

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:

```

1925  /*Reasoning: The student correctly identifies the forbidden range but inverts the
1926      logic of the conditional. The code returns the original sum for values inside the
1927      forbidden range and 20 for values outside of it.*/
1928  ```java
1929  class Solution {
1930      public int sortaSum(int a, int b) {
1931          int sum = a + b;
1932          if (sum >= 10 && sum <= 19) {
1933              return sum;
1934          } else {
1935              return 20;
1936          }
1937      }
1938  }
1939  }
1940  ```
1941
1942
1943  Submission 4:
1944  /*Reasoning: The student misunderstands the meaning of a "range" and only checks the
1945      endpoints. This implementation correctly forbids sums of 10 and 19 but fails to
1946      forbid any of the numbers in between (11 through 18).*/
1947  ```java
1948  class Solution {
1949      public int sortaSum(int a, int b) {
1950          int sum = a + b;
1951          if (sum == 10 || sum == 19) {
1952              return 20;
1953          }
1954          return sum;
1955      }
1956  }
1957  }
1958  ```
1959
1960
1961  Submission 5:
1962  /*Reasoning: The student correctly identifies the lower bound of the forbidden range
1963      but extends the upper bound one too far. The condition `sum <= 20` incorrectly
1964      includes 20 in the set of forbidden sums, which is not part of the requirement.*/
1965  ```java
1966  class Solution {
1967      public int sortaSum(int a, int b) {
1968          int sum = a + b;
1969          if (sum >= 10 && sum <= 20) {
1970              return 20;
1971          }
1972          return sum;
1973      }
1974  }
1975  }
1976

```

```

1977 }
1978 ...

```

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.

```

1993 Submission 1:
1994 /*Reasoning: The student attempts to find all `5`s that are not preceded by a `4` to
1995    identify which ones are available to be moved. The check for this condition is
1996    flawed because it starts its loop at index 1 to avoid an out-of-bounds error,
1997    thus completely ignoring a `5` if it happens to be the first element of the
1998    array.*/
1999 ```java
2000
2001 import java.util.ArrayList;
2002 import java.util.List;
2003
2004 class Solution {
2005     public int[] fix45(int[] nums) {
2006         List<Integer> fiveIndices = new ArrayList<>();
2007         List<Integer> toFillIndices = new ArrayList<>();
2008
2009         for (int i = 0; i < nums.length; i++) {
2010             if (nums[i] == 4 && i + 1 < nums.length && nums[i + 1] != 5) {
2011                 toFillIndices.add(i + 1);
2012             }
2013             // This check incorrectly skips index 0
2014             if (i > 0 && nums[i] == 5 && nums[i - 1] != 4) {
2015                 fiveIndices.add(i);
2016             }
2017         }
2018
2019         for (int i = 0; i < toFillIndices.size(); i++) {
2020             int temp = nums[toFillIndices.get(i)];
2021             nums[toFillIndices.get(i)] = 5;
2022             nums[fiveIndices.get(i)] = temp;
2023         }
2024     }
2025 }

```

```

2029         return nums;
2030     }
2031 }
2032 ...
2033
2034
2035 Submission 2:
2036 /*Reasoning: The student correctly identifies when a `4` is not followed by a `5`,
2037    but instead of performing a proper swap, they simply overwrite the incorrect
2038    value with a `5`. This causes a number to be lost from the array, violating the
2039    problem's constraint to retain all original numbers.*/
2040 ```java
2041 class Solution {
2042     public int[] fix45(int[] nums) {
2043         for (int i = 0; i < nums.length - 1; i++) {
2044             if (nums[i] == 4 && nums[i + 1] != 5) {
2045                 for (int j = 0; j < nums.length; j++) {
2046                     // Find a 5 that is not already correctly placed
2047                     if (nums[j] == 5) {
2048                         if (j > 0 && nums[j - 1] == 4) {
2049                             continue;
2050                         }
2051                         // The original value at nums[j] is never used,
2052                         // and the value at nums[i+1] is overwritten.
2053                         nums[i + 1] = 5;
2054                         break;
2055                     }
2056                 }
2057             }
2058         }
2059         return nums;
2060     }
2061 }
2062 ...
2063
2064 Submission 3:
2065 /*Reasoning: The student uses nested loops where the outer loop finds a `4` and the
2066    inner loop searches for any `5` starting from the beginning of the array. This
2067    approach is flawed because it may select a `5` that has already been correctly
2068    placed after a different `4` in a previous iteration, undoing correct work.*/
2069 ```java
2070 class Solution {
2071     public int[] fix45(int[] nums) {
2072         for (int i = 0; i < nums.length - 1; i++) {
2073             if (nums[i] == 4 && nums[i + 1] != 5) {
2074                 // This inner loop always restarts from the beginning

```

```

2081         for (int j = 0; j < nums.length; j++) {
2082             if (nums[j] == 5) {
2083                 int temp = nums[i + 1];
2084                 nums[i + 1] = nums[j];
2085                 nums[j] = temp;
2086                 // It might have just moved a 5 that was already
2087                 // correctly placed after a previous 4.
2088                 break;
2089             }
2090         }
2091     }
2092 }
2093 }
2094 }
2095 return nums;
2096 }
2097 }
2098 ...
2099
2100 Submission 4:
2101 /*Reasoning: The student gathers the indices of all `5`s into a list, but fails to
2102    distinguish between `5`s that are free to move and those already correctly
2103    placed. When a `4` needs a `5`, the code swaps its subsequent element with the
2104    next `5` from the list, which might be one that was already in a correct
2105    position.*/
2106 ```java
2107 import java.util.ArrayList;
2108 import java.util.List;
2109
2110 class Solution {
2111     public int[] fix45(int[] nums) {
2112         List<Integer> fiveLocations = new ArrayList<>();
2113         for (int i = 0; i < nums.length; i++) {
2114             if (nums[i] == 5) {
2115                 fiveLocations.add(i);
2116             }
2117         }
2118
2119         int fiveIndex = 0;
2120         for (int i = 0; i < nums.length - 1; i++) {
2121             if (nums[i] == 4 && nums[i + 1] != 5) {
2122                 int swapPos = fiveLocations.get(fiveIndex);
2123                 int temp = nums[i + 1];
2124                 nums[i + 1] = 5;
2125                 nums[swapPos] = temp;
2126                 fiveIndex++;
2127             }
2128         }
2129     }
2130 }
2131
2132

```

```
2133     }  
2134     return nums;  
2135 }  
2136 }  
2137 }  
2138 ...
```

2139

2140

2141

2142

2143

2144

2145

2146

2147

2148

2149

2150

2151

2152

2153

2154

2155

2156

2157

2158

2159

2160

2161

2162

2163

2164

2165

2166

2167

2168

2169

2170

2171

2172

2173

2174

2175

2176

2177

2178

2179

2180

2181

2182

2183

2184