

# Assignment 3

## Exercise 1

### Problem 0

For the following natural language problem description:

```
def minPath(grid, k):  
    """
```

*Given a grid with  $N$  rows and  $N$  columns ( $N \geq 2$ ) and a positive integer  $k$ , each cell of the grid contains a value. Every integer in the range  $[1, N * N]$  inclusive appears exactly once on the cells of the grid.*

*You have to find the minimum path of length  $k$  in the grid. You can start from any cell, and in each step you can move to any of the neighbor cells, in other words, you can go to cells which share an edge with your current cell.*

*Please note that a path of length  $k$  means visiting exactly  $k$  cells (not necessarily distinct).*

*You CANNOT go off the grid.*

*A path  $A$  (of length  $k$ ) is considered Less than a path  $B$  (of length  $k$ ) if after making the ordered lists of the values on the cells that  $A$  and  $B$  go through (let's call them  $Lst_A$  and  $Lst_B$ ),  $Lst_A$  is lexicographically less than  $Lst_B$ , in other words, there exist an integer index  $i$  ( $1 \leq i \leq k$ ) such that  $Lst_A[i] < Lst_B[i]$  and for any  $j$  ( $1 \leq j < i$ ) we have  $Lst_A[j] = Lst_B[j]$ .*

*It is guaranteed that the answer is unique.*

*Return an ordered list of the values on the cells that the minimum path goes through.*

*Examples:*

```
Input: grid = [ [1,2,3], [4,5,6], [7,8,9]], k = 3  
Output: [1, 2, 1]
```

```
Input: grid = [ [5,9,3], [4,1,6], [7,8,2]], k = 1  
Output: [1]
```

"""

With the method signature 'list of ints minPath(lists of lists matrix, int)'  
Please write at least 5 unique formal specifications as Python assertions that describe the correct behavior of this method.

Let 'res' denote the expected return value of the given method.

Do not call the method in your assertions.

Do not use methods with side effects such as System.out.println, file I/O, random number generation, or timing functions.

Express the relationship between 'n' and 'res' using pure arithmetic and boolean logic only

Listing 1: Prompt for specifications of problem 0

Assertions 1 to 3 are correct assertions. Assertions 4 and 5 seem unnecessarily complex, but after careful consideration seem valid.

This yields an accuracy rate of 100%.

```
### output for problem_0
Here are 5 unique formal specifications as Python assertions for the `minPath` method:

Let `N` be the dimension of the grid, derived as `N = len(grid)`.

1. **The length of the returned path `res` must be exactly `k`**  

```python
assert len(res) == k
```

2. **All values in the path `res` must be valid values from the grid, i.e., integers in the range `[1, N*N]` inclusive.**  

```python
assert all(1 <= val <= N*N for val in res)
```

3. **The first element of the path `res[0]` must be `1`**. This is because the problem states that every integer in `[1, N*N]` appears exactly once in the grid, meaning `1` is always the minimum value available. To achieve a lexicographically smallest path, the first element must be the smallest possible value.  

```python
assert res[0] == 1
```

4. **For any two consecutive values `res[i]` and `res[i+1]` in the path, the cells containing these values in the grid must be adjacent (share an edge).**  

```python
assert all(
    any( # Check if there exist coordinates (r1, c1) for res[i]
        any( # Check if there exist coordinates (r2, c2) for res[i+1]
            grid[r1][c1] == res[i] and grid[r2][c2] == res[i+1] and (
                abs(r1 - r2) + abs(c1 - c2) == 1
            )
            for r2 in range(N) for c2 in range(N)
        )
        for r1 in range(N) for c1 in range(N)
    )
    for i in range(k - 1)
)
```

5. **If `k` is at least 2, the second element of the path `res[1]` must be the smallest value among all direct neighbors of the cell containing `1` in the grid.**  

This ensures the lexicographical minimality of the path's second element after the initial `1`.
```python
assert (k < 2) or (
    res[1] == min(
        grid[r_neigh][c_neigh]
        for r_1 in range(N) for c_1 in range(N) if grid[r_1][c_1] == 1
        for dr, dc in [(-1, 0), (1, 0), (0, -1), (0, 1)] # Possible neighbor offsets
    )
)
```
```

```

        for r_neigh, c_neigh in [(r_1 + dr, c_1 + dc)]
            if 0 <= r_neigh < N and 0 <= c_neigh < N # Check bounds for neighbor
        )
    )
```

```

Listing 2: Generated formal specifications for problem 0

# assertion	Correct	Issue
Assertion 1	Yes	-
Assertion 2	Yes	-
Assertion 3	Yes	-
Assertion 4	Yes	unnecessarily complex in my opinion, no logical errors
Assertion 5	Yes	unnecessarily complex in my opinion, no logical errors

Table 1: Accuracy of generated assertions for problem 0

## Problem 1

For the following natural language problem description:

```
def numerical_letter_grade(grades):
    """It is the last week of the semester and the teacher has to give the grades
    to students. The teacher has been making her own algorithm for grading.
    The only problem is, she has lost the code she used for grading.
    She has given you a list of GPAs for some students and you have to write
    a function that can output a list of letter grades using the following table:
        GPA           | Letter grade
        4.0           | A+
        > 3.7         | A
        > 3.3         | A-
        > 3.0         | B+
        > 2.7         | B
        > 2.3         | B-
        > 2.0         | C+
        > 1.7         | C
        > 1.3         | C-
        > 1.0         | D+
        > 0.7         | D
        > 0.0         | D-
        0.0           | E
```

Example:

```
grade_equation([4.0, 3, 1.7, 2, 3.5]) ==> ['A+', 'B', 'C-', 'C', 'A-']
"""
With the method signature 'list of strings numerical_letter_grade(list of float)'
Please write least 5 unique formal specifications as Python assertions that describe the
correct behavior of this
method.
```

Let 'res' denote the expected return value of the given method.  
Do not call the method in your assertions.  
Do not use methods with side effects such as System.out.println, file I/O, random number  
generation, or timing functions.  
Express the relationship between 'n' and 'res' using pure arithmetic and boolean logic  
only

Listing 3: Prompt for specifications of problem 1

For this problem all the assertions generated by the AI model are correct. They are also a lot easier to understand compared to the last assertions for problem 0.

```
### output for problem_1
Here are 5 unique formal specifications as Python assertions for the
`numerical_letter_grade` method, where `n` is the input list of floats and `res` is
the expected return value (list of strings):

1. **Assertion 1: Output list length and element types.**
   The output list `res` must have the same length as the input list `n`, and every
   element in `res` must be a string.
   ```python
   assert len(res) == len(n) and all(isinstance(grade, str) for grade in res)
```

```

```
2. **Assertion 2: Specific mapping for the highest exact GPA.**
   If any GPA in the input list `n` is exactly 4.0, its corresponding grade in
   `res` must be 'A+'.
   ```python
   assert all(res[i] == 'A+' for i in range(len(n)) if n[i] == 4.0)
   ```

3. **Assertion 3: Specific mapping for a typical mid-range GPA interval.**
   If a GPA is strictly greater than 2.7 and less than or equal to 3.0, its
   corresponding letter grade must be 'B'. This demonstrates correct handling of open
   and closed bounds.
   ```python
   assert all(res[i] == 'B' for i in range(len(n)) if 2.7 < n[i] <= 3.0)
   ```

4. **Assertion 4: Specific mapping for the lowest exact GPA.**
   If any GPA in the input list `n` is exactly 0.0, its corresponding grade in
   `res` must be 'E'.
   ```python
   assert all(res[i] == 'E' for i in range(len(n)) if n[i] == 0.0)
   ```

5. **Assertion 5: Comprehensive check for a broad range of high-tier grades.**
   For any GPA strictly greater than 2.7, its corresponding grade must correctly
   map to 'A+', 'A', 'A-', 'B+', or 'B' according to the specified rules. This
   assertion checks multiple interconnected grading rules.
   ```python
   assert all(
       (n[i] == 4.0 and res[i] == 'A+') or
       (3.7 < n[i] < 4.0 and res[i] == 'A') or
       (3.3 < n[i] <= 3.7 and res[i] == 'A-') or
       (3.0 < n[i] <= 3.3 and res[i] == 'B+') or
       (2.7 < n[i] <= 3.0 and res[i] == 'B')
       for i in range(len(n)) if n[i] > 2.7
   )
   ```

```

Listing 4: Generated formal specifications for problem 1

| # assertion | Correct | Issue |
|-------------|---------|-------|
| Assertion 1 | Yes     | -     |
| Assertion 2 | Yes     | -     |
| Assertion 3 | Yes     | -     |
| Assertion 4 | Yes     | -     |
| Assertion 5 | Yes     | -     |

Table 2: Accuracy of generated assertions for problem 1

## Exercise 2

Since I already achieved 100% for both line and branch coverage at the end of Assignment 2, I decided to use the initial test cases provided by the HumanEval+ Dataset for both problems. As a refresher, the coverage was 93% and 89% respectively, as shown in Listing 5 and 6.

| Name                                    | Stmts | Miss | Branch | BrPart | Cover | Missing   |
|-----------------------------------------|-------|------|--------|--------|-------|-----------|
| assignment3/generated_code/problem_0.py | 28    | 1    | 18     | 2      | 93%   | 5, 15->24 |
| TOTAL                                   | 28    | 1    | 18     | 2      | 93%   |           |

Listing 5: Previous test coverage for problem 0

| Name                                    | Stmts | Miss | Branch | BrPart | Cover | Missing    |
|-----------------------------------------|-------|------|--------|--------|-------|------------|
| assignment3/generated_code/problem_1.py | 29    | 3    | 26     | 3      | 89%   | 10, 18, 20 |
| TOTAL                                   | 29    | 3    | 26     | 3      | 89%   |            |

Listing 6: Previous test coverage for problem 1

For the formal specification assert statements:

### output for problem\_0

Here are 5 unique formal specifications as Python assertions for the `minPath` method:

Let `N` be the dimension of the grid, derived as `N = len(grid)`.

1. **\*\*The length of the returned path `res` must be exactly `k`\*\***

```
```python
assert len(res) == k
```
```

2. **\*\*All values in the path `res` must be valid values from the grid, i.e., integers in the range `[1, N\*N]` inclusive.\*\***

```
```python
assert all(1 <= val <= N*N for val in res)
```
```

3. **\*\*The first element of the path `res[0]` must be `1`\*\*** This is because the problem states that every integer in `[1, N\*N]` appears exactly once in the grid,

```

meaning `1` is always the minimum value available. To achieve a lexicographically
smallest path, the first element must be the smallest possible value.
```python
assert res[0] == 1
```

4. **For any two consecutive values `res[i]` and `res[i+1]` in the path, the cells
containing these values in the grid must be adjacent (share an edge).**
```python
assert all(
    any( # Check if there exist coordinates (r1, c1) for res[i]
        any( # Check if there exist coordinates (r2, c2) for res[i+1]
            grid[r1][c1] == res[i] and grid[r2][c2] == res[i+1] and (
                abs(r1 - r2) + abs(c1 - c2) == 1
            )
            for r2 in range(N) for c2 in range(N)
        )
        for r1 in range(N) for c1 in range(N)
    )
    for i in range(k - 1)
)
```

5. **If `k` is at least 2, the second element of the path `res[1]` must be the
smallest value among all direct neighbors of the cell containing `1` in the grid.**
This ensures the lexicographical minimality of the path's second element after the
initial `1`.
```python
assert (k < 2) or (
    res[1] == min(
        grid[r_neigh][c_neigh]
        for r_1 in range(N) for c_1 in range(N) if grid[r_1][c_1] == 1
        for dr, dc in [(-1, 0), (1, 0), (0, -1), (0, 1)] # Possible neighbor
    offsets
        for r_neigh, c_neigh in [(r_1 + dr, c_1 + dc)]
        if 0 <= r_neigh < N and 0 <= c_neigh < N # Check bounds for neighbor
    )
)
```
Create assert statements with the following signature:
pytest.assert_(min_path([[1, 2], [3, 4]], 10) == [1, 2, 1, 2, 1, 2, 1, 2, 1, 2])

```

Listing 7: Prompt for formal specification test generation for problem 0

```
For the formal specification assert statements:
```

```
### output for problem_1
```

Here are 5 unique formal specifications as Python assertions for the `numerical\_letter\_grade` method, where `n` is the input list of floats and `res` is the expected return value (list of strings):

```
1. **Assertion 1: Output list length and element types.**
```

The output list `res` must have the same length as the input list `n`, and every element in `res` must be a string.

```
```python
assert len(res) == len(n) and all(isinstance(grade, str) for grade in res)
```
```

```
2. **Assertion 2: Specific mapping for the highest exact GPA.**
```

If any GPA in the input list `n` is exactly 4.0, its corresponding grade in `res` must be 'A+'.

```
```python
assert all(res[i] == 'A+' for i in range(len(n)) if n[i] == 4.0)
```
```

```
3. **Assertion 3: Specific mapping for a typical mid-range GPA interval.**
```

If a GPA is strictly greater than 2.7 and less than or equal to 3.0, its corresponding letter grade must be 'B'. This demonstrates correct handling of open and closed bounds.

```
```python
assert all(res[i] == 'B' for i in range(len(n)) if 2.7 < n[i] <= 3.0)
```
```

```
4. **Assertion 4: Specific mapping for the lowest exact GPA.**
```

If any GPA in the input list `n` is exactly 0.0, its corresponding grade in `res` must be 'E'.

```
```python
assert all(res[i] == 'E' for i in range(len(n)) if n[i] == 0.0)
```
```

```
5. **Assertion 5: Comprehensive check for a broad range of high-tier grades.**
```

For any GPA strictly greater than 2.7, its corresponding grade must correctly map to 'A+', 'A', 'A-', 'B+', or 'B' according to the specified rules. This assertion checks multiple interconnected grading rules.

```
```python
assert all(
    (n[i] == 4.0 and res[i] == 'A+') or
    (3.7 < n[i] < 4.0 and res[i] == 'A') or
    (3.3 < n[i] <= 3.7 and res[i] == 'A-') or
    (3.0 < n[i] <= 3.3 and res[i] == 'B+') or
    (2.7 < n[i] <= 3.0 and res[i] == 'B')
    for i in range(len(n)) if n[i] > 2.7
)
```
```

Create assert statements with the following signature:

```
pytest.assert_(numerical_letter_grade([0, 0.7]) == ['E', 'D-'])
```

**Listing 8: Prompt for formal specification test generation for problem 1**



```

# Added Test with spec assertions
# 1. Test with k=1 (minimum path length)
pytest.assume(min_path([[1, 2], [3, 4]], 1) == [1])

# 2. Test with a 2x2 grid where the smallest neighbor is 2, k=2
pytest.assume(min_path([[1, 3], [2, 4]], 2) == [1, 2])

# 3. Test with a 2x2 grid where numbers are arranged differently, k=4
pytest.assume(min_path([[1, 4], [2, 3]], 4) == [1, 2, 1, 2])

# 4. Test with a 3x3 grid where 1 is in the center, k=5
pytest.assume(min_path([[5, 6, 7], [4, 1, 8], [3, 2, 9]], 5) == [1, 2, 1, 2, 1])

# 5. Test with a 3x3 grid where 1 is in a corner and its smallest neighbor is 4, k=3
pytest.assume(min_path([[1, 5, 6], [4, 3, 2], [7, 8, 9]], 3) == [1, 4, 1])

```

Listing 9: New generated test cases for problem 0

```

# Added Test with spec assertions
# 1. Assertion 1: Output list length and element types.
# Demonstrates that the output has the same length as input and all elements are
strings.
pytest.assume(numerical_letter_grade([0.0, 4.0, 2.5]) == ['E', 'A+', 'C+'])

# 2. Assertion 2: Specific mapping for the highest exact GPA.
# Demonstrates that a GPA of exactly 4.0 maps to 'A+'.
pytest.assume(numerical_letter_grade([4.0]) == ['A+'])

# 3. Assertion 3: Specific mapping for a typical mid-range GPA interval.
# Demonstrates that GPAs strictly greater than 2.7 and less than or equal to 3.0 map to
'B'.
pytest.assume(numerical_letter_grade([2.8, 3.0]) == ['B', 'B'])

# 4. Assertion 4: Specific mapping for the lowest exact GPA.
# Demonstrates that a GPA of exactly 0.0 maps to 'E'.
pytest.assume(numerical_letter_grade([0.0]) == ['E'])

# 5. Assertion 5: Comprehensive check for a broad range of high-tier grades.
# Demonstrates correct mappings for GPAs > 2.7 across multiple defined intervals.
pytest.assume(numerical_letter_grade([4.0, 3.8, 3.4, 3.1, 2.9]) == ['A+', 'A', 'A-', 'B+', 'B'])

```

Listing 10: New generated test cases for problem 1

| Name                                    | Stmts | Miss | Branch | BrPart | Cover | Missing   |
|-----------------------------------------|-------|------|--------|--------|-------|-----------|
| assignment3/generated_code/problem_0.py | 28    | 1    | 18     | 2      | 93%   | 5, 15->24 |
| TOTAL                                   | 28    | 1    | 18     | 2      | 93%   |           |

Listing 11: New coverage after adding formal spec tests for problem 0

| Name                                    | Stmts | Miss | Branch | BrPart | Cover | Missing |
|-----------------------------------------|-------|------|--------|--------|-------|---------|
| assignment3/generated_code/problem_1.py | 29    | 1    | 26     | 1      | 96%   | 20      |
| TOTAL                                   | 29    | 1    | 26     | 1      | 96%   |         |

Listing 12: New coverage after adding formal spec tests for problem 1

| Problem # | Old Stmt % | New Stmt % | Old Branch % | New Branch % |
|-----------|------------|------------|--------------|--------------|
| 0         | 93         | 93         | 88           | 88           |
| 1         | 89         | 96         | 88           | 96           |

Table 3: Comparison of old and new test results

As shown in Listing 11 and Table 3 the coverage for problem 0 did not increase at all. Line 5 was never checked because there was no specification stating that on an empty grid, the method should return an empty list. The same is true for line 15. However, if we think back to assignment 2 these were the only two branches which were not covered and the 100% coverage in the last assignment was only archived through specific guidance to the missing branch.

The results of problem 1, shown in Listing 12 are more interesting. Here the line coverage was increased by adding tests generated through formal specification. The only missing line is the conditional code in line 20. This could be easily fixed by providing more edge cases (or a check for a grade of  $> 2.0$ , which was not part of the formal specifications) However, when generating the tests with the prompt shown in Listing 8 the model generated a test case that led to an assertion error.

```
pytest.assume(numerical_letter_grade([0.0, 4.0, 2.5]) == ['E', 'A+', 'C+'])
```

Listing 13: Assertion error in new test cases

The issue in Listing 13 is that 2.5 is not a GPA of “C+” but instead “B-”. This could have happened because we only included the formal specification and not the original natural language description, therefore the model had less information and did not know how to evaluate “2.5”.

This leads to the assumption that only prompting with formal specification can be insufficient if these do not cover the full behaviour of the program. In addition to that, providing existing test cases like we did in assignment 2 allows the model to get even more context and

potentially generate tests for edge cases that are not yet covered, especially when the program contains a lot of if statements like in problem 1.

---

GitRepository available under:

<https://github.com/VMM2000/520-Prompting-Debugging-and-Innovation-for-Code/tree/main/assignment3>