G LLM-generated code Example for Two Problems based on Gemini Pro 2.5 and CoT

G.1 Problem: sortasum (Low struggling level, Low diversity):

The following problem description was provided to the model as input:

```
Write a function in Java that implements the following logic: Given 2 ints, a and b,
    return their sum. However, sums in the range 10..19 inclusive, are forbidden, so
    in that case just return 20.
```

Given the prompt in Appendix A.2, the model produced the following submissions.

~~~
Submission 1:
/*Reasoning: The student calculates the sum correctly but implements the range check
    using exclusive boundaries (`> 10` and `< 19`). This approach fails to forbid
    sums that are exactly 10 or 19, which should be included in the forbidden range.*/
```java
class Solution {
    public int sortaSum(int a, int b) {
        int sum = a + b;
        if (sum > 10 && sum < 19) {
            return 20;
        }
        return sum;
    }
}
~~~

Submission 2:
/*Reasoning: The student correctly identifies the lower bound of the forbidden range
    but makes an off-by-one error on the upper bound. The condition `sum < 19`
    incorrectly allows the sum of 19 to be returned instead of 20.*/
```java
class Solution {
    public int sortaSum(int a, int b) {
        int sum = a + b;
        if (sum >= 10 && sum < 19) {
            return 20;
        } else {
            return sum;
        }
    }
}
```

Submission 3:
```
```

~~~
/*Reasoning: The student correctly identifies the forbidden range but inverts the
    logic of the conditional. The code returns the original sum for values inside the
    forbidden range and 20 for values outside of it.*/
```java
class Solution {
    public int sortaSum(int a, int b) {
        int sum = a + b;
        if (sum >= 10 && sum <= 19) {
            return sum;
        } else {
            return 20;
        }
    }
}
```

Submission 4:
/*Reasoning: The student misunderstands the meaning of a "range" and only checks the
    endpoints. This implementation correctly forbids sums of 10 and 19 but fails to
    forbid any of the numbers in between (11 through 18).*/
```java
class Solution {
    public int sortaSum(int a, int b) {
        int sum = a + b;
        if (sum == 10 || sum == 19) {
            return 20;
        }
        return sum;
    }
}
```

Submission 5:
/*Reasoning: The student correctly identifies the lower bound of the forbidden range
    but extends the upper bound one too far. The condition `sum <= 20` incorrectly
    includes 20 in the set of forbidden sums, which is not part of the requirement.*/
```java
class Solution {
    public int sortaSum(int a, int b) {
        int sum = a + b;
        if (sum >= 10 && sum <= 20) {
            return 20;
        }
        return sum;
    }
~~~

~~~
}
```
~~~

G.2 Problem: fix45 (High struggling level, High diversity):

The following problem description was provided to the model as input:

```
Given an array, return an array that contains exactly the same numbers as the
    original array, but rearranged so that every 4 is immediately followed by a 5. Do
    not move the 4's, but every other number may move. The array contains the same
    number of 4's and 5's, and every 4 has a number after it that is not a 4. In this
    version, 5's may appear anywhere in the original array.
```

Given the prompt in Appendix A.2, the model produced the following submissions.

~~~
Submission 1:
/*Reasoning: The student attempts to find all `5`s that are not preceded by a `4` to
    identify which ones are available to be moved. The check for this condition is
    flawed because it starts its loop at index 1 to avoid an out-of-bounds error,
    thus completely ignoring a `5` if it happens to be the first element of the
    array.*/
```java
import java.util.ArrayList;
import java.util.List;

class Solution {
    public int[] fix45(int[] nums) {
        List<Integer> fiveIndices = new ArrayList<>();
        List<Integer> toFillIndices = new ArrayList<>();

        for (int i = 0; i < nums.length; i++) {
            if (nums[i] == 4 && i + 1 < nums.length && nums[i + 1] != 5) {
                toFillIndices.add(i + 1);
            }
            // This check incorrectly skips index 0
            if (i > 0 && nums[i] == 5 && nums[i - 1] != 4) {
                fiveIndices.add(i);
            }
        }

        for (int i = 0; i < toFillIndices.size(); i++) {
            int temp = nums[toFillIndices.get(i)];
            nums[toFillIndices.get(i)] = 5;
            nums[fiveIndices.get(i)] = temp;
        }

~~~

~~~
        return nums;
    }
}
```

Submission 2:
/*Reasoning: The student correctly identifies when a `4` is not followed by a `5`,
    but instead of performing a proper swap, they simply overwrite the incorrect
    value with a `5`. This causes a number to be lost from the array, violating the
    problem's constraint to retain all original numbers.*/
```java
class Solution {
    public int[] fix45(int[] nums) {
        for (int i = 0; i < nums.length - 1; i++) {
            if (nums[i] == 4 && nums[i + 1] != 5) {
                for (int j = 0; j < nums.length; j++) {
                    // Find a 5 that is not already correctly placed
                    if (nums[j] == 5) {
                        if (j > 0 && nums[j - 1] == 4) {
                            continue;
                        }
                        // The original value at nums[j] is never used,
                        // and the value at nums[i+1] is overwritten.
                        nums[i + 1] = 5;
                        break;
                    }
                }
            }
        }
        return nums;
    }
}
```

Submission 3:
/*Reasoning: The student uses nested loops where the outer loop finds a `4` and the
    inner loop searches for any `5` starting from the beginning of the array. This
    approach is flawed because it may select a `5` that has already been correctly
    placed after a different `4` in a previous iteration, undoing correct work.*/
```java
class Solution {
    public int[] fix45(int[] nums) {
        for (int i = 0; i < nums.length - 1; i++) {
            if (nums[i] == 4 && nums[i + 1] != 5) {
                // This inner loop always restarts from the beginning
~~~

~~~
                for (int j = 0; j < nums.length; j++) {
                    if (nums[j] == 5) {
                        int temp = nums[i + 1];
                        nums[i + 1] = nums[j];
                        nums[j] = temp;
                        // It might have just moved a 5 that was already
                        // correctly placed after a previous 4.
                        break;
                    }
                }
            }
        }
        return nums;
    }
}
```


Submission 4:
/*Reasoning: The student gathers the indices of all `5`s into a list, but fails to
    distinguish between `5`s that are free to move and those already correctly
    placed. When a `4` needs a `5`, the code swaps its subsequent element with the
    next `5` from the list, which might be one that was already in a correct
    position.*/
```java
import java.util.ArrayList;
import java.util.List;

class Solution {
    public int[] fix45(int[] nums) {
        List<Integer> fiveLocations = new ArrayList<>();
        for (int i = 0; i < nums.length; i++) {
            if (nums[i] == 5) {
                fiveLocations.add(i);
            }
        }

        int fiveIndex = 0;
        for (int i = 0; i < nums.length - 1; i++) {
            if (nums[i] == 4 && nums[i + 1] != 5) {
                int swapPos = fiveLocations.get(fiveIndex);
                int temp = nums[i + 1];
                nums[i + 1] = 5;
                nums[swapPos] = temp;
                fiveIndex++;
            }
~~~

```
        }
        return nums;
    }
}
```
```
```