

520: Exercise 2

Sahil Kamath

[Github](#)

For the third exercise, I will be using the same 2 problems from the APPS dataset, the Sparrow Grains and the Valid brackets problems, with the added edge case clause checks that were added as a part of exercise 2 for coverage detection.

1) Sparrow Problem

Prompt used :

You are a formal verification expert.

Problem Description:

[problem statement taken from the APPS dataset and pasted in prompt]

Method Signature:

def solve_sparrow(n: int, m: int) -> int:

Please write formal specifications as Python assertions that describe the correct behavior of this method.

1. Let 'res' denote the expected return value of `solve_sparrow(n, m)`.
2. Do NOT call the function `solve_sparrow()` itself in the assertions (no self-reference).
3. Do NOT use methods with side effects (`print`, `read`, `write`).
4. Generate about 5 unique specifications.
5. Express the relationship between '`n`', '`m`', and '`res`' using pure arithmetic and boolean logic.
6. Consider edge cases implied by the logic (e.g. $n \leq m$).

Generated Assertions (Before Correction): The LLM generated 5 specifications. It correctly identified basic properties but failed on the complex logic regarding the barn's capacity limit.

- **Spec 1:** assert `res >= 1` (Correct)
- **Spec 2:** assert `not (m >= 1) or res >= 1` (Correct)
- **Spec 3:** assert `(n + m * res - sum(range(1, res + 1))) <= 0` (Incorrect - assumes infinite barn capacity)
- **Spec 4:** assert `n + m * (res - 1) - sum(range(1, res)) > 0` (Incorrect - assumes infinite barn capacity)

- **Spec 5:** assert n + m * res - (res * (res + 1)) // 2 <= 0 (Incorrect - same fallacy)

Accuracy Rate:

- Total Assertions: 5
- Correct: 2
- Incorrect: 3
- Accuracy: 40%

Incorrect Assertion	Issue	Corrected Assertion (Refined)
$n + m * res - sum(...) <= 0$	Infinite Barn Fallacy: The LLM assumed grains are added indefinitely ($n + m * res$). It ignored the constraint that the barn is capped at n .	Two-Phase Logic: if $n > m$: $k = res - m$; assert $(k * (k + 1)) // 2 \geq (n - m)$
$n + m * (res-1) ... > 0$	Infinite Barn Fallacy: Same as above.	Two-Phase Logic: if $n > m$: $k = res - m$; assert $((k - 1) * k) // 2 < (n - m)$
(Missing Specification)	Missing Error Handling: The natural language prompt did not describe invalid inputs, so the LLM missed the ValueError guard clauses I added in Exercise 2.	if $m < 1$: try: solve_sparrow(n, m); assert False except ValueError: assert True
(Missing Specification)	Missing Error Handling: Missed negative input check.	if $n < 0$: try: solve_sparrow(n, m); assert False except ValueError: assert True
(Missing)	Missing Type Safety: The LLM missed the Python specific type-checking.	if not isinstance(n, int): assert raises(TypeError)

Part 2:

Prompt used for LLM:

I have a Python function `solve_sparrow(n, m)` and a list of refined, corrected formal specifications (assertions) that describe its behavior.

Please generate a set of `pytest` test cases that verify these specifications are met.

1. Label these tests as "spec_guided".
2. Ensure you cover the edge cases implied by the math (e.g., $n=m$, $n=0$).
3. Use `pytest.raises` for the error cases.

Here are the Corrected Specifications:

```
# Specification 1: The "Always Full" Case
if n <= m and n >= 0 and m >= 1:
    assert res == n

# Specification 2: The "Two-Phase" Case (Corrected Logic)
if n > m and n >= 0 and m >= 1:
    # Phase 1: Remaining grain after m days
    remaining = n - m
    # Phase 2: 'k' additional days
    k = res - m

    # Assertion: The total sparrows over 'k' days must consume at least 'remaining'
    sparrow_consumption = (k * (k + 1)) // 2
    assert sparrow_consumption >= remaining

    # Assertion: The day before (k-1) was NOT enough
    prev_consumption = ((k - 1) * k) // 2
    assert prev_consumption < remaining

# Specification 3: Error Handling (Defensive Coding)
if m < 1:
    # Assertion: Must raise ValueError
    try:
        solve_sparrow(n, m)
    assert False
except ValueError:
    assert True

if n < 0:
```

```
# Assertion: Must raise ValueError
```

```
try:
```

```
    solve_sparrow(n, m)
```

```
    assert False
```

```
except ValueError:
```

```
    assert True
```

Spec guided tests:

```
# Specification 1: The "Always Full" Case
```

```
def test_spec_guided_always_full_case_1():
```

```
    assert solve_sparrow(0, 5) == 0
```

```
def test_spec_guided_always_full_case_2():
```

```
    assert solve_sparrow(3, 5) == 3
```

```
def test_spec_guided_always_full_case_3():
```

```
    assert solve_sparrow(5, 5) == 5
```

```
# Specification 2: The "Two-Phase" Case (Corrected Logic)
```

```
def test_spec_guided_two_phase_case_1():
```

```
    # n=6, m=5 -> remaining=1, sparrow_consumption=1 when k=1 => res=6
```

```
    assert solve_sparrow(6, 5) == 1 + 5 # n=6, m=5 => remaining=1, k=1, res=6
```

```
def test_spec_guided_two_phase_case_2():
```

```
    # n=15, m=5 -> remaining=10
```

```
    # k=4 -> (4*5)/2 = 10 => res=9
```

```
    assert solve_sparrow(15, 5) == 9
```

```
def test_spec_guided_two_phase_case_3():
```

```
    # n=16, m=5 -> remaining=11
```

```
    # k=4 -> (4*5)/2 = 10 < 11, k=5 -> (5*6)/2 = 15 >= 11 => res=10
```

```
    assert solve_sparrow(16, 5) == 10
```

```

# Specification 3: Error Handling (Defensive Coding)

def test_spec_guided_error_handling_m_less_1():

    with pytest.raises(ValueError):
        solve_sparrow(5, 0)

    with pytest.raises(ValueError):
        solve_sparrow(5, -1)

def test_spec_guided_error_handling_n_less_0():

    with pytest.raises(ValueError):
        solve_sparrow(-5, 3)

def test_spec_guided_error_handling_invalid_types():

    with pytest.raises(TypeError):
        solve_sparrow("5", 5)

    with pytest.raises(TypeError):
        solve_sparrow(5, "5")

```

Baseline coverage from exercise 2 part 2:

```

PS C:\Users\Hp\Desktop\College\MS\520\ex2\Part2\p1> python -m pytest --cov=solution --cov-branch test_solutions.py
===== test session starts =====
platform win32 -- Python 3.13.1, pytest-8.4.2, pluggy-1.5.0
rootdir: C:\Users\Hp\Desktop\College\MS\520\ex2\Part2\p1
plugins: anyio-4.8.0, devtools-0.12.2, logfire-2.8.0, base-url-2.1.0, cov-7.0.0, playwright-0.6.2, typeguard-4.4.2
collected 1 item

test_solutions.py .

===== tests coverage =====
===== coverage: platform win32, python 3.13.1-final-0 =====

      Name     Stmts   Miss Branch BrPart Cover
solution.py      19      3     12      3    81%
-----TOTAL----- 19      3     12      3    81%
===== 1 passed in 0.10s =====

```

New test cases on code from ex 2:

```

rootdir: C:\Users\Hp\Desktop\College\MS\520\ex3\SparrowProblem
plugins: anyio-4.8.0, devtools-0.12.2, logfire-2.8.0, base-url-2.1.0, cov-7.0.0, playwright-0.6.2, typeguard-4.4.2
collected 9 items

test_spec_guided.py ..... [100%]

===== tests coverage =====
coverage: platform win32, python 3.13.1-final-0

Name     Stmts Miss Branch BrPart Cover
-----
solution.py    19     0    12     0  100%
-----
TOTAL        19     0    12     0  100%
===== 9 passed in 0.11s =====

```

Problem	Old Stmt % (Ex 2 Baseline)	New Stmt % (Spec-Guided)	Old Branch %	New Branch %
APPS/Sparrow	84%	100%	75%	100%

Case-Specific Insight: My coverage improved to 100% because of manual refinement. The LLM's initial specifications focused purely on the mathematical problem description and missed the implementation specific guard clauses I added in Exercise 2 (specifically the ValueError and TypeError checks). By manually adding formal specifications for these "robustness" constraints, the resulting spec-guided test suite was able to cover the error-handling paths that would have otherwise been missed.

2) Bracket problem

Part 1:

Prompt used :

You are a formal verification expert.

Problem Description:

[problem statement taken from the APPS dataset and pasted in prompt]

Method Signature:

```
def solve_bracket(cnt1: int, cnt2: int, cnt3: int, cnt4: int) -> int:
```

Please write formal specifications as Python assertions that describe the correct behavior of this method.

1. Let 'res' denote the expected return value (1 for True, 0 for False).
2. Do NOT call the function `solve_bracket()` itself in the assertions.
3. Do NOT use methods with side effects.
4. Generate about 5 unique specifications.
5. Express the relationship between the inputs (`cnt1..cnt4`) and output (`res`) using arithmetic and boolean logic.
6. Consider the balance of open vs. closed brackets and the order of traversal.

Incorrect Assertion	Issue	Corrected Assertion (Refined)
<code>(cnt2 == 1 ...) == (res == 1)</code>	Too Specific: This is a single test case, not a general specification. It fails to define the behavior for all other inputs.	Total Balance: if <code>cnt1 != cnt4</code> : assert <code>res == 0</code>
<code>(cnt1 == 0 ... cnt4 == 2) == (res == 0)</code>	Too Specific: Another hardcoded example. It misses the general rule about why it failed (imbalance).	Total Balance: if <code>cnt1 == cnt4</code> and <code>(cnt1 > 0 or cnt3 == 0)</code> : assert <code>res == 1</code>
<code>(cnt3 == 2 ... cnt1 == 0) == (res == 0)</code>	Too Specific: While logically true for this specific case, it fails to capture the general "Traversal Constraint."	Traversal Check: if <code>cnt1 == 0</code> and <code>cnt3 != 0</code> : assert <code>res == 0</code>
(Missing)	Missing Error Handling: The LLM missed the defensive checks we added.	<code>if any(c < 0 ...): try: solve_bracket(...) assert False except ValueError: assert True</code>

Incorrect Assertion	Issue	Corrected Assertion (Refined)
(Missing)	Missing Type Safety: The LLM missed the type checks.	if not all(isinstance(c, int)...): try: solve_bracket(...) assert False except TypeError: assert True

Generated Assertions (Before Correction): The LLM struggled to find the general rules. Instead of logical specifications, it generated specific hardcoded examples or insufficient balance checks.

- **Spec 1:** assert (cnt2==1 and cnt1==0...) == (res==1) (Incorrect - Too specific/hardcoded example)
- **Spec 2:** assert (cnt1==0 ... cnt4==2) == (res==0) (Incorrect - Too specific)
- **Spec 3:** assert (total_open == total_closed) == (res == 1) (Incorrect - Necessary but not sufficient; ignores traversal order)
- **Spec 4:** assert res == 0 or res == 1 (Correct but trivial)
- **Spec 5:** assert cnt1 >= 0 (Correct constraint, but not a behavioral spec)

Accuracy Rate:

- **Total Assertions:** 5
- **Correct:** 2 (Trivial ones)
- **Incorrect:** 3
- **Accuracy:** 40%

Part 2:

LLM Prompt used to generate test cases

I have a Python function `solve_bracket(cnt1, cnt2, cnt3, cnt4)` and a list of refined, corrected formal specifications (assertions) that describe its behavior.

Please generate a set of `pytest` test cases that verify these specifications are met.

1. Label these tests as "spec_guided".
2. Ensure you cover edge cases implied by the math (e.g., $cnt1=0$).
3. Use `pytest.raises` for the error cases.

Here are the Corrected Specifications:

Specification 1: Total Balance Rule

The number of "(" strings ($cnt1$) MUST equal the number of ")" strings ($cnt4$).

if $cnt1 \neq cnt4$:

 assert $res == 0$

Specification 2: Traversal / Buffer Rule

If we have no "(" strings ($cnt1 == 0$), we cannot start a sequence with ")" ($cnt3$).

if $cnt1 == 0$ and $cnt3 \neq 0$:

 assert $res == 0$

Specification 3: Valid Case

If balance is met ($cnt1 == cnt4$) AND we respect the traversal rule, it is valid.

if $cnt1 == cnt4$ and ($cnt1 > 0$ or $cnt3 == 0$):

 assert $res == 1$

Specification 4: Error Handling (Negative Inputs)

if any($c < 0$ for c in [$cnt1, cnt2, cnt3, cnt4$]):

 # Assertion: Must raise ValueError

 try:

 solve_bracket($cnt1, cnt2, cnt3, cnt4$)

 assert False

```

except ValueError:
    assert True

# Specification 5: Error Handling (Type Checking)
if not all(isinstance(c, int) for c in [cnt1, cnt2, cnt3, cnt4]):
    # Assertion: Must raise TypeError
    try:
        solve_bracket(cnt1, cnt2, cnt3, cnt4)
        assert False
    except TypeError:
        assert True

```

Spec guided tests:

```

import pytest

# Import your solve_bracket function
from your_module import solve_bracket # Replace with actual import path

# Labeling the tests as 'spec_guided'
pytestmark = pytest.mark.spec_guided

# -----
# Specification 1: Total Balance Rule (cnt1 != cnt4 => res == 0)
# -----
def test_spec1_balance_mismatch_1():
    assert solve_bracket(2, 0, 0, 1) == 0

def test_spec1_balance_mismatch_2():
    assert solve_bracket(1, 0, 0, 0) == 0

# -----
# Specification 2: Traversal/Buffer Rule (cnt1 == 0 and cnt3 != 0 => res == 0)
# -----
def test_spec2_traversal_rule_1():
    assert solve_bracket(0, 0, 2, 0) == 0

```

```
def test_spec2_traversal_rule_2():

    assert solve_bracket(0, 0, 5, 0) == 0


# ----

# Specification 3: Valid Case (cnt1 == cnt4 and valid traversal => res == 1)

# ----

def test_spec3_valid_case_1():

    assert solve_bracket(1, 0, 0, 1) == 1


def test_spec3_valid_case_2():

    assert solve_bracket(3, 0, 0, 3) == 1


def test_spec3_edge_case_zero():

    assert solve_bracket(0, 0, 0, 0) == 1 # Valid by spec


def test_spec3_with_cnt2():

    assert solve_bracket(2, 5, 0, 2) == 1


# ----

# Specification 4: Error Handling - Negative Inputs

# ----

def test_spec4_negative_input_1():

    with pytest.raises(ValueError):

        solve_bracket(-1, 0, 0, 0)


def test_spec4_negative_input_2():

    with pytest.raises(ValueError):

        solve_bracket(0, -2, 0, 0)


def test_spec4_negative_input_4():

    with pytest.raises(ValueError):

        solve_bracket(0, 0, 0, -4)


# -----
```

```

# Specification 5: Error Handling - Type Checking

# ----

def test_spec5_type_check_string():
    with pytest.raises(TypeError):
        solve_bracket(0, 'abc', 0, 0)

def test_spec5_type_check_float():
    with pytest.raises(TypeError):
        solve_bracket(0, 3.5, 0, 0)

def test_spec5_multiple_invalid_types():
    with pytest.raises(TypeError): # Should still fail on one of them
        solve_bracket('x', [1], 0, 10)

# ----

# Additional Edge Cases for Coverage
# ----

def test_edge_case_all_zero():
    assert solve_bracket(0, 0, 0, 0) == 1

def test_edge_case_large_inputs():
    assert solve_bracket(100, 200, 0, 100) == 1

def test_edge_case_cnt2_only():
    assert solve_bracket(0, 5, 0, 0) == 1 # Only cnt2 used

def test_edge_case_cnt1_zero_and_cnt3_zero():
    assert solve_bracket(0, 0, 0, 0) == 1

```

Baseline coverage from exercise 2 part 2:

```
test_solutions.py .

D ===== tests coverage =====
----- coverage: platform win32, python 3.13.1-final-0 ----

Name     Stmts Miss Branch BrPart Cover
-----
solution.py    11     2     8     2   79%
-----
TOTAL        11     2     8     2   79%
===== 1 passed in 0.11s =====
```

New test cases on code from ex 2:

```
-- Docs: https://docs.pytest.org/en/stable/how-to/capture-warnings.html
===== tests coverage =====
----- coverage: platform win32, python 3.13.1-final-0 ----

Name     Stmts Miss Branch BrPart Cover
-----
solution.py    11     0     8     0  100%
-----
TOTAL        11     0     8     0  100%
===== 18 passed, 1 warning in 0.14s =====
PS C:\Users\hp\Desktop\College\MS\520\ex3\BracketProblem>
```

Problem	Old Stmt % (Ex 2 Baseline)	New Stmt % (Spec-Guided)	Old Branch %	New Branch %
APPS/Bracket	86%	100%	83%	100%

Case Specific Insight

Insight: The coverage improvement from 86% to 100% highlights the limitations of relying solely on natural language for test generation. The LLM's initial attempt produced specific examples rather than general properties, effectively mimicking the "hardcoded" nature of the original benchmark tests. By manually refining these into general logical rules (e.g., "Total Balance" and "Traversal Check") and explicitly adding specifications for implementation-level constraints (Input Validation), the resulting spec-guided tests were able to cover the edge cases and error paths that the LLM originally ignored. This confirms that formal specifications are most powerful when they combine high-level logic with implementation-specific robustness requirements.