(a) (1 point) Which one of the following is not an application of hash functions?
Answer: B. Virus detection.
Hash functions are used in applications like one-way password files, intrusion detection, and key wrapping, but virus detection typically involves pattern matching or signature analysis, not hash functions directly.

(b) (1 point) When a hash function is used to provide message authentication, the hash function value is referred to as
Answer: C. Message Digest.
The output of a hash function, when used for message authentication, is typically called the message digest.

(c) (1 point) An orphan block is only created when 51% attack is successful.
Answer: B. False.
An orphan block is a block that is mined but not accepted by the main blockchain because another block is added at the same time. It doesn't require a 51% attack; a 51% attack can lead to reorganizing the blockchain, but orphan blocks can occur without one.

(d) (3 points) H is a hash function and for input X, it maps an output H(X). Which of the following properties does H satisfy? (can have multiple correct options)
Answer: A, D, E.

- **A. Efficiently computed.**
  A good hash function is designed to be computed quickly.
- **D. It is infeasible to determine X from H(X).**
  This is a key property of hash functions, known as pre-image resistance.
- **E. Change X slightly and H(X) changes significantly.**
  This is known as the avalanche effect, which ensures that small changes in the input lead to large, unpredictable changes in the output.

(e) (1 point) Which of the following is an example of a hash function?
Answer: A. SHA-1.
SHA-1 is a cryptographic hash function, while RSA, AES, and Diffie-Hellman are not.

(f) (1 point) Which algorithm provides the private key and its corresponding public key?
Answer: A. Key generation algorithm.
The key generation algorithm is used to generate both the private and public keys.

(g) (1 point) A digital signature needs a/an system
Answer: B. Asymmetric key.

A digital signature uses asymmetric (public-private key) cryptography.

(h) (1 point) Which of the following can the digital signature not provide for the message?
Answer: B. Confidentiality.
A digital signature ensures integrity, non-repudiation, and authentication, but not confidentiality.

(i) (1 point) What feature about enterprise blockchain is accurate?
Answer: B. Has trust problems.
Enterprise blockchains often have trust issues in centralized contexts, where participants may not trust each other as much as in fully decentralized blockchains.

(j) (1 point) What is a blockchain?
Answer: D. 4 - A distributed ledger on a peer-to-peer network.
Blockchain is a decentralized, distributed ledger technology that enables secure, transparent transactions on a peer-to-peer network.

(k) (1 point) What is a node?
Answer: D. A computer on a Blockchain network.
In a blockchain, a node refers to any computer that participates in the blockchain network.

(l) (1 point) What is a genesis block?
Answer: C. The first block of a Blockchain.
The genesis block is the very first block in a blockchain.

(m) (1 point) What does the block in the blockchain consist of?
Answer: D. All of these.
A block in the blockchain consists of transaction data, a hash point (hash of the previous block), and a timestamp.

(n) (3 points) Hash Functions used in Merkle Trees?
Answer: C. SHA-256.
Merkle trees typically use hash functions like SHA-256 to generate the hash for the leaves and inner nodes.

(o) (3 points) Does the following program have a compile error?

```
solidity

pragma solidity >=0.8.2 <0.9.0;
contract studentInfo {
    uint16 constant public studentID;
    uint32 public class;

    function setStudentID(string _studentID) public {
        studentID = _studentID;
    }

    function getStudentID() public pure returns(uint16) {
        return studentID;
    }

    function setClass(uint32 _class) public {
        class = _class;
    }

    function getClass() public pure returns(uint32) {
        return class;
    }
}
```

Answer: B. Yes, it has a compile error.
The variable studentID is a constant, which means it cannot be assigned a value after declaration. This will result in a compile-time error.

(p) (3 points) Does the following program have a compile error?

```
solidity
```

```
pragma solidity >=0.8.2 <0.9.0;
interface MoblieOS { }
interface PowerFullChipset { }
contract Phone is MoblieOS { }
contract IPhone is Phone, PowerFullChipset { }
contract Test1 {
    uint16 public price;
    function TestPhone() public {
        Phone ph = new Phone();
        Phone ph2 = new IPhone();
        IPhone iph = new IPhone();
    }
}
```

Answer: A. No, it does not have an error.
This contract is syntactically correct. Multiple inheritance is valid in Solidity, so IPhone can inherit from both Phone and PowerFullChipset.

(q) (3 points) Does the following program have a compile error?

solidity

```
pragma solidity >=0.8.2 <0.9.0;
contract Test1 {
    uint32 public testVar1;

    function setClass(uint32 _test) public {
        testVar1 = _test;
    }

    function getClass() public pure returns(uint32) {
        return testVar1;
    }

    function funTest1() public {
        if (((x + y) + 3) = 2 * (y + 5) + 7) {
            testVar1 = 23;
        }
        else {
            testVar1 = 2 * 23;
        }
    }
}
```

Answer: B. Yes, it has a compile error.
The issue is in the conditional if (((x + y) + 3) = 2 * (y + 5) + 7). The = is an assignment operator, not a comparison operator. It should be == for equality comparison.

(r) (3 points) Does the following program have a compile error?

```
solidity
pragma solidity >=0.8.2 <0.9.0;
contract ArrayTest {
    function ArrayTestFun() public {
        uint[] f1;
        uint[] f2;
    }
}
```

Answer: A. No, it does not have an error.
This program does not have a compile error. The arrays f1 and f2 are properly declared, even though they are not used.

Question 2 [34 points]

(a) (5 points) Which of the following statements are true?

solidity

复制代码

pragma solidity >=0.8.2 <0.9.0;

```
contract contract_A {

    uint private AX;

    uint public BX;

    string internal Cstr;


    constructor() {
        BX = 24;
    }

    function increment(uint Dr) private pure returns(uint) {
        return Dr + 1;
    }

    function updateAX(uint Dr) public {
        AX = Dr;
    }

    function getAX() public view returns(uint) {
        return AX;
    }

    function setCstr(string memory _Cstr) public;
    function getCstr() public view returns(string memory);
}

contract contract_B is contract_A {
    function setCstr(string memory _Cstr) public {
        Cstr = _Cstr;
    }

    function getCstr() public view returns(string memory) {
        return Cstr;
    }
```

```
}


contract contract_D {
    function readData() public payable returns(string memory, uint) {
        contract_A cA = new contract_A();
        cA.setCstr('Question 2a');
        cA.updateAX(25);
        return (cA.getAX(), cA.getCstr());
    }
}
```

Answer:

- **A. private increment function defined in contract_A is accessible in derived contract contract_B but not accessible outside the derived contracts.**
  False. The private function increment is **not accessible in the derived contract contract_B** because it is restricted to contract_A only.
- **B. private increment function defined in contract_A is accessible only inside contract_A and not accessible to derived contracts as well.**
  True. private functions are only accessible within the contract where they are defined (i.e., in contract_A).
- **C. internal variable Cstr defined in contract_A is accessible in derived contract contract_B but not outside the derived contracts.**
  True. The internal variable Cstr is accessible within the contract and by derived contracts, but **not outside** the contract or its derived contracts.
- **D. internal variable Cstr defined in contract_A is accessible only inside contract_A and not accessible to derived contracts as well.**
  False. internal variables are accessible in derived contracts.

(b) (5 points) Which of the following statements are true?

solidity

复制代码

```
pragma solidity >=0.8.2 <0.9.0;


contract contract_Q2B {
        uint input;
        mapping(address => uint) bal;


        constructor() public {
                input = 42;
        }


        function setInput(uint _input) private pure returns(bool) {
                input = _input;
                return true;
        }
}


contract sender {
        function setInput(uint _input) public payable {
                address _receiver = 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4;
                _receiver.transfer(100);
        }
}
```

Answer:

- **A. function () payable { bal[msg.sender] += msg.value; }**
  True. This is a valid fallback function definition that increases the balance of the sender's address when Ether is sent to the contract.
- **B. function (address _receiver) public payable { bal[msg.sender] += msg.value; }**

False. This is not a valid fallback function signature. A fallback function does not accept parameters, like _receiver.

- **C. function fallback() () public payable { bal[msg.sender] += msg.value; }**
  False. The syntax of the function is incorrect. The fallback function should be defined as function () external payable {}.
- **D. function () public payable { bal[msg.sender] += msg.value; }**
  True. This is a valid definition of a payable fallback function.
- **E. Fallback function is executed if contract receives plain Ether without any data.**
  True. A fallback function is triggered when the contract receives Ether without any data.
- **F. Fallback function is executed if caller calls a function that is not available.**
  True. If a caller calls a function that doesn't exist in the contract, the fallback function is executed.
- **G. If multiple unnamed functions are defined for a contract, the cheapest function is used as fallback function.**
  False. A fallback function must be unnamed and cannot be chosen based on cost.

(c) (8 points) Complete the following code. What string will return by retnStr of Child contract?

solidity

复制代码

```
abstract contract ParentA {

    function retnStr() public virtual returns (string memory) {

        return 'From ParentA';

    }

}


abstract contract ParentB {

    function retnStr() public virtual returns (string memory) {
        return 'From ParentB';
    }
}


contract Child is ParentA, ParentB {
    function retnStr() public override returns (string memory) {
        return super.retnStr(); // Calls the retnStr function from one parent contract
    }
}
```

Answer: The string returned will depend on the order of inheritance and how super is used. Since Child inherits from both ParentA and ParentB, Solidity resolves the function call to the first parent contract it encounters in the inheritance list. In this case, it will call ParentA's retnStr function because ParentA is listed first.

- **Answer: 'From ParentA'**.

(d) (5 points) Which statements about the following contract are true?

solidity

复制代码

```
pragma solidity >=0.8.2 <0.9.0;


abstract contract Book {
    string internal material = 'papyrus';
    constructor () {}
}


contract Encyclopedia is Book {
    constructor () {}
    function getMaterial() public view returns(string memory) {
        return super.material;
    }
}
```

Answer:

- **1. Both the contract Book and Encyclopedia compiles.**
  True. Both contracts compile without issues because super.material correctly refers to the inherited material variable in the parent contract.
- **2. The contract Book compiles, but contract Encyclopedia does not compile.**
  False. Encyclopedia compiles as it correctly inherits from Book and uses super.material.
- **3. The contract Encyclopedia does not compile because of a blank constructor, as child contracts cannot have blank constructors.**
  False. Child contracts can have an empty constructor if the parent contract does not require any specific initialization.
- **4. getMaterial returns 'papyrus'.**
  True. The getMaterial function returns the inherited material variable, which is 'papyrus'.
- **5. super cannot be used in Solidity.**
  False. super is valid in Solidity and refers to functions or variables in parent contracts.
- **6. super cannot be used for variable names.**
  False. super can be used to access inherited variables.

Answer: F. 1 and 4.

(e) (5 points) Is it possible to override the modifier? If possible, then complete the following code of MathBase and MathDivisor. If not possible, then complete the following code of MathBase only.

solidity

复制代码

```
contract MathBase {
    modifier exactDividedBy2And3(uint _a) {

        require(_a % 2 == 0 && _a % 3 == 0, "Not divisible by 2 and 3");

        _;

    }

}


contract MathDivisor is MathBase {

    modifier exactDividedBy2And3(uint _a) override {

        require(_a % 2 == 0 && _a % 3 == 0 && _a % 7 == 0, "Not divisible by 2, 3, and
7");

        _;

    }

}
```

Answer: Yes, it is possible to override a modifier in Solidity. The MathDivisor contract overrides the exactDividedBy2And3 modifier to include an additional condition for divisibility by 7.

(f) (6 points) Will the following code compile? If not, correct the code.

solidity

复制代码

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

pragma solidity ^0.8.20;


contract BigCat {

    event BigCatRoar(string message);


    function roar(uint level) public pure returns (string memory result) {

        require(level >= 0, "Not a valid Roar");
        require(level != 1, "Meow is not Roar");

        emit BigCatRoar("Valid Roar of Big Cat");

        return "Valid Roar of Big Cat";

    }

}


contract Lion is BigCat {

    event LionRoar(string message);

    BigCat public bigCat;


    constructor() {

        bigCat = new BigCat();

    }


    function roarCall(uint _i) public {

        try bigCat.roar(_i) returns (string memory result) {

            emit LionRoar(result);

        } catch {

            emit LionRoar("Roar call failed");

        }

    }

}
```

Answer:

The code **will not compile** because the try/catch statement needs to handle the error properly. The function roar in the BigCat contract has the pure modifier, which does not allow try/catch to catch errors. You can either remove the pure modifier or refactor the error handling.

Corrected code:

solidity

复制代码

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

contract BigCat {
    event BigCatRoar(string message);

    function roar(uint level) public returns (string memory result) {
        require(level >= 0, "Not a valid Roar");
        require(level != 1, "Meow is not Roar");
        emit BigCatRoar("Valid Roar of Big Cat");
        return "Valid Roar of Big Cat";
    }
}

contract Lion is BigCat {
    event LionRoar(string message);
    BigCat public bigCat;

    constructor() {
        bigCat = new BigCat();
    }

    function roarCall(uint _i) public {
        try bigCat.roar(_i) returns (string memory result) {
            emit LionRoar(result);
        } catch {
            emit LionRoar("Roar call failed");
        }
    }
}
```

Expected event output:

- roarCall(-1) will emit "Valid Roar of Big Cat".
- roarCall(0) will emit "Valid Roar of Big Cat".
- roarCall(1) will emit "Roar call failed".
- roarCall(2) will emit "Valid Roar of Big Cat".

Question 3 [20 points]

(a) (10 points) "Two contracts are used to raise money for several targets. The first sets the investment threshold rules. The second provides a method to invest tokens for each target." Is the following smart contract safe? If not, what type of vulnerabilities are associated with it?

Here's the code for analysis:

solidity

复制代码

```
contract RuleContract {
uint private base_threshold = 50000;
function getThreshold ( uint256 currentTarget )
public returns ( uint256 ) {
require ( currentTarget >= 2020 && currentTarget <= 2050) ;
if( currentTarget == 2020) {

return base_threshold ;
}
```

```
else {

return currentTarget *100;
}
}
}

contract InvestContract {
uint public total_2030 = 0;
uint public cur_invest = 0;
// some fixed address for rule contract
address fixed_rule = "0 xa214bd6 ...... "
function invest (
RuleContract rule ,
uint target ,
uint investToken ) public {
assert ( rule == fixed_rule ) ;
// the max threshold is 205000;
uint threshold = rule . getThreshold ( target ) ;
if( threshold == 203000 && threshold <= investToken ) {

total_2030 += investToken ;
cur_invest = investToken ;
} else if( investToken > 300000) {

cur_invest = investToken - threshold /100;
}
```

```
}
}
```

Analysis:

1. **assert (rule == fixed_rule) ; Vulnerability:**
   - The use of assert is risky here. The condition is checking if the rule contract's address matches the fixed address fixed_rule. This is a **logical error** because assert is meant to check for conditions that should always be true (invariants), and failing it consumes all the remaining gas. Instead, a more suitable approach is using require to ensure the contract address matches. The current implementation could lead to unintended reverts, wasting gas.
2. **Unreliable getThreshold() Function:**
   - The getThreshold() function in RuleContract only has a check for whether the currentTarget is between 2020 and 2050, but there's no proper check to handle incorrect or unexpected values, which could allow malicious actors to manipulate the target input and potentially cause the contract to behave unexpectedly. This can be fixed by improving input validation or adding bounds checks.
3. **Logical Issues in if and else if Clauses:**
   - The line if(threshold == 203000 && threshold <= investToken) could allow for unexpected behavior because the condition uses threshold == 203000, but then it checks if threshold is also less than or equal to investToken. This is contradictory because if threshold == 203000, it will never be less than or equal to investToken unless investToken is also exactly 203000.
   - Also, the line else if(investToken > 300000) is somewhat arbitrary. There should be more clarity on how the contract is intended to handle investments beyond this threshold.
4. **No Access Control:**
   - There is no access control on the invest() function. Anyone can call this function, which may allow malicious actors to perform unauthorized actions. The contract should include checks like onlyOwner or onlyAllowedAddress to control who can call the invest function.

5. **Possible Integer Overflow:**
    - If the contract allows a very large value for investToken, there's a potential for an **overflow** in the variable assignments (total_2030 += investToken;). Solidity provides protections for this with SafeMath or checked arithmetic (using unchecked or SafeMath), which should be used to avoid overflows.

Conclusion: The contract is not safe due to:

- The use of assert for contract address validation.
- Potential logical flaws in the threshold check.
- Lack of input validation and access control.
- The potential for integer overflow.

(b) (10 points) Is the following smart contract safe? If not, what type of vulnerabilities are associated with it?

Here’s the code for analysis:

solidity

复制代码

```
1 contract Trader {

2 TokenSale tokenSale = new TokenSale(); // Internal Contract defined at line in the same file

3 function combination () {

4 tokenSale.buyTokensWithWei ();

5 tokenSale.buyTokens(beneficiary);

6 }

7 }


8 contract TokenSale {

9 TokenOnSale tokenOnSale; // External Contract
```

```
...
function set (address _add) {
tokenOnSale = TokenOnSale(_add);
}
function buyTokens(address beneficiary) {
if (starAllocationToTokenSale > 0) {
tokenOnSale.mint(beneficiary, tokens);
}
}
function buyTokensWithWei() onlyTrader {
wallet.transfer(weiAmount);
}
```

Analysis:

1. **Uncontrolled Access to TokenSale Methods:**
   - The buyTokensWithWei() function in TokenSale is marked as onlyTrader, but there's no definition of who the trader is. Without proper access control or modifiers like onlyOwner or onlyTrader, this function could potentially be called by anyone, allowing them to send Ether to the wallet. This is a **critical access control issue**.
2. **Potential Reentrancy Risk:**
   - The wallet.transfer(weiAmount) call in buyTokensWithWei() can potentially lead to a **reentrancy attack** if wallet is a contract that calls back into TokenSale during the transfer. Although Solidity 0.8.x has built-in checks to mitigate this, it is a good practice to follow the Checks-Effects-Interactions pattern to avoid reentrancy issues. A more secure approach would be to use call instead of transfer to mitigate reentrancy risks, and follow the Checks-Effects-Interactions pattern.
3. **Use of External Contract Without Validation (TokenOnSale):**
   - TokenSale relies on an external contract TokenOnSale, and the set() function is used to set the address of this contract. However, there is no

validation of the address being set. A malicious user could set tokenOnSale to a malicious contract that implements the mint function in a harmful way. This is a **trust issue** that can be mitigated by validating the contract address in set() (e.g., ensuring it’s a valid TokenOnSale contract before setting it).

4. **Lack of Modifiers for Sensitive Functions:**
   - The buyTokens() function doesn't have an access modifier to restrict who can call it. Anyone can call it and trigger token minting. It should be restricted to only certain addresses (e.g., a privileged account, the owner, etc.) using an appropriate access control modifier.
5. **No Event Emission for Critical Functions:**
   - The contract lacks event emissions for critical functions like buyTokens() or buyTokensWithWei(). Emitting events provides transparency and allows users and external systems to track contract activity. This is a **best practice** that’s missing from the contract.
6. **Uninitialized Variables:**
   - There are variables like starAllocationToTokenSale and wallet that are not initialized in the code provided. If they are not initialized properly before being used, this could cause the contract to malfunction.

Conclusion: The contract is **not safe** due to:

- **Access control issues** on critical functions.
- **Potential reentrancy vulnerabilities.**
- Lack of **validation for external contract addresses.**
- **Missing events** for transparency.

Question 4 [10 points]

To fix the query and add the missing parts, we need to achieve three things:

1. **Calculate the daily transaction counts** — This part is already being done correctly by the lz_send_summary CTE.
2. **Add a running total of transactions** — This is done using the SUM(transaction_count) OVER (ORDER BY block_date) window function, which is correct, but we will ensure the proper sorting and aggregate the correct data.
3. **Add a 7-day moving average** — This part is missing, and we will need to use a

window function that looks at the previous 7 days' transaction counts and calculates the moving average.

Revised Query:

```
WITH lz_send_summary AS (
   SELECT
      date_trunc('day', block_time) AS block_date,
      COUNT(*) AS transaction_count    -- Fix to count transactions by day
   FROM layerzero.send
   GROUP BY 1
)
SELECT
   block_date,
   transaction_count,
   SUM(transaction_count) OVER (ORDER BY block_date) AS accumulate_transaction_count,
   AVG(transaction_count) OVER (ORDER BY block_date ROWS BETWEEN 6 PRECEDING AND CURRENT ROW) AS seven_day_moving_avg    -- Add 7-day moving average
FROM lz_send_summary
ORDER BY block_date;
```

Explanation of Changes:

1. **Fixing Transaction Count Calculation:**
   - The original query used MAX(block_number) AS transaction_count, which would return the maximum block number per day. This does not provide the daily transaction count, so we changed this to COUNT(*) AS transaction_count to count the number of transactions (rows) per day.
2. **Seven-Day Moving Average:**
   - To calculate the 7-day moving average of the transaction count, we use the AVG() window function:
   - AVG(transaction_count) OVER (ORDER BY block_date ROWS BETWEEN 6

    PRECEDING AND CURRENT ROW)

        - This calculates the average of the transaction_count for the current row and the previous 6 rows, effectively giving us a 7-day moving average.
        - The ROWS BETWEEN 6 PRECEDING AND CURRENT ROW specifies that the window should include the current row and the previous 6 rows (total of 7 days).

3. **Sorting the Results:**
    - Finally, we added ORDER BY block_date to ensure the results are sorted by block_date, which is crucial for the running total and moving average calculations to be correct.

Output Columns:

1. block_date — Date of the transaction.
2. transaction_count — The number of transactions on that specific day.
3. accumulate_transaction_count — The running total of transactions up to that date.
4. seven_day_moving_avg — The 7-day moving average of transaction counts.

This query should now provide the correct results: daily transaction counts, the running total of transactions, and the 7-day moving average of transactions, all sorted by date.

Question 5

Here is the implementation of the String contract with the functions you requested:

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

pragma solidity ^0.8.0;


contract String {


    // (a) compareTwoString function: Compares two strings and returns 0 if equal, 1
```

```
otherwise.
    function compareTwoString(string memory str1, string memory str2) public pure returns (uint) {
        if (keccak256(abi.encodePacked(str1)) == keccak256(abi.encodePacked(str2))) {
            return 0;    // Strings are equal
        } else {
            return 1;    // Strings are not equal
        }
    }

    // (b) concatenate function: Concatenates three strings and returns the result.
    function concatThreeString(string memory str1, string memory str2, string memory str3) public pure returns (string memory) {
        return string(abi.encodePacked(str1, str2, str3));
    }

    // (c) substring function: Extracts a substring from a given string between start and end index.
    function substring(string memory str, uint startIdx, uint endIdx) public pure returns (string memory) {
        require(startIdx <= endIdx, "Start index should be less than or equal to end index");
        require(endIdx < bytes(str).length, "End index exceeds string length");

        bytes memory strBytes = bytes(str);
        bytes memory result = new bytes(endIdx - startIdx + 1);

        for (uint i = startIdx; i <= endIdx; i++) {
            result[i - startIdx] = strBytes[i];
        }

        return string(result);
    }

    // (d) charAt function: Returns the character at the specified index of the string.
    function charAt(string memory str, uint index) public pure returns (bytes1) {
        require(index < bytes(str).length, "Index out of bounds");
        bytes memory strBytes = bytes(str);
        return strBytes[index];
    }

    // (e) replace function: Replaces a character at a specified position in the string.
    function replace(string memory str, bytes1 letter, uint position) public pure returns
(string memory) {
        require(position < bytes(str).length, "Position out of bounds");
        bytes memory strBytes = bytes(str);
        strBytes[position] = letter;
        return string(strBytes);
    }
}
```

Explanation of Functions:

(a) compareTwoString

This function compares two strings by using the keccak256 hash function, which hashes both strings and compares their hash values. If the hash values match, it means the strings are identical, and the function returns 0. Otherwise, it returns 1.

function compareTwoString(string memory str1, string memory str2) public pure returns

```
(uint) {
    if (keccak256(abi.encodePacked(str1)) == keccak256(abi.encodePacked(str2))) {
        return 0;    // Strings are equal
    } else {
        return 1;    // Strings are not equal
    }
}
```

(b) concatThreeString

This function concatenates three strings using the abi.encodePacked function, which concatenates the strings in a more efficient manner. The result is then cast back to a string.

```
function concatThreeString(string memory str1, string memory str2, string memory str3)
public pure returns (string memory) {
    return string(abi.encodePacked(str1, str2, str3));
}
```

(c) substring

This function extracts a substring from the given string starting from the startIdx and ending at endIdx. It uses a loop to copy the characters from the original string to a new bytes array, and then returns the new string.

```
function substring(string memory str, uint startIdx, uint endIdx) public pure returns
(string memory) {
    require(startIdx <= endIdx, "Start index should be less than or equal to end index");
    require(endIdx < bytes(str).length, "End index exceeds string length");

    bytes memory strBytes = bytes(str);
    bytes memory result = new bytes(endIdx - startIdx + 1);

    for (uint i = startIdx; i <= endIdx; i++) {
        result[i - startIdx] = strBytes[i];
    }


    return string(result);
}
```

(d) charAt

This function returns the character at a specified index of the string. It checks if the index is valid, and then retrieves the character using the bytes representation of the string.

```
function charAt(string memory str, uint index) public pure returns (bytes1) {
    require(index < bytes(str).length, "Index out of bounds");
    bytes memory strBytes = bytes(str);
    return strBytes[index];
}
```

(e) replace

This function replaces the character at a specific position with the provided letter. It ensures the position is valid before replacing the character.

```
function replace(string memory str, bytes1 letter, uint position) public pure returns
(string memory) {
    require(position < bytes(str).length, "Position out of bounds");
    bytes memory strBytes = bytes(str);
    strBytes[position] = letter;
    return string(strBytes);
}
```

Summary of Functions:

1. **compareTwoString(str1, str2)**: Compares two strings and returns 0 if they are equal, 1 otherwise.
2. **concatThreeString(str1, str2, str3)**: Concatenates three strings and returns the

result.

3. **substring(str, startIdx, endIdx)**: Extracts a substring from a string between specified start and end indices.
4. **charAt(str, index)**: Returns the character at the given index in the string.
5. **replace(str, letter, position)**: Replaces the character at a specific position in the string with a given letter.

This contract allows basic string operations and is designed to work efficiently with string data in Solidity.

Question 6

To implement the Merkle tree contract in Solidity, we'll break down the tasks and implement each function as requested. The contract will support creating a Merkle tree from a set of transactions, verifying the integrity of a specific transaction using a Merkle proof, and hashing a transaction to produce its corresponding Merkle tree node hash.

Merkle Tree Contract:

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

pragma solidity ^0.8.0;


contract MerkleTree {

    bytes32[] public hashes;

    string[4] transactions = [

        "TX1: Sherlock -> John",

        "TX2: John -> Sherlock",

        "TX3: John -> Mary",

        "TX4: Mary -> Sherlock"

    ];
    // (a) Constructor: Initializes the contract and constructs the Merkle Tree.
    constructor() {
        // Step 1: Initialize the leaf nodes (hashes of transactions)
        for (uint i = 0; i < transactions.length; i++) {
            bytes32 txHash = makeHash(transactions[i]);
            hashes.push(txHash);
        }

        // Step 2: Build the Merkle tree (compute parent hashes)
        uint len = hashes.length;
        while (len > 1) {
            for (uint i = 0; i < len / 2; i++) {
                bytes32 combinedHash = keccak256(abi.encodePacked(hashes[2 * i],
hashes[2 * i + 1]));
                hashes.push(combinedHash);
            }
            len = len / 2;
        }
    }

    // (b) Verify function: Verifies if a transaction is valid using the Merkle proof
    function verify(
        string memory transaction,
        uint index,
        bytes32 root,
        bytes32[] memory proof
    )
```

```
        public
        pure
        returns(bool)
    {
        bytes32 txHash = makeHash(transaction);
        bytes32 computedHash = txHash;

        for (uint i = 0; i < proof.length; i++) {
            if (index % 2 == 0) {
                computedHash = keccak256(abi.encodePacked(computedHash,
proof[i]));
            } else {
                computedHash = keccak256(abi.encodePacked(proof[i],
computedHash));
            }
            index = index / 2; // Move to the parent index in the tree.
        }

        // The final computed hash should match the root hash
        return computedHash == root;
    }

    // (c) makeHash function: Generates the hash of a transaction string.
    function makeHash(string memory input) public pure returns(bytes32) {
        return keccak256(abi.encodePacked(input));
    }
```

```
        // Getter function for root (first element in the hash array)
        function getRoot() public view returns(bytes32) {
            return hashes[hashes.length - 1];
        }
}
```

Explanation of Each Part:

(a) Constructor:

The constructor initializes the Merkle tree. It performs the following steps:

1. **Hash the transaction strings**: Each transaction is hashed using the makeHash function and stored in the hashes array.
2. **Build the Merkle tree**: The Merkle tree is built iteratively by pairing adjacent hashes, hashing them together, and storing the resulting parent hashes in the hashes array. This continues until only one hash remains, which is the root of the Merkle tree.

(b) Verify Function:

The verify function checks if a given transaction, identified by its index, is part of the Merkle tree corresponding to a given root. It does this by traversing the tree from the transaction up to the root, using the proof (a series of hashes) that is provided as part of the Merkle proof. Here's how the function works:

1. **Hash the transaction**: The provided transaction is first hashed.
2. **Iterate through the proof**: The proof consists of hashes that allow us to reconstruct the path from the transaction to the root. For each proof element, the function combines the current hash with the proof hash in the correct order (based on whether the current index is even or odd).
3. **Check the root**: After processing the entire proof, the computed hash should match the root provided to the function. If they match, the transaction is valid.

(c) MakeHash Function:

The makeHash function takes a transaction string as input and returns its Keccak-256 hash. This is the basic building block of the Merkle tree.

```
function makeHash(string memory input) public pure returns(bytes32) {
    return keccak256(abi.encodePacked(input));
}
```

This function is used both in the constructor to hash each transaction and in the verify function to hash the transaction being verified.

Additional Information:

1. **Merkle Tree Structure**: The tree starts with leaf nodes, each representing a hash of a transaction. The parent nodes are computed by hashing pairs of adjacent nodes until a single root node is reached.
2. **Proof Array**: The proof parameter in the verify function represents the series of hashes that will allow the function to reconstruct the path from the transaction to the root. For example, if a user wants to prove that "TX2" is part of the tree, they will need to provide the hashes of its sibling nodes and parent nodes up to the root.
3. **Root**: The Merkle tree's root is stored as the last element in the hashes array after the tree has been built. The function getRoot allows users to retrieve this root.

Example Use:

1. **Constructor**: When the contract is deployed, it automatically hashes the transactions and constructs the Merkle tree.
2. **Verify**: If someone wants to verify that "TX2" is part of the Merkle tree with the root rootHash, they would provide the transaction, the index (1 in this case for "TX2"), the root hash, and the Merkle proof (the hashes needed to verify the path from the leaf to the root).
3. **MakeHash**: A user can also call makeHash to hash any string, which can be useful for generating hashes of new transactions.

Example Flow for Verification:

- Let's say "TX2" is at index 1, and the root hash is known.
- To verify it, a user will need to provide the following:
  - The transaction: "TX2: John -> Sherlock".
  - The index: 1.
  - The root hash: e.g., 0xabc123... (root of the tree).

- A proof consisting of hashes of the sibling nodes, which will allow the function to traverse the tree upwards and check if it reaches the root. If it does, the verification returns true, otherwise false.

This contract provides the basic structure for a Merkle tree and supports verifying data integrity with Merkle proofs in a decentralized manner.

Question 7

Here is the Solidity implementation of the contract you described:

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

pragma solidity ^0.8.0;


contract Competition {

    // EVENTS

    event NewSolutionFound(address indexed _addressOfWinner, uint256 _solution);

    event BountyTransferred(address indexed _to, uint256 _amount);

    event BountyIncreased(uint256 _amount);

    event CompetitionTimeExtended(uint256 _to);


    // STATE VARIABLES

    address public owner;

    uint256 public x1;

    uint256 public x2;

    uint256 public bestSolution;

    address public addressOfWinner;

    uint256 public durationInBlocks;

    uint256 public competitionEnd;

    uint256 public claimPeriodeLength;
    // Constructor

    constructor(uint256 _durationInBlocks) {

        owner = msg.sender;

        bestSolution = 0;

        durationInBlocks = _durationInBlocks;

        competitionEnd = block.number + _durationInBlocks;

        addressOfWinner = address(0); // Zero address indicates no winner yet

        // Assuming an average blocktime of 14 seconds, so 86400/14 = 6172 blocks

        claimPeriodeLength = 6172;

    }


    // Fallback function: Returns any funds sent to the contract address directly

    receive() external payable {

        payable(msg.sender).transfer(msg.value);

    }


    // Private function to calculate the new solution

    function _calculateNewSolution(uint256 _x1, uint256 _x2) private pure returns
(uint256) {

        // Check new parameters against constraints

        require(_x1 <= 40, "x1 must be <= 40");

        require(_x2 <= 35, "x2 must be <= 35");

        require((3 * _x1) + (2 * _x2) <= 200, "(3 * x1) + (2 * x2) must be <= 200");

        require(_x1 + _x2 <= 120, "x1 + x2 must be <= 120");

        require(_x1 > 0 && _x2 > 0, "x1 and x2 must be greater than 0");
        // Calculate and return new solution
        return (4 * _x1) + (6 * _x2);
    }

    // Public function to submit a solution
    function submitSolution(uint256 _x1, uint256 _x2) public returns (uint256) {
        uint256 newSolution = _calculateNewSolution(_x1, _x2);

        // Ensure the new solution is better than the current best solution
        require(newSolution > bestSolution, "Solution must be better than the current
best solution");

        // Save the solution and its values
        x1 = _x1;
        x2 = _x2;
        bestSolution = newSolution;
        addressOfWinner = msg.sender;

        // Emit the event for the new solution
        emit NewSolutionFound(msg.sender, newSolution);

        return newSolution;
    }

    // Public function to claim the bounty
    function claimBounty() public {
        require(block.number > competitionEnd, "Competition has not ended yet");
        if (addressOfWinner == address(0)) {
            // No solution was found -> extend duration of competition
            competitionEnd = block.number + durationInBlocks;
        } else {
            require(block.number < competitionEnd + claimPeriodeLength, "Claim
period has ended");
            require(msg.sender == addressOfWinner, "Only the winner can claim the
bounty");

            // Transfer the bounty to the winner
            uint256 balance = address(this).balance;
            payable(addressOfWinner).transfer(balance);

            // Extend the competition duration
            competitionEnd = block.number + durationInBlocks;

            // Emit the BountyTransferred event
            emit BountyTransferred(addressOfWinner, balance);
        }
    }

    // Public function to top-up the bounty
    function topUpBounty() public payable {
        emit BountyIncreased(msg.value);
    }
    // Public function to extend the competition time if no solution was found
    function extendCompetition() public {
        // Only if no valid solution has been submitted
        require(addressOfWinner == address(0), "A solution has already been submitted");
        require(block.number > competitionEnd + claimPeriodeLength, "Claim period has not ended yet");

        // Extend the competition duration
        competitionEnd = block.number + durationInBlocks;

        // Reset winner address
        addressOfWinner = address(0);

        // Emit the CompetitionTimeExtended event
        emit CompetitionTimeExtended(competitionEnd);
    }
}
```

Explanation of the Code:

State Variables:

- owner: The address of the owner of the contract (set during deployment).
- x1, x2: The parameters for the solution.
- bestSolution: The current best solution found.
- addressOfWinner: The address of the winner (the person who submitted the best solution).
- durationInBlocks: The duration of the competition in blocks.
- competitionEnd: The block number when the competition ends.
- claimPeriodeLength: The length of the claim period (set to 6172 blocks,

assuming 14-second block times).

Constructor:

- Initializes the contract with the durationInBlocks parameter.
- Sets the competition end time (competitionEnd) and the claim period length.

Fallback Function (receive()):

- Allows the contract to accept direct payments and return any funds sent to it to the sender.

Private Function (_calculateNewSolution):

- Validates the inputs (_x1, _x2) according to the given constraints.
- Calculates and returns the new solution if the inputs are valid.

Public Function (submitSolution):

- Allows participants to submit their solution by providing values for _x1 and _x2.
- The function checks that the new solution is better than the current best solution and saves the solution.
- The event NewSolutionFound is emitted when a new solution is submitted.

Public Function (claimBounty):

- Allows the winner to claim the bounty after the competition has ended.
- If no solution was found, the competition duration is extended.
- The winner can claim the bounty if they are the correct address and the claim period is still open.

Public Function (topUpBounty):

- Allows anyone to send Ether to the contract to increase the bounty. The event BountyIncreased is emitted when the bounty is topped up.

Public Function (extendCompetition):

- If no valid solution has been submitted, the owner can extend the competition duration.
- Resets the addressOfWinner to the zero address and extends the competition time.

- The event CompetitionTimeExtended is emitted when the competition duration is extended.

Events:

- NewSolutionFound: Emitted when a new solution is found and submitted by a participant.
- BountyTransferred: Emitted when the bounty is successfully transferred to the winner.
- BountyIncreased: Emitted when the bounty is increased (i.e., when funds are sent to the contract).
- CompetitionTimeExtended: Emitted when the competition duration is extended.

Usage Flow:

1. The contract owner deploys the contract by specifying the competition duration in blocks.
2. Participants submit solutions using the submitSolution method.
3. The winner can claim the bounty using the claimBounty method after the competition ends.
4. If no valid solution is found, the owner can extend the competition time using extendCompetition.
5. Anyone can top-up the bounty using topUpBounty.

This contract covers the requested functionality and events to manage a competition, handle the submission of solutions, manage the bounty, and allow extensions to the competition duration.