Task 3 (ARYAVART)

Ques 1

Code Snippet:

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

contract FactorialCalculator {

    // ITERATIVE
    function factorial(uint x) public pure returns (uint) {
        uint result = 1;
        for (uint i = 1; i <= x; i++) {
            result *= i;
        }
        return result;
    }

    // RECURSIVE
    function factRec(uint x) public pure returns (uint) {
        if (x == 0) {
            return 1;
        }
        return x * factRec(x - 1);
    }
}
```

GAS FEES: Gas fees in Solidity refer to the computational cost required to execute a function or transaction on the Ethereum network.
Gas Fee Comparison:

- **Iterative Approach (factorial):**
  - Time complexity of this function is lower and thereby gas fees is generally low for this.
- **Recursive Approach (factRec):**
  - Time complexity of this function is higher and thereby gas fees is generally high for this.

Ques 2

Function Modifiers: Function behavior can be changed using function modifiers. Function modifier can be used to automatically check the condition prior to executing the function. The modifiers can be used when there is a need to verify the condition automatically before executing a particular function.

```
Syntax:
Modifier modifier_name
{
    // action to be taken
}
```

Visibility Modifiers: The function visibility is set in the parent contract according to which the child contract can use its parent's function. The function visibility of the function provides the accessibility of the function to the child contract. The visibility defined in the parent contract is used to identify whether the child contract can use the function or not or if it is private to the parent contract only. The four function visibility modifiers are: public, private, internal, and external.

Mutability Modifiers: It defines the behaviour of functions and how they interact with data stored on the blockchain. The two major state mutability modifiers in Solidity are:

- View
- Pure

Code of Ques 1 after making necessary changes:

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

contract FactorialCalculator {
    address public owner;

    // Constructor to set the initial owner
    constructor() {
        owner = msg.sender;
    }

    // Modifier to restrict access to the owner
    modifier onlyOwner() {
        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;
    }
```

```
    // Iterative function to calculate factorial
    function factorial(uint x) public pure returns (uint) {
        uint result = 1;
        for (uint i = 1; i <= x; i++) {
            result *= i;
        }
        return result;
    }

    // Recursive function to calculate factorial (restricted to owner)
    function factRec(uint x) public pure onlyOwner returns (uint) {
        if (x == 0) {
            return 1;
        }
        return x * factRec(x - 1);
    }
}
```

Ques 3

Error Handling in Solidity:

- **Require:** The require function checks a condition and throws an error if the condition is not met. It is typically used to validate inputs and conditions before executing the main logic of the function.
- **Revert:** The revert function is used to manually trigger an error. It can be used for more complex error handling, where specific conditions need to be checked and an error message needs to be returned.
- **Assert:** The assert function checks for conditions that should never occur. If an assert statement fails, it indicates a bug in the code, and the transaction is reverted without any gas refund.

Ques 4

The self-destruct function in Solidity: The self-destruct function in Solidity is used to destroy a smart contract and transfer its remaining Ether balance to a specified address. Once a contract is destroyed, it can no longer be interacted with.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.25;

contract Bank {
    mapping(address => uint) private balanceLedge;
    address public owner;

    modifier balanceCheck(uint amt) {
        require(balanceLedge[msg.sender] >= amt, "Insufficient Balance");
        _;
    }

    modifier onlyOwner() {
        require(msg.sender == owner, "Only the owner can perform this action");
        _;
    }

    constructor() {
        owner = msg.sender;
    }

    function deposit() public payable {
        balanceLedge[msg.sender] += msg.value;
    }

    function withdraw(uint _amt) public balanceCheck(_amt) {
        balanceLedge[msg.sender] -= _amt;
        payable(msg.sender).transfer(_amt);
    }

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

    function transfer(uint _amt, address recipient) public balanceCheck(_amt) {
        balanceLedge[msg.sender] -= _amt;
        balanceLedge[recipient] += _amt;
    }

    function closeBank(address payable recipient) public onlyOwner {
        selfdestruct(recipient);
    }
}
```

closeBank Function:

- Calls the self-destruct opcode to destroy the contract.
- Transfers any remaining Ether in the contract to the specified recipient address.

Effect on Smart Contract Behavior:

- The code and storage of the contract are deleted from the blockchain.
- Any interactions with the contract do not exist anymore.
- The contract's remaining Ether is sent to the address specified in self-destruct.
- Gas Refund: There's a gas refund for clearing the contract's storage and code from the blockchain.

self-destruct Status Update:

- **EIP-4758 Proposal:** There is a current Ethereum community discussion on the possible deprecation of the self-destruct opcode. Reasons include potential misuse by intermediaries and challenges for state management in blockchain systems.
- **Alternatives:** Rather than calling self-destruct, developers are encouraged to provide mechanisms to disable or lock up contract functionality whilst not losing state.

Ques 5

For transferEther Function: When this function is executed, output is not shown in Remix IDE. The reason for that could be: This function is payable type and the payable type functions that handle Ether are transactional functions. And, the transactional functions do not directly return the outputs to the caller in Remix IDE.

For display Function:

- This function is marked as view and these do not modify the blockchain state. They are executed as read-only operations and their output can be directly retrieved and displayed by Remix.

General Rules for Output Display in Remix:

- **Transactional Functions:**
  - Functions that modify the state are processed as transactions.
  - Their output is not shown in Remix.
- **View/Pure Functions:**
  - Functions marked as view or pure are executed locally without creating a transaction.
  - Their output is displayed immediately in Remix.