Redux (pseudocode code/ reasoning explanation)

Part 1

1. The countReducer function manages a counter's state, initialised by initialState, which is likely set as { value: 0 }. It reacts to the 'increment' action by increasing the counter by 1

2.

```
if (action.type === 'decrement') {
    return {
        value: state.value - 1
    }
}
```

3.

```
if (action.type === 'reset') {
    return initialState;
}
```

Part 2

1.
   a. On line 34, the useState hook is used to create a variable named `studentsCount` and a function called `setStudentsCount.` studentCount value is set to 0 (zero) and the useState hook allows the function component in React to manage the local state.
   b. On line 39,

2.
   a.

```
FUNCTION handleAddStudent():

    // Start with zero students
    SET count to 0

    // Go through each student
    FOR each student IN students:

        // Add to count if the student is present
        IF student is present:
            ADD 1 to count

    // Set the count to the state
    UPDATE StudentsCount with count

END FUNCTION
```

   b. I would ensure the function is triggered by using React's `onclick` event handler.

```
<button onClick={handleAddStudent}>Add Student</button>
```

   c. I would update the state with the count variable.

```
setStudentsCount(count);
```

Part 3
Ran out of time.

Algorithms 1 (Coding) (Please see AlgorithmQuestions.js for code)
The loop that reverses the string runs in O(n) time, where n is the length of the input string word.

The comparison reversedString === word is also O(n) in the worst case, as it compares two strings of length n character by character.

Given that both the reversal and the comparison are sequential (not nested), the overall time complexity of the function is O(n) + O(n) = O(2n), which simplifies to O(n).

Algorithms 2 (Coding) (Please see AlgorithmQuestions.js for code)

Sorting: Unable to suggest an O value as I'm using the built-in javascript 'sort' method. Different sort methods have different complexity, if a Bubble Sort was used, the complexity could be anywhere from O(n) O(2n). Other sorting methods like Quicksort and Mergesort are far more efficient.

The 'compareNumbers' function goes through each index of the array once, again giving it an O(n) value. When the function checks for invalid inputs, it quits, so would not need to run through the whole set.

O(n) + O(n) = O(2n), which simplifies to O(n)