Title: Writing a Solidity Program to Reverse Digits of a Positive Integer

Objective: The objective of this lab is to design and implement a Solidity program that takes a positive integer as input and returns the integer obtained by reversing its digits.

Prerequisites:

- Basic understanding of Solidity programming language
- Familiarity with Ethereum development tools (e.g., Remix, Truffle)
- Access to a test Ethereum network (e.g., Ganache, Ropsten)

Requirements:

- Solidity compiler
- Remix or another Solidity development environment

Step-by-Step Instructions:

Step 1: Write the Solidity Program

1. Open Remix or your preferred Solidity development environment.
2. Create a new file named **`ReverseInteger.sol`**.
3. Write the Solidity code to implement the program logic. Below is an example:

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

contract ReverseInteger {
    function reverse(uint256 n) public pure returns (uint256) {
        require(n > 0, "Input must be a positive integer");

        uint256 reversed = 0;
        uint256 remainder;

        while (n != 0) {
            remainder = n % 10;
            reversed = reversed * 10 + remainder;
            n = n / 10;
        }

        return reversed;
    }
}
```

Step 2: Compile the Solidity Program

1. Compile the Solidity program in Remix or your Solidity development environment to ensure there are no syntax errors.

Step 3: Deploy the Smart Contract

1. Deploy the compiled smart contract to a test Ethereum network (e.g., Ganache, Ropsten) using Remix or another deployment tool.

Step 4: Interact with the Smart Contract

1. Use Remix or any Ethereum wallet to interact with the deployed smart contract.
2. Call the `reverse` function with a positive integer as input to test the functionality.
3. Verify that the contract returns the integer obtained by reversing the digits of the input integer.

Conclusion:

In this lab, we have designed and implemented a Solidity program to reverse the digits of a positive integer. By writing Solidity code to iterate through the digits of the input integer and construct the reversed integer, we have created a useful function for manipulating integer values on the Ethereum blockchain. This lab provides hands-on experience with Solidity programming and demonstrates the versatility of smart contracts in performing various tasks.

Title: Writing a Solidity Program to Sum All Prime Numbers from 1 to 300

Objective: The objective of this lab is to design and implement a Solidity program that calculates the sum of all prime numbers from 1 to 300.

Prerequisites:

- Basic understanding of Solidity programming language
- Familiarity with Ethereum development tools (e.g., Remix, Truffle)
- Access to a test Ethereum network (e.g., Ganache, Ropsten)

Requirements:

- Solidity compiler
- Remix or another Solidity development environment

Step-by-Step Instructions:

Step 1: Write the Solidity Program

1. Open Remix or your preferred Solidity development environment.
2. Create a new file named **`PrimeNumberSum.sol`**.
3. Write the Solidity code to implement the program logic. Below is an example:

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

contract PrimeNumberSum {
    function isPrime(uint256 number) internal pure returns (bool) {
        if (number <= 1) {
            return false;
        }
        for (uint256 i = 2; i <= number / 2; i++) {
            if (number % i == 0) {
                return false;
            }
        }
        return true;
    }

    function sumPrimes() public pure returns (uint256) {
        uint256 sum = 0;
        for (uint256 i = 2; i <= 300; i++) {
            if (isPrime(i)) {
                sum += i;
```

```
            }
        }
        return sum;
    }
}
```

Step 2: Compile the Solidity Program

1. Compile the Solidity program in Remix or your Solidity development environment to ensure there are no syntax errors.

Step 3: Deploy the Smart Contract

1. Deploy the compiled smart contract to a test Ethereum network (e.g., Ganache, Ropsten) using Remix or another deployment tool.

Step 4: Interact with the Smart Contract

1. Use Remix or any Ethereum wallet to interact with the deployed smart contract.
2. Call the **`sumPrimes`** function to calculate the sum of all prime numbers from 1 to 300.
3. Verify that the contract returns the correct sum.

Conclusion:

In this lab, we have designed and implemented a Solidity program to sum all prime numbers from 1 to 300. By writing Solidity code to iterate through the numbers and check for primality using the **`isPrime`** function, we have created a useful function for performing mathematical computations on the Ethereum blockchain. This lab provides hands-on experience with Solidity programming and demonstrates the versatility of smart contracts in performing various tasks.

Title: Designing a Smart Contract for Storing and Displaying Information of Students

Objective: The objective of this lab is to design and implement a Solidity smart contract that allows for the storage and display of information of students, including their name, roll number, percentage, email ID, date of birth, class, and admission year.

Prerequisites:

- Basic understanding of Solidity programming language
- Familiarity with Ethereum development tools (e.g., Remix, Truffle)
- Access to a test Ethereum network (e.g., Ganache, Ropsten)

Requirements:

- Solidity compiler
- Remix or another Solidity development environment

Step-by-Step Instructions:

Step 1: Write the Solidity Smart Contract

1. Open Remix or your preferred Solidity development environment.
2. Create a new file named **`StudentInformation.sol`**.
3. Write the Solidity code to implement the program logic. Below is an example:

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

contract StudentInformation {
    struct Student {
        string name;
        uint256 rollNo;
        uint256 percentage;
        string email;
        uint256 dob; // Date of Birth as Unix timestamp
        string studentClass;
        uint256 admissionYear;
    }

    mapping(address => Student) public students;

    function storeStudentData(
        string memory _name,
        uint256 _rollNo,
```

```
        uint256 _percentage,
        string memory _email,
        uint256 _dob,
        string memory _studentClass,
        uint256 _admissionYear
    ) public {
        students[msg.sender] = Student({
            name: _name,
            rollNo: _rollNo,
            percentage: _percentage,
            email: _email,
            dob: _dob,
            studentClass: _studentClass,
            admissionYear: _admissionYear
        });
    }

    function displayStudentData() public view returns (
        string memory,
        uint256,
        uint256,
        string memory,
        uint256,
        string memory,
        uint256
    ) {
        Student memory student = students[msg.sender];
        return (
            student.name,
            student.rollNo,
            student.percentage,
            student.email,
            student.dob,
            student.studentClass,
            student.admissionYear
        );
    }
}
```

Step 2: Compile the Solidity Smart Contract

1. Compile the Solidity contract in Remix or your Solidity development environment to ensure there are no syntax errors.

Step 3: Deploy the Smart Contract

1. Deploy the compiled smart contract to a test Ethereum network (e.g., Ganache, Ropsten) using Remix or another deployment tool.

Step 4: Interact with the Smart Contract

1. Use Remix or any Ethereum wallet to interact with the deployed smart contract.
2. Call the `storeStudentData` function to store the information of a student.
3. Call the `displayStudentData` function to view the stored information of the student.

Conclusion:

In this lab, we have designed and implemented a Solidity smart contract for storing and displaying information of students. By defining a `Student` struct and using a mapping to store the information of each student,we have created a decentralized solution for managing student data on the Ethereum blockchain. This lab provides hands-on experience with Solidity programming and demonstrates the power of smart contracts in handling data storage and retrieval tasks.

Title: Designing a Smart Contract for Bank Account Management

Objective: The objective of this lab is to design and implement a Solidity smart contract to represent a bank account, including state variables for depositor information and account details, as well as methods to deposit, withdraw, and display account information.

Prerequisites:

- Basic understanding of Solidity programming language
- Familiarity with Ethereum development tools (e.g., Remix, Truffle)
- Access to a test Ethereum network (e.g., Ganache, Ropsten)

Requirements:

- Solidity compiler
- Remix or another Solidity development environment

Step-by-Step Instructions:

Step 1: Write the Smart Contract

1. Open Remix or your preferred Solidity development environment.
2. Create a new file named **`BankAccount.sol`**.
3. Write the Solidity code for the bank account smart contract. Include state variables for depositor name, account number, account type, and balance, as well as methods to deposit, withdraw, and display account information. Here's an example:

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

contract BankAccount {
    string public depositorName;
    uint256 public accountNumber;
    string public accountType;
    uint256 public balance;

    constructor(string memory _depositorName, uint256 _accountNumber, string memory
_accountType) {
        depositorName = _depositorName;
        accountNumber = _accountNumber;
        accountType = _accountType;
        balance = 0;
```

```
    }

    function deposit(uint256 amount) public {
        require(amount > 0, "Deposit amount must be greater than 0");
        balance += amount;
    }

    function withdraw(uint256 amount) public {
        require(amount > 0, "Withdrawal amount must be greater than 0");
        require(amount <= balance, "Insufficient balance");
        balance -= amount;
    }

    function displayBalance() public view returns (uint256) {
        return balance;
    }

    function displayNameAndBalance() public view returns (string memory, uint256) {
        return (depositorName, balance);
    }
}
```

Step 2: Compile the Smart Contract

1. Compile the smart contract in Remix or your Solidity development environment to ensure there are no syntax errors.

Step 3: Deploy the Smart Contract

1. Deploy the compiled smart contract to a test Ethereum network (e.g., Ganache, Ropsten) using Remix or another deployment tool.

Step 4: Interact with the Smart Contract

1. Use Remix or any Ethereum wallet to interact with the deployed smart contract.
2. Test the functionality of the contract by depositing funds, withdrawing funds, and displaying account information.

Conclusion:

In this lab, we have designed and implemented a Solidity smart contract to represent a bank account. By defining state variables for depositor information and account details, as well as methods to deposit, withdraw, and display

account information,we have created a decentralized solution for bank account management on the Ethereum blockchain. This lab provides a hands-on introduction to Solidity programming and Ethereum smart contract development, demonstrating the power of blockchain technology in financial applications.

Title: Designing a Web3.js Application for Student Information Management

Objective: The objective of this lab is to design and implement a decentralized application (DApp) using web3.js to manage student information, including reading and storing data, as well as displaying the stored information.

Prerequisites:

- Basic understanding of JavaScript, HTML, and Ethereum smart contracts
- Familiarity with the use of MetaMask or another Ethereum wallet extension
- Access to a test Ethereum network (e.g., Ganache, Ropsten)
- Solidity compiler (e.g., Remix, Truffle)

Requirements:

- Node.js installed on your computer
- Text editor (e.g., Visual Studio Code)
- Basic knowledge of Ethereum development tools (e.g., Truffle, Remix)

Step-by-Step Instructions:

Step 1: Write the Smart Contract

1. Open your preferred text editor.
2. Write the Solidity code for the student information smart contract. The contract should define state variables for student attributes and include methods for storing and reading student data. Here's an example:

```
// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;



contract StudentInformation {

    struct Student {

        string name;

        uint256 rollNo;
```

```
    uint256 percentage;
    string email;
    uint256 dob; // Date of Birth as Unix timestamp
    string studentClass;
    uint256 admissionYear;
}

mapping(address => Student) public students;

function storeStudentData(
    string memory _name,
    uint256 _rollNo,
    uint256 _percentage,
    string memory _email,
    uint256 _dob,
    string memory _studentClass,
    uint256 _admissionYear
) public {
    students[msg.sender] = Student({
        name: _name,
        rollNo: _rollNo,
        percentage: _percentage,
        email: _email,
```

```
        dob: _dob,

        studentClass: _studentClass,

        admissionYear: _admissionYear

    });

  }
```

}**Step 2: Compile and Deploy the Smart Contract**

1. Compile the smart contract using Remix or any other Solidity compiler.
2. Deploy the compiled smart contract to an Ethereum network. Note down the contract address and ABI (Application Binary Interface).

Step 3: Create the HTML Interface

1. Create a new HTML file (e.g., **`index.html`**) in your project directory.
2. Write the HTML code to create a user interface for interacting with the smart contract. Include input fields to enter student information and buttons to store and display data. Here's an example:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Student Information DApp</title>
  <script src="https://cdn.jsdelivr.net/npm/web3@1.5.3/dist/web3.min.js"></script>
  <script>
    window.addEventListener('load', async () => {
      if (window.ethereum) {
        window.web3 = new Web3(window.ethereum);
        await window.ethereum.enable();
      } else {
        console.log('Web3 not found');
      }
    });

    async function storeStudentData() {
      const accounts = await web3.eth.getAccounts();
      const contractAddress = 'CONTRACT_ADDRESS'; // Replace with the address of
the deployed contract
      const contract = new web3.eth.Contract(ABI, contractAddress);
      const name = document.getElementById('name').value;
      const rollNo = document.getElementById('rollNo').value;
```

```
        const percentage = document.getElementById('percentage').value;
        const email = document.getElementById('email').value;
        const dob = Date.parse(document.getElementById('dob').value);
        const studentClass = document.getElementById('class').value;
        const admissionYear = document.getElementById('admissionYear').value;
        try {
            await contract.methods.storeStudentData(name, rollNo, percentage, email, dob,
studentClass, admissionYear).send({
                from: accounts[0]
            });
            alert('Student data stored successfully!');
        } catch (error) {
            console.error('Error storing student data:', error);
        }
    }

    async function displayStudentData() {
        const accounts = await web3.eth.getAccounts();
        const contractAddress = 'CONTRACT_ADDRESS'; // Replace with the address of
the deployed contract
        const contract = new web3.eth.Contract(ABI, contractAddress);
        try {
            const student = await contract.methods.students(accounts[0]).call();
            alert('Student Information:\nName: ' + student.name + '\nRoll No: ' +
student.rollNo + '\nPercentage: ' + student.percentage + '\nEmail: ' + student.email +
'\nDate of Birth: ' + new Date(student.dob * 1000).toDateString() + '\nClass: ' +
student.studentClass + '\nAdmission Year: ' + student.admissionYear);
        } catch (error) {
            console.error('Error displaying student data:', error);
        }
    }
  </script>
</head>
<body>
  <h1>Student Information DApp</h1>
  <label for="name">Name:</label>
  <input type="text" id="name"><br>
  <label for="rollNo">Roll No:</label>
  <input type="number" id="rollNo"><br>
  <label for="percentage">Percentage:</label>
  <input type="number" id="percentage"><br>
  <label for="email">Email:</label>
  <input type="email" id="email"><br>
  <label for="dob">Date of Birth:</label>
  <input type="date" id="dob"><br>
  <label for="class">Class:</label>
  <input type="text" id="class"><br>
  <label for="admissionYear">Admission Year:</label>
```

```
    <input type="number" id="admissionYear"><br>
    <button onclick="storeStudentData()">Store Student Data</button>
    <button onclick="displayStudentData()">Display Student Data</button>
</body>
</html>
```

Replace **`CONTRACT_ADDRESS`** and **`ABI`** with the address and ABI of your deployed smart contract respectively.

Step 4: Testing

1. Open the HTML file in a web browser with MetaMask or another Ethereum wallet extension installed.
2. Use the input fields to enter student information and click the "Store Student Data" button to store the data on the blockchain.
3. Click the "Display Student Data" button to retrieve and display the stored student information.

Conclusion:

In this lab, we have learned how to design and implement a web3.js application for managing student information on the Ethereum blockchain. By creating a user interface with HTML and implementing JavaScript functions to interact with the deployed smart contract, we have built a decentralized application that allows users to store and retrieve student data. This lab provides a hands-on introduction to Ethereum development and DApp creation, demonstrating the power and versatility of blockchain technology in various applications.

Title: Designing a Web Application for Bank Account Management

Objective: To design and implement a web application using Node.js and Express framework for managing bank accounts, including depositing and withdrawing funds, as well as displaying account details.

Prerequisites:

- Basic understanding of JavaScript and Node.js
- Familiarity with RESTful API concepts
- Knowledge of HTTP methods (GET, POST)

Requirements:

- Node.js installed on your computer
- Text editor (e.g., Visual Studio Code)
- Postman or similar tool for testing API endpoints

Procedure:

Step 1: Set Up the Project

1. Open your terminal or command prompt.
2. Create a new directory for your project.
   ```
   mkdir bank-account-management
   cd bank-account-management
   ```
3. Initialize a new Node.js project.
   ```
   npm init -y
   ```
4. Install the necessary dependencies: Express and Body-parser.
   ```
   npm install express body-parser
   ```

Step 2: Create the Web Application

1. Create a new file named **`app.js`**.
2. Open **`app.js`** in your text editor.
3. Start by requiring the necessary modules.
4. `const express = require('express');`
5. `const bodyParser = require('body-parser');`
6. `const app = express();`
7. Set up middleware for parsing JSON requests.
   `app.use(bodyParser.json());`
8. Define the state variables and methods for the bank account.
   `let accounts = [];`

```
class BankAccount {
  constructor(name, accountNumber, accountType, balance) {
    this.name = name;
    this.accountNumber = accountNumber;
    this.accountType = accountType;
    this.balance = balance;
  }

  deposit(amount) {
    this.balance += amount;
    return this.balance;
  }

  withdraw(amount) {
    if (amount > this.balance) {
      return "Insufficient balance";
    }
    this.balance -= amount;
    return this.balance;
  }

  display() {
    return `Name: ${this.name}, Balance: ${this.balance}`;
  }
}
```

9. Implement the API endpoints for creating an account, depositing funds, withdrawing funds, and displaying account details.

```
// API endpoints

// Create an account
app.post('/account', (req, res) => {
  const { name, accountNumber, accountType, balance } = req.body;
  const newAccount = new BankAccount(name, accountNumber, accountType, balance);
  accounts.push(newAccount);
  res.json({ message: 'Account created successfully' });
});

// Deposit to an account
app.post('/account/:accountNumber/deposit', (req, res) => {
  const accountNumber = req.params.accountNumber;
  const { amount } = req.body;

  const account = accounts.find(acc => acc.accountNumber === accountNumber);
  if (!account) {
    return res.status(404).json({ message: 'Account not found' });
  }
```

```
  account.deposit(amount);
  res.json({ message: 'Amount deposited successfully', balance: account.balance });
});

// Withdraw from an account
app.post('/account/:accountNumber/withdraw', (req, res) => {
  const accountNumber = req.params.accountNumber;
  const { amount } = req.body;

  const account = accounts.find(acc => acc.accountNumber === accountNumber);
  if (!account) {
    return res.status(404).json({ message: 'Account not found' });
  }

  const remainingBalance = account.withdraw(amount);
  if (typeof remainingBalance === 'string') {
    return res.status(400).json({ message: remainingBalance });
  }

  res.json({ message: 'Amount withdrawn successfully', balance: remainingBalance });
});

// Display account details
app.get('/account/:accountNumber', (req, res) => {
  const accountNumber = req.params.accountNumber;
  const account = accounts.find(acc => acc.accountNumber === accountNumber);
  if (!account) {
    return res.status(404).json({ message: 'Account not found' });
  }

  res.json({ name: account.name, balance: account.balance });
});
```

Step 3: Test the Application

1. Save **`app.js`**.
2. Open your terminal and navigate to the project directory.
3. Start the server by running:
   node app.js
4. Open Postman or a similar tool to test the API endpoints.
5. Send requests to create accounts, deposit and withdraw funds, and display account details.

Conclusion: In this lab, we have successfully designed and implemented a web application for managing bank accounts using Node.js and Express. We learned how to define state variables and methods for a bank account class, create API

endpoints for various account operations, and test the application using Postman. This project provides a foundation for building more complex banking systems or similar applications in the future.

Title: Designing a web3.js Application for a Lottery DApp

Objective: The objective of this lab is to design and implement a decentralized application (DApp) for a lottery using web3.js to interact with a smart contract deployed on the Ethereum blockchain.

Prerequisites:

- Basic understanding of JavaScript and HTML
- Familiarity with Ethereum smart contracts and Solidity programming
- MetaMask or another Ethereum wallet extension installed in the browser
- Access to a test Ethereum network (e.g., Ganache, Ropsten)

Requirements:

- Node.js installed on your computer
- Text editor (e.g., Visual Studio Code)
- Basic knowledge of Ethereum development tools (e.g., Truffle, Remix)

Step-by-Step Instructions:

Step 1: Write the Smart Contract

1. Open your preferred text editor.
2. Write the Solidity code for the lottery smart contract. You can use the example provided in the previous response.

Step 2: Compile and Deploy the Smart Contract

1. Compile the smart contract using tools like Remix, Truffle, or Hardhat.
2. Deploy the compiled smart contract to a test Ethereum network. Make sure to note down the contract address and ABI (Application Binary Interface).

Step 3: Create the HTML Interface

1. Create a new HTML file (e.g., **`index.html`**) in your project directory.
2. Write the HTML code to create a simple user interface for the lottery DApp. You can use the example provided in the previous response.

Step 4: Implement the JavaScript Functions

1. Open the HTML file in your text editor.

2. Write JavaScript functions to interact with the deployed smart contract using web3.js. Use the example provided in the previous response as a reference.
3. Replace **`ABI`** and **`CONTRACT_ADDRESS`** with the ABI and address of your deployed smart contract respectively.

Step 5: Test the DApp

1. Open the HTML file in a web browser with MetaMask or another Ethereum wallet extension installed.
2. Test the functionality of the DApp by entering the lottery, picking a winner, and retrieving the list of players.
3. Verify that transactions are executed successfully and that the DApp behaves as expected.

Conclusion:

In this lab, we have learned how to design and implement a web3.js application for a lottery DApp. We created a user interface using HTML, wrote JavaScript functions to interact with the Ethereum blockchain using web3.js, and tested the DApp's functionality. This lab provides a hands-on introduction to Ethereum development and DApp creation, laying the foundation for building more complex decentralized applications in the future.

Title: Designing a Web3.js Application for Property Transfer

Objective: The objective of this lab is to design and implement a decentralized application (DApp) using web3.js to facilitate property transfer by interacting with a smart contract deployed on the Ethereum blockchain.

Prerequisites:

- Basic understanding of JavaScript and HTML
- Familiarity with Ethereum smart contracts and Solidity programming
- MetaMask or another Ethereum wallet extension installed in the browser
- Access to a test Ethereum network (e.g., Ganache, Ropsten)

Requirements:

- Node.js installed on your computer
- Text editor (e.g., Visual Studio Code)
- Basic knowledge of Ethereum development tools (e.g., Truffle, Remix)

Step-by-Step Instructions:

Step 1: Write the Smart Contract

1. Open your preferred text editor.
2. Write the Solidity code for the property transfer smart contract. You can use the example provided in the previous response.

Step 2: Compile and Deploy the Smart Contract

1. Compile the smart contract using tools like Remix, Truffle, or Hardhat.
2. Deploy the compiled smart contract to a test Ethereum network. Make sure to note down the contract address and ABI (Application Binary Interface).

Step 3: Create the HTML Interface

1. Create a new HTML file (e.g., **`index.html`**) in your project directory.
2. Write the HTML code to create a simple user interface for the property transfer DApp. You can use the example provided in the previous response.

Step 4: Implement the JavaScript Functions

1. Open the HTML file in your text editor.

2. Write JavaScript functions to interact with the deployed smart contract using web3.js. Use the example provided in the previous response as a reference.
3. Replace **`ABI`** and **`CONTRACT_ADDRESS`** with the ABI and address of your deployed smart contract respectively.

Step 5: Test the DApp

1. Open the HTML file in a web browser with MetaMask or another Ethereum wallet extension installed.
2. Test the functionality of the DApp by entering the address of the new owner and clicking the "Transfer Ownership" button.
3. Verify that the transaction is executed successfully and that ownership of the property is transferred as expected.

Conclusion:

In this lab,we have learned how to design and implement a web3.js application for property transfer on the Ethereum blockchain. By creating a user interface with HTML and implementing JavaScript functions to interact with the deployed smart contract,we have built a simple DApp that allows users to transfer ownership of properties. This lab provides a hands-on introduction to Ethereum development and DApp creation, laying the foundation for building more complex decentralized applications in the future.