Solidity Math Operations Quiz

Quiz 1: Basic Math Operations in Solidity

Instructions: You're helping a market vendor create a simple calculator for adding up customer purchases. Look at the code below and find the mistake.

```
pragma solidity ^0.8.0;

contract PalengkeCalculator {
    // Function to calculate total price of items
    function calculateTotal(
        uint256 ricePrice,
        uint256 fishPrice,
        uint256 vegetablePrice
    ) public pure returns (uint256) {
        // Add up all items
        uint256 totalPrice = ricePrice + fishPrice - vegetablePrice;
        return totalPrice;
    }
}
```

What's wrong with this calculator function?

- A) It should subtract the prices instead of adding them
- B) It adds rice and fish prices but subtracts vegetable price
- C) It uses the wrong data type for prices
- D) It doesn't include a tax calculation

Answer: B) It adds rice and fish prices but subtracts vegetable price

Explanation: The function is supposed to add up all the prices to get the total cost, but there's a mistake in the math. Instead of adding the vegetable price (`+`), it's subtracting it (`-`). This would make the total lower than it should be! The correct line should be: `uint256 totalPrice = ricePrice + fishPrice + vegetablePrice;`

Quiz 2: Integer Division in Solidity

Instructions: A market vendor wants to divide their earnings equally among 3 people. Look at their calculation and choose the correct result.

```
pragma solidity ^0.8.0;

contract EarningsSplitter {
    function splitEarnings(uint256 totalEarnings) public pure returns (uint256) {
        // Divide total earnings among 3 people
        uint256 amountPerPerson = totalEarnings / 3;
```

```
        return amountPerPerson;
    }
}
```

If `totalEarnings` is 100, what will `amountPerPerson` be?

- A) 33.33
- B) 33
- C) 34
- D) 33 with a remainder of 1

Answer: B) 33

Explanation: In Solidity, division with integers (whole numbers) always rounds down to the nearest whole number. So 100 ÷ 3 = 33.33..., but Solidity will just return 33. The remainder (1) is lost in this calculation. This is different from regular math where you might get a decimal value. If you need to keep track of the remainder, you would need to use the modulo (%) operator.

Quiz 3: Using Modulo in Solidity

Instructions: A market stall is tracking inventory, and needs to know if they have enough fruits to make complete fruit baskets (5 fruits per basket). Which function correctly tells them if they have any leftover fruits?

```
pragma solidity ^0.8.0;

contract FruitBasketChecker {
    // Option A
    function checkLeftoverFruits1(uint256 fruitCount) public pure returns
(uint256) {
        return fruitCount / 5;
    }

    // Option B
    function checkLeftoverFruits2(uint256 fruitCount) public pure returns
(uint256) {
        return fruitCount % 5;
    }

    // Option C
    function checkLeftoverFruits3(uint256 fruitCount) public pure returns (bool) {
        return fruitCount > 5;
    }

    // Option D
    function checkLeftoverFruits4(uint256 fruitCount) public pure returns
(uint256) {
        return 5 - fruitCount;
    }
}
```

Which function correctly returns the number of leftover fruits?

- A) Option A
- B) Option B
- C) Option C
- D) Option D

Answer: B) Option B

Explanation: Option B uses the modulo operator (`%`), which gives you the remainder after division. If you have 17 fruits and each basket needs 5 fruits, you can make 3 complete baskets (15 fruits), with 2 fruits left over. The calculation 17 % 5 = 2 tells you exactly how many fruits are left over. Option A gives you how many full baskets you can make, not the leftover fruits. Option C just checks if there are more than 5 fruits. Option D would give a negative number if there are more than 5 fruits.

Quiz 4: Preventing Overflow/Underflow in Math Operations

Instructions: In older versions of Solidity (before 0.8.0), there was a risk of numeric overflow. What happens in this code if it were run in Solidity 0.7.0?

```
pragma solidity ^0.7.0;

contract PriceTracker {
    uint8 public maxPrice = 255; // Maximum value for uint8

    function increasePrice() public {
        maxPrice = maxPrice + 1;
    }
}
```

What would happen when `increasePrice()` is called?

- A) `maxPrice` becomes 256
- B) `maxPrice` becomes 0
- C) The function reverts with an error
- D) `maxPrice` stays at 255

Answer: B) `maxPrice` becomes 0

Explanation: In Solidity versions before 0.8.0, if a number got too big for its data type, it would "roll over" (like a car odometer). Since `uint8` can only store numbers from 0 to 255, when you add 1 to 255, it rolls over to 0 instead of becoming 256. This is called an "overflow."

In Solidity 0.8.0 and later, this would cause the transaction to fail with an error message because overflow checking is built-in. This is why newer versions of Solidity are safer for math operations!

Solidity Require Statements Quiz

Quiz 1: Understanding require() Statements

Instructions: You're helping to secure a Community Fund smart contract. Which of these explains what the `require()` statement does in Solidity?

```
pragma solidity ^0.8.0;

contract CommunityFund {
    address public owner = msg.sender;

    function withdraw(uint256 amount) public {
        require(msg.sender == owner, "Only the owner can withdraw");
        // Code to withdraw funds
    }
}
```

What does the `require()` statement do in this function?

- A) It simply displays a warning message but allows the function to continue
- B) It checks a condition and reverts the transaction if the condition is false
- C) It's just a comment to remind the developer about a rule
- D) It logs an error message to the blockchain but doesn't stop the function

Answer: B) It checks a condition and reverts the transaction if the condition is false

Explanation: Think of `require()` as a security guard at the entrance of a function. It checks if a condition is true (in this case, if the person calling the function is the owner). If the condition is true, the function continues normally. But if the condition is false, the function immediately stops, all changes are undone, and the error message "Only the owner can withdraw" is returned to the user. This is how Solidity protects functions from being used in ways they shouldn't be.

Quiz 2: Multiple Conditions with require()

Instructions: A smart contract needs to verify multiple conditions before allowing a donation. Which option correctly uses `require()` statements to check both conditions?

```
pragma solidity ^0.8.0;

contract DonationChecker {
    function donate(uint256 amount) public payable {
        // We need to check:
        // 1. The donation amount must be greater than zero
        // 2. The amount of Ether sent must match the donation amount

        // Option A
        require(amount > 0);
        require(msg.value == amount);

        // Option B
```

```
        require(amount > 0 && msg.value == amount);

        // Option C
        require(amount > 0, "Amount must be greater than zero");
        require(msg.value == amount, "Sent value must match amount");

        // Option D
        if(amount > 0 && msg.value == amount) {
            require(true);
        }

        // Rest of the function
        // ...
    }
}
```

Which option is the best way to check these conditions?

- A) Option A
- B) Option B
- C) Option C
- D) Option D

Answer: C) Option C

Explanation: Option C is best because it uses separate `require()` statements with clear error messages for each condition. This helps users understand exactly what went wrong if their transaction fails. For example, if they try to donate 0 tokens, they'll see "Amount must be greater than zero." If they send the wrong amount of Ether, they'll see "Sent value must match amount." Using separate statements with helpful messages makes your code more user-friendly and easier to debug.

Quiz 3: Using require() for Input Validation

Instructions: The contract below is supposed to set a minimum age requirement for users. There's a problem with the validation. Can you spot it?

```
pragma solidity ^0.8.0;

contract AgeVerifier {
    uint256 public minimumAge = 18;
    mapping(address => uint256) public userAges;

    function setUserAge(uint256 age) public {
        require(age < minimumAge, "User must be of minimum age");
        userAges[msg.sender] = age;
    }
}
```

What's wrong with the `require()` statement in this contract?

- A) It should use >= instead of <
- B) It's missing an error message
- C) It should check msg.sender, not age
- D) require() can't be used for age verification

Answer: A) It should use >= instead of <

Explanation: The `require()` statement has the logic backwards! It says `require(age < minimumAge)`, which means the function will only continue if the age is LESS than the minimum age (exactly the opposite of what we want). It should be `require(age >= minimumAge, "User must be of minimum age")` to ensure the user is at least the minimum age. This is a common mistake when writing conditions - always double-check your comparison operators (<, >, <=, >=).

Quiz 4: Using require() for Fund Protection

Instructions: A community fund contract needs to prevent withdrawals that exceed the available balance. Which function correctly implements this check?

```
pragma solidity ^0.8.0;

contract ProtectedFund {
    uint256 public totalFunds = 100;
    address public owner = msg.sender;

    // Option A
    function withdraw1(uint256 amount) public {
        require(msg.sender == owner);
        totalFunds -= amount;
        payable(owner).transfer(amount);
    }

    // Option B
    function withdraw2(uint256 amount) public {
        require(amount <= totalFunds, "Insufficient funds");
        require(msg.sender == owner, "Only owner can withdraw");
        totalFunds -= amount;
        payable(owner).transfer(amount);
    }

    // Option C
    function withdraw3(uint256 amount) public {
        if (amount <= totalFunds && msg.sender == owner) {
            totalFunds -= amount;
            payable(owner).transfer(amount);
        }
    }

    // Option D
    function withdraw4(uint256 amount) public {
        require(msg.sender == owner, "Only owner can withdraw");
        if (amount <= totalFunds) {
```

```
            totalFunds -= amount;
            payable(owner).transfer(amount);
        }
    }
}
```

Which function correctly protects the funds?

- A) Option A
- B) Option B
- C) Option C
- D) Option D

Answer: B) Option B

Explanation: Option B is correct because it uses `require()` statements to check both necessary conditions before allowing the withdrawal:

1. It checks if there are enough funds available (`amount <= totalFunds`)
2. It verifies that only the owner can withdraw (`msg.sender == owner`)

If either condition fails, the transaction will be reverted and no funds will be lost. Option A doesn't check if there are enough funds. Option C uses an if statement instead of require, which means it would silently do nothing if the conditions aren't met (no error message to the user). Option D only uses require for the owner check, but uses an if statement for the funds check, which could lead to confusion if there aren't enough funds.