NAME : ANSHIKA SINGH

Roll no : 230163

BLOCKBLOOM Assignment 3:

Q1) a) contract SimpleFactorial {

```
function factorial(uint x) public pure returns (uint) {      // Function to calculate factorial using  loop
    require(x >= 0, "Input must be non-negative number.");   // Make sure input is non-negative no.
    uint result = 1;          // Start the result at 1 (factorial of 0 is 1)
    for (uint i = 1; i <= x; i++) {    // Multiply numbers from 1 to x
        result *= i;
    }
     return result;      // Return the final factorial
}
```

b) function factRec(uint x) public pure returns (uint) {   //f unction to calculate fact using recursion

```
    require(x >= 0, "Input must be a non-negative number.");
    if (x == 0 || x == 1) {                // Base case: factorial of 0 or 1 is 1
        return 1;   }
    return x * factRec(x - 1);   }   // Recursive case: x * factorial of (x - 1)
```

Gas Fee Comparison:

1. Iterative Function (factorial)

The loop in factorial iterates x times, with each iteration involving a multiplication operation.

Gas costs increase linearly with x because the number of iterations is proportional to the input size.

The loops are more gas-efficient compared to recursive calls for tasks like this.

Gas Cost Factors:- Cost of initializing the loop, incrementing the counter, and performing the multiplication operation per iteration.

2. Recursive Function (factRec)

Each recursive call consumes additional gas for function call overhead and multiplication.

Gas costs grow exponentially with x because each recursion adds a new stack frame.

Gas Cost Risks:- For large inputs, the recursion could cause the transaction to fail due to out-of-gas errors or stack overflow.

Comment:- The iterative approach (factorial) is the preferred method for calculating factorials due to its predictable and lower gas costs, as well as its safety against stack overflow and out-of-gas errors. Recursive functions, while elegant in theory, are less suitable for blockchain environments like Ethereum due to these limitations.

Q2) Modifiers are special functions used to add reusable conditions or logic to other functions. They allow developers to enforce rules such as access control or input validation before a function is executed. Modifiers are declared using the modifier keyword and can be applied to functions using their name.

Types of Modifiers:

1) Visibility Modifiers:- These control who can call a function or access a state variable.

Examples: public: Accessible from anywhere.

internal: Accessible only within the contract and derived contracts.

private: Accessible only within the contract.

external: Accessible only from outside the contract.

2) Mutability Modifiers:- These describe whether a function modifies the blockchain state.

Examples: pure: Does not read or modify state variables.

view: Reads state variables but does not modify them.

No modifier: Indicates the function can modify state variables.

Updated Code with onlyOwner Modifier:

```
contract FactorialWithOwner {
    address public owner;
    constructor() {       // Constructor to set the owner at deployment
        owner = msg.sender; }    // msg.sender is the address deploying the contract
    modifier onlyOwner() {       // Modifier to restrict access to the owner only
        require(msg.sender == owner, "Caller is not the owner.");   }
```

Function to change the owner

```
    function changeOwner(address newOwner) public onlyOwner {
        require(newOwner != address(0), "New owner cannot be the zero address.");
```

```
        owner = newOwner;  }
```

Factorial using a loop (accessible to anyone)

```
    function factorial(uint x) public pure returns (uint) {
        require(x >= 0, "Input must be a non-negative number.");
        uint result = 1;
        for (uint i = 1; i <= x; i++) {
            result *= i;  }
        return result;   }
```

Factorial using recursion (restricted to the owner)

```
    function factRec(uint x) public pure onlyOwner returns (uint) {
        require(x >= 0, "Input must be a non-negative number.");
        if (x == 0 || x == 1) {
            return 1;  }
        return x * factRec(x - 1);
    }
}
```

Q3) Error Handling Mechanisms provides several mechanisms for handling errors to ensure contracts behave as expected and handle unexpected conditions gracefully. The primary keywords for error handling are require, assert, and revert. Each has specific use cases based on the nature of the error and the desired outcome

1. require

Purpose:- Used to validate inputs and conditions before executing a function.

It reverts the transaction if the condition evaluates to false.

2. assert

Purpose:- Used to check for internal errors and invariants (conditions that should never fail).

It reverts the transaction if the condition evaluates to false.

3. revert

Purpose:- Used to explicitly trigger an error and revert the transaction.

Allows for custom error messages.

Q4) Implementing selfdestruct in a Bank Contract:

```
contract SimpleBank {
```

```
    address public owner;     // Address of the owner (person who created the contract)
    mapping(address => uint) public balances;      // Store how much Ether each user has deposited
    constructor() {        // Constructor: Runs only once when the contract is created
        owner = msg.sender;  }   // Set the owner to the person who deployed the contract

    modifier onlyOwner() {      // A rule that allows only the owner to do certain actions
        require(msg.sender == owner, "You are not the owner.");  }  // Stop if the caller is not the owner

    // Function to let users deposit Ether into the contract
    function deposit() public payable {
        require(msg.value > 0, "You must deposit some Ether.");    // Ensure a valid amount is sent
        balances[msg.sender] += msg.value;  }  // Add the amount to the sender's balance
    // Function to let users withdraw Ether they have deposited
    function withdraw(uint amount) public {
        require(amount > 0, "Withdrawal amount must be greater than zero."); // Check for a valid
amount
        require(balances[msg.sender] >= amount, "Not enough balance."); // Ensure they have enough
balance
        balances[msg.sender] -= amount;    // Deduct the amount from their balance
        payable(msg.sender).transfer(amount);  }   // Send the Ether to the user

    // Function to close the bank (only the owner can do this)
   function closeBank() public onlyOwner {   // It deletes contract and send all remain Ether to owner
        selfdestruct(payable(owner)); } // Destroy the contract and send any remaining Ether to owner
}
```

Understanding the Impact of Selfdestruct on Smart Contracts:

1. Contract Deletion:- When the selfdestruct function is executed in a smart contract, it removes the contract's bytecode from the blockchain entirely. This action deletes all the data and code associated with the contract, rendering it completely inaccessible and unusable thereafter.

2. Ether Management:- At the time of selfdestruct, any remaining Ether within the contract is automatically transferred to a designated address. This process allows the contract owner to securely close the contract and recover any leftover funds, ensuring nothing is lost.

3.Gas Incentives:- Selfdestruct also offers a gas refund, which acts as a return of some fees associated with blockchain operations. This refund serves as an incentive for developers to remove contracts that are no longer needed, helping to maintain an efficient and organized blockchain environment.

Exploring the selfdestruct Opcode:

1.Opcode Functionality:- The EVM opcode SELFDESTRUCT removes the contract and sends its remaining balance to the specified address.

Syntax: selfdestruct(address recipient)

2.Gas Behavior:- Provides a gas refund to reduce network congestion by cleaning up unnecessary storage and code.

Security Implications

1.Advantages:- Allows safe recovery of funds from unused or deprecated contracts. Helps clean up contracts that are no longer needed, reducing blockchain bloat.

2.Risks:- Unexpected Contract Destruction:- If a malicious actor gains access to the selfdestruct function, they could destroy the contract and steal its funds.

Solution: Use strict access control (e.g., onlyOwner modifier).

Q5) 1. function transferEther() public payable returns (address, uint) {

return (msg.sender, msg.value); }

Key Features:

1. Payable Function: This function allows for sending Ether by making a transaction that involves transferring funds.

2. Provides Two Details: `msg.sender`: Displays the address of person who initiated the transaction.

`msg.value`: Shows the amount of Ether included in the transaction.

3. Behavior in Remix: No Immediate Output: When you use the `transferEther` function, results are not immediately visible in the Remix interface.

Explanation: The function triggers a blockchain transaction by involving Ether transfer, which alters states, although no state is changed in this specific case.

Solidity transactions do not provide direct outputs. Instead, the outcomes are recorded in the transaction receipt or event logs.

To view the transaction result in Remix, check the transaction details in the terminal logs or examine the event logs.

2. Function: display()

Key Characteristics:

1.View Function: This function is read-only and does not change the contract state.

2.Returns One Value: msg.sender: The address of the caller.

3.Behavior in Remix: Output Shown Directly: When you call display, the return value is immediately visible in Remix under the function call button.

Reason: This function does not trigger a transaction; it is a simple "call" operation.

Calls are executed locally (off-chain) and return the result immediately to the caller, making it easy for Remix to display the output directly.

General Rule:

1.When Output is Displayed in Remix: Functions marked as view or pure return results directly because they do not modify the blockchain state.

These functions are treated as calls and are executed locally on your machine.

2.When Output is Not Displayed: Functions that involve state changes or payable functions always result in a transaction.

The output is not returned directly but is logged in the transaction receipt, which you can view in the terminal or transaction details.

Q6) A Denial of Service (DoS) attack is designed to make a service or system unavailable to its users. In the realm of blockchain and smart contracts, such attacks target the contract's resources or its logic to disrupt normal operations. This may result in excessive gas consumption, transaction blockages, or complete stoppage of the contract.

In Solidity, a DoS attack can occur through various methods:

1. Block Gas Limit: Attackers can manipulate transactions to need more gas than a block can handle, preventing other transactions from being processed.

2. Reentrancy Attack: This involves repeatedly calling a vulnerable contract before it can complete its initial task, which disrupts its normal execution.

3. Insufficient Gas for Transactions: Some contracts have functions that may run out of gas due to their complexity, making them targets for attacks.

4. Empty State Variables: Attackers can target contracts that depend on specific states, forcing them to fail. For instance, a simple DoS attack might involve causing a contract function to use excessive gas, thus blocking its operation.

Code:

```
contract DoSAttack {
    address public owner;          // The owner of the contract
```

```
mapping(address => uint256) public balances;          // This keeps track of each user's balance
constructor() {         // This is the constructor, it runs once when the contract is deployed
    owner = msg.sender;   }      // Set the owner to the person who deploys the contract
function deposit() external payable {        // This function allows users to deposit ETH into contract
    require(msg.value > 0, "You must send ETH");    // Make sure they send some ETH
    balances[msg.sender] += msg.value;  }   // Add the sent ETH to the sender's balance
// This function allows users to withdraw ETH from the contract
function withdraw(uint256 amount) external {
    require(balances[msg.sender] >= amount,"No enough balance"); // Check they have enough ETH
    balances[msg.sender] -= amount;  // Deduct the withdrawn amount from their balance
    // This line tries to send the ETH back to the user
    // However, if there's a problem (like running out of gas), it will fail
    payable(msg.sender).transfer(amount);  }
// This is the malicious attack function
// It creates an infinite loop to use up all the gas, causing the contract to fail
function attack() external payable {
    while (true) {}    }    // Infinite loop to block the contract
// This function allows anyone to check the balance of the contract
function contractBalance() external view returns (uint256) {
    return address(this).balance;   }   // Returns the contract's ETH balance
}
```