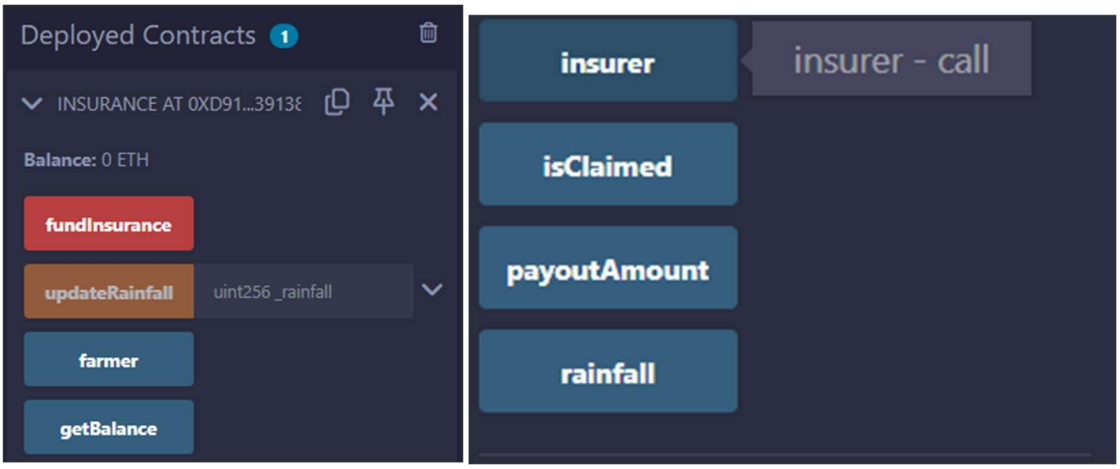
This smart contract simulates a decentralized insurance system where:
1. Users can purchase insurance by paying a fixed premium.
2. The insurer (contract deployer) can verify if an insured event has occurred.
3. Upon verification, the contract automatically pays out the claim amount to the user from the contract balance.
4. The process eliminates the need for manual claim reviews or third-party intermediaries.
5. The insurer can also fund the contract to ensure enough liquidity for payouts.

DDBCA Lab Expts – 8,9,10

CODE:

```
// 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() {    343985 gas 248800 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 {}    164 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) {    357 gas
        return address(this).balance;
    }
}
```

OUTPUT:



Explore all the functions

DDBCA Lab Expts – 8,9,10

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.

Voting is a core mechanism in any decision-making system. Traditional voting systems rely on central authorities which can be vulnerable to fraud or manipulation. By using blockchain and smart contracts, we can build a decentralized, tamper-proof voting system.

In this system:

• A list of candidates is initialized.
• Voters can vote once.
• Only registered users can vote.
• The contract stores and tallies all votes.
• Final results can be viewed publicly and securely.
• Results are automatically calculated and displayed.
• Data is transparent and stored permanently on the blockchain.

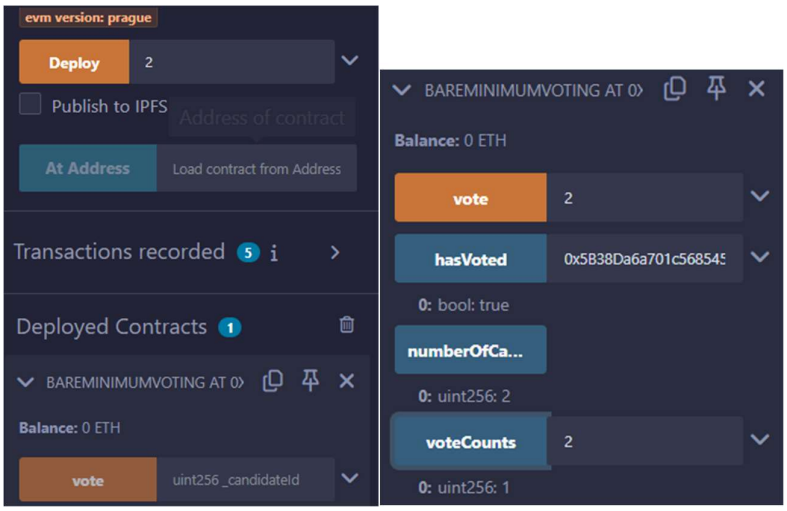
CODE:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;
contract BareMinimumVoting {
    uint public immutable numberOfCandidates;
    mapping(uint => uint) public voteCounts;
    mapping(address => bool) public hasVoted;
    constructor(uint _numberOfCandidates) {    infinite gas 232600 gas
        require(_numberOfCandidates > 0, "Voting requires at least one candidate/option");
        numberOfCandidates = _numberOfCandidates;
    }

    function vote(uint _candidateId) public {    infinite gas
        // require(!hasVoted[msg.sender], "Voter has already voted");
        // - If this added we can vote only once
        require(_candidateId > 0 && _candidateId <= numberOfCandidates, "Invalid candidate ID");
        hasVoted[msg.sender] = true;
        voteCounts[_candidateId]++;
    }
}
```

DDBCA Lab Expts – 8,9,10

OUTPUT:



Check for voting of no of candidates you give and also vote once per address