CS 520 Homework 3 - Aryan Tipnis

Specification-Guided Test Improvement

Github Link: https://github.com/aryantipnis/CS520_HW3

Part 1 - Generate, Evaluate and Refine Specifications:

For this part, we analyze two problems with the weakest test coverage from Exercise 2 and generate formal specifications to better understand their expected behavior. I chose Problem 3 and 9. Then we provide each problem's method signature and natural language description to an LLM to produce specification-style assertions that capture the logical relationship between inputs and outputs without calling the method or using side effects. We evaluate each generated assertion for correctness, calculate the overall accuracy, and manually fix any incorrect ones. The resulting refined specifications form the basis for improving test cases in Part 2.

Problem 3:

- Natural Language Problem Description:

```
Given a positive integer N, return the total sum of its digits in binary.
Example:
For N = 1000, the sum of digits will be 1 the output should be "1".

For N = 150, the sum of digits will be 6 the output should be "110".

For N = 147, the sum of digits will be 12 the output should be "1100".

Variables: @N integer
Constraints: 0 ≤ N ≤ 10000.
Output: a string of binary numbers.
```

- Function Signature:

```
def digitsSuminBinary(N: int) -> str
```

- LLM Prompt used:

  Problem description: <insert here>
  Method signature: <insert here>

  Please generate formal specifications written as assertion statements that describe the correct behavior of this method.

Requirements:
- Let 'res' denote the expected return value of the method.
- DO NOT call the method itself in the assertions (no self-reference).
- DO NOT use any side-effecting operations, including modifying data structures, I/O, randomness, or timing.
- Use only pure logic, arithmetic, boolean expressions, set membership, and comparisons.
- Provide around 5 distinct assertions unless the method allows fewer.

- LLM Generated Assertions (Before Correction):

```
S = sum(int(d) for d in str(N))

assert all(c in ('0', '1') for c in res)
assert int(res, 2) == S
assert (S == 0) == (res == "0")
assert (S > 0) == (not res.startswith("0"))
assert len(res) == (1 if S == 0 else S.bit_length())
```

- Table for LLM-Generated Assertions Correction:

S = sum(int(d) for d in str(N))

#	Assertion	Correct?	Notes
1	assert all(c in ('0','1') for c in res)	Correct	Ensures res is a valid binary string.
2	assert int(res, 2) == S	Correct	Properly checks binary–decimal equivalence using pure operations.
3	assert (S == 0) == (res == "0")	Correct	Exactly captures the zero case.
4	assert (S > 0) == (not res.startswith("0"))	Correct	Valid binary strings >0 must not have leading zeros.
5	assert len(res) == (1 if S == 0 else (S.bit_length()))	Correct	Binary length matches necessary bit-length.

- Accuracy Rate: Accuracy = 5/5 = 100%

- Final Corrected Set:

```
S = sum(int(d) for d in str(N))

assert all(c in ('0', '1') for c in res)
assert int(res, 2) == S
assert (S == 0) == (res == "0")
assert (S > 0) == (not res.startswith("0"))
assert len(res) == (1 if S == 0 else S.bit_length())
```

Problem 9:

- Natural Language Problem Description:

```
Given a, t and r, write a function to find the t-nth term of geometric series.
```

- Function Signature:

```
def findNthTerm(a: int, t: int, r: int) -> int
```

- LLM Prompt used:

  Problem description: <insert here>
  Method signature: <insert here>

  Please generate formal specifications written as assertion statements that describe the correct behavior of this method.

  Requirements:
  - Let 'res' denote the expected return value of the method.
  - DO NOT call the method itself in the assertions (no self-reference).
  - DO NOT use any side-effecting operations, including modifying data structures, I/O, randomness, or timing.
  - Use only pure logic, arithmetic, boolean expressions, set membership, and comparisons.
  - Provide around 5 distinct assertions unless the method allows fewer.

- LLM Generated Assertions (Before Correction):

```
assert isinstance(t, int) and t >= 1
assert res == a * (r ** (t - 1))
assert (t == 1) == (res == a)
assert (r == 1) == (res == a)
assert (a == 0) == (res == 0)
```

- Table for LLM-Generated Assertions Correction:

#	**Assertion**	**Correct?**	**Notes**
1	assert isinstance(t, int) and t >= 1	Correct	Matches preconditions in code exactly.
2	assert res == a * (r ** (t - 1))	Correct	This *is* the geometric term formula.
3	assert (t == 1) == (res == a)	Correct	If `t=1`, exponent is 0 so the term = a.
4	assert (r == 1) == (res == a)	Correct	With ratio 1, series becomes constant = a
5	assert (a == 0) == (res == 0)	Correct	Zero first term makes every term zero.

- Accuracy Rate: Accuracy = 5/5 = 100%

- Final Corrected Set:

```
assert isinstance(t, int) and t >= 1
assert res == a * (r ** (t - 1))
assert (t == 1) == (res == a)
assert (r == 1) == (res == a)
assert (a == 0) == (res == 0)
```

Part 2 - Use Specifications To Guide Test Improvements:

In this part, we take our corrected specifications and use them to generate new test cases. These “spec-guided” tests are added to our Exercise 2 test suite. Then we run both the original tests and the new spec-guided tests and compare their statement coverage, branch coverage, and any new failures or insights.

Problem 3:

- LLM Prompt to generate test cases:

Using the following corrected specifications, generate Python test cases that thoroughly reflect these logical properties.
Label each test clearly as "spec-guided".
Do not call the specifications inside the tests; the tests should call only the target function.

Corrected specifications:
<PASTED FINAL ASSERTION SET HERE>

- Resulting Test cases:

```
# ----------- SPEC-GUIDED TESTS FOR PROBLEM 3 -----------
def test_spec_guided_zero_sum():
   # S == 0 → result must be exactly "0"
   assert solution1(0) == "0"             # sum = 0
   assert solution1(1000) == "1"          # sum = 1, ensure not "0"

def test_spec_guided_valid_binary_string():
   res = solution1(147)    # sum = 12 -> binary "1100"
   assert all(c in ('0', '1') for c in res)

def test_spec_guided_binary_value_matches_sum():
   N = 150
   S = 1 + 5 + 0
   res = solution1(N)
   assert int(res, 2) == S

def test_spec_guided_no_leading_zeros():
   N = 999
   res = solution1(N)
   assert not res.startswith("0")

def test_spec_guided_length_matches_bit_length():
   N = 147    # sum = 12 → bit_length = 4
   res = solution1(N)
   assert len(res) == 4
```

- Before/After Coverage Table:

#	Old Line Coverage	Old Branch Coverage	New Line Coverage	New Branch Coverage
Question 3	71%	62%	83%	75%

- Case-specific Insights:

The line and branch coverage increased after adding the specification-guided tests because the new tests were directly derived from the logical assertions about the function's behavior rather than from example-based intuition. The new tests included a check for N==0 which explored an additional branch and increased both line and branch coverage. However, coverage did not reach 100% because certain branches like specifically the type-checking (not isinstance(N, int)) and negative-value branch (N < 0)—were not triggered by the specification-guided tests. The refined specifications focused only on valid input behavior rather than error handling, leaving those exception-raising lines uncovered. Thus, the improvement reflects deeper behavioral testing but does not include explicit invalid-input scenarios, preventing full coverage.

Problem 9:

- LLM Prompt to generate test cases:

Using the following corrected specifications, generate Python test cases that thoroughly reflect these logical properties.
Label each test clearly as "spec-guided".
Do not call the specifications inside the tests; the tests should call only the target function.

Corrected specifications:
<PASTED FINAL ASSERTION SET HERE>

- Resulting Test cases:

```
# ----------- SPEC-GUIDED TESTS FOR PROBLEM 9 -----------


def test_spec_guided_first_term():
   # t = 1 → result = a
```

```python
    assert solution1(5, 1, 3) == 5
    assert solution1(-2, 1, 10) == -2

def test_spec_guided_ratio_one():
    # r = 1 → all terms equal a
    assert solution1(6, 5, 1) == 6

def test_spec_guided_zero_first_term():
    # a = 0 → every term must be 0
    assert solution1(0, 10, 100) == 0

def test_spec_guided_general_case():
    # Standard geometric term
    assert solution1(3, 4, 2) == 3 * (2 ** 3)

def test_spec_guided_type_and_range_validation():
    # non-integer t → TypeError
    try:
        solution1(5, 2.5, 3)
        assert False
    except TypeError:
        assert True

    # t < 1 → ValueError
    try:
        solution1(5, 0, 3)
        assert False
    except ValueError:
        assert True
```

- Before/After Coverage Table:

#	Old Line Coverage	Old Branch Coverage	New Line Coverage	New Branch Coverage
Question 9	64%	50%	100%	100%

- Case-specific Insights:

The line and branch coverage increased because the specification-guided tests explicitly targeted behaviors that the baseline tests never checked, such as the special cases where $t = 1$, $r = 1$ or $a = 0$, along with the branches for invalid inputs (t not an integer and $t < 1$). These tests exercised all remaining branches in the function, including the exception-raising paths, which were previously untested. As a result, every line and decision point in the function was executed at least once, bringing both line and branch coverage up to 100%.