Assignment 2

Exercise 1

```
tests/test_0.py FFFFF                                              [ 10%]
tests/test_1.py FFFFF                                              [ 20%]
tests/test_2.py FFF.F                                              [ 31%]
tests/test 3.py FF.FF                                              [ 41%]
tests/test_4.py FFF                                                [ 47%]
tests/test_5.py FFFFF                                              [ 58%]
tests/test_6.py FF.F.                                              [ 68%]
tests/test_7.py .....                                              [100%]
tests/test_8.py FFFFF                                              [ 49%]
tests/test_9.py F....                                              [ 84%]
```

Listing 1: % of test-cases passed

Coverage and failure are shown in Listing 1 and 2.
After a lot of trial and error, these results seem workable even with really bad code.
For a lot of cases with low cover rate this is because the generated function does not expect the right number of arguments.
For each other case Table 1 shows a short explanation why they did not cover everything:

Problem ID	Description
problem_2_iteration_0.py	Low coverage because of always true in if statement and a loop that never executes
problem_2_iteration_1.py	Low branch cov because if statement was always true
problem_2_iteration_4.py	same as p_2_it_4
problem_5_iteration_0.py	Low branch cov because if statement always true, so else if not checked
problem_5_iteration_1.py	Low branch cov because if statement never true
problem_5_iteration_2.py	same as p_5_it_1
problem_5_iteration_4.py	same as p_5_it_1
problem_8_iteration_0.py	loop didn't complete because of always true in if which breaks loop
problem_8_iteration_1.py	similar to p_8_it_0

Table 1: Low % Description

Every case with coverage below 100% not listed in here can be classified as low coverage due to a wrong number of arguments.
Line % and branch % are combined in Cover. Miss means which lines where not covered, Branch is the amount of branches and BrPart which brants where only partially or not covered.
Source code available in the repository as “model_outputs_zephyr_cot.csv”

```
======================================================================
Name                                    Stmts   Miss Branch BrPart  Cover   Missing
------------------------------------------------------------------------------------
generated_code/problem_0_iteration_0.py      12     11      8      0     5%   2-12
generated_code/problem_0_iteration_1.py       7      0      4      0   100%
generated_code/problem_0_iteration_2.py      11     10      8      0     5%   2-11
generated_code/problem_0_iteration_3.py       8      0      4      0   100%
generated_code/problem_0_iteration_4.py      11     10      8      0     5%   2-11
generated_code/problem_1_iteration_0.py       2      0      0      0   100%
generated_code/problem_1_iteration_1.py       4      2      0      0    50%   3-4
generated_code/problem_1_iteration_2.py      12      0      8      0   100%
generated_code/problem_1_iteration_3.py       6      0      4      0   100%
generated_code/problem_1_iteration_4.py       5      3      0      0    40%   3-5
generated_code/problem_2_iteration_0.py       8      3      6      2    50%   4, 6-7
generated_code/problem_2_iteration_1.py      13      7     10      2    35%   4, 7-13
generated_code/problem_2_iteration_2.py       7      0      6      0   100%
generated_code/problem_2_iteration_3.py       5      0      4      0   100%
generated_code/problem_2_iteration_4.py      13      7     10      2    35%   4, 7-13
generated_code/problem_3_iteration_0.py       8      6      4      0    17%   4-9
generated_code/problem_3_iteration_1.py       6      4      2      0    25%   4-7
generated_code/problem_3_iteration_2.py       4      0      0      0   100%
generated_code/problem_3_iteration_3.py       4      2      0      0    50%   4-5
generated_code/problem_3_iteration_4.py       3      1      0      0    67%   4
generated_code/problem_4_iteration_0.py      14      0     12      0   100%
generated_code/problem_4_iteration_3.py      10      1      6      1    88%   9
generated_code/problem_4_iteration_4.py      10      9      8      0     6%   3-17
generated_code/problem_5_iteration_0.py       6      1      2      1    75%   7
generated_code/problem_5_iteration_1.py       6      1      2      1    75%   5
generated_code/problem_5_iteration_2.py       6      1      2      1    75%   8
generated_code/problem_5_iteration_3.py       5      0      0      0   100%
generated_code/problem_5_iteration_4.py       6      1      2      1    75%   5
generated_code/problem_6_iteration_0.py       4      0      0      0   100%
generated_code/problem_6_iteration_1.py       5      0      0      0   100%
generated_code/problem_6_iteration_2.py       5      0      0      0   100%
generated_code/problem_6_iteration_3.py       5      0      0      0   100%
generated_code/problem_6_iteration_4.py       5      0      0      0   100%
generated_code/problem_7_iteration_0.py       2      0      0      0   100%
generated_code/problem_7_iteration_1.py       2      0      0      0   100%
generated_code/problem_7_iteration_2.py       2      0      0      0   100%
generated_code/problem_7_iteration_3.py       2      0      0      0   100%
generated_code/problem_7_iteration_4.py       2      0      0      0   100%
generated_code/problem_8_iteration_0.py       8      0      4      2    83%   4->8, 5->4
generated_code/problem_8_iteration_1.py      11      3      6      3    65%   7->4, 9-11
generated_code/problem_8_iteration_2.py       4      0      0      0   100%
generated_code/problem_8_iteration_3.py      22     20     12      0     6%   2-14,
17-23
generated_code/problem_8_iteration_4.py       9      8      2      0     9%   3-20
generated_code/problem_9_iteration_0.py       2      0      0      0   100%
generated_code/problem_9_iteration_1.py       4      0      0      0   100%
generated_code/problem_9_iteration_2.py       4      0      0      0   100%
generated_code/problem_9_iteration_3.py       4      0      0      0   100%
generated_code/problem_9_iteration_4.py       4      0      0      0   100%
------------------------------------------------------------------------------------
TOTAL                                       318    111    144     16    58%
======================================================================
```

Listing 2: Code coverage results

On closer inspection there is an issue within the assert statement and wrong return types. As soon as one of the assert statements crashes the rest of the test exists automatically.
So instead of running only assert statements, we import pytest-assume, to catch type-errors during testing and continue executing the rest of the tests.
This change improves coverage from 58% to 68% as shown in Listing 3. The final html-documentation can be seen in the repository with the name htmlcov

```
======================================================================
Name                                      Stmts   Miss Branch BrPart  Cover   Missing
-------------------------------------------------------------------------------------
generated_code/problem_0_iteration_0.py      12     11      8      0     5%   2-12
generated_code/problem_0_iteration_1.py       7      0      4      0   100%
generated_code/problem_0_iteration_2.py      11     10      8      0     5%   2-11
generated_code/problem_0_iteration_3.py       8      0      4      0   100%
generated_code/problem_0_iteration_4.py      11     10      8      0     5%   2-11
generated_code/problem_1_iteration_0.py       2      0      0      0   100%
generated_code/problem_1_iteration_1.py       4      2      0      0    50%   3-4
generated_code/problem_1_iteration_2.py      12      0      8      0   100%
generated_code/problem_1_iteration_3.py       6      0      4      0   100%
generated_code/problem_1_iteration_4.py       5      3      0      0    40%   3-5
generated_code/problem_2_iteration_0.py       8      0      6      0   100%
generated_code/problem_2_iteration_1.py      13      0     10      0   100%
generated_code/problem_2_iteration_2.py       7      0      6      0   100%
generated_code/problem_2_iteration_3.py       5      0      4      0   100%
generated_code/problem_2_iteration_4.py      13      0     10      0   100%
generated_code/problem_3_iteration_0.py       8      6      4      0    17%   4-9
generated_code/problem_3_iteration_1.py       6      4      2      0    25%   4-7
generated_code/problem_3_iteration_2.py       4      0      0      0   100%
generated_code/problem_3_iteration_3.py       4      2      0      0    50%   4-5
generated_code/problem_3_iteration_4.py       3      1      0      0    67%   4
generated_code/problem_4_iteration_0.py      14      0     12      0   100%
generated_code/problem_4_iteration_3.py      10      0      6      0   100%
generated_code/problem_4_iteration_4.py      10      9      8      0     6%   3-17
generated_code/problem_5_iteration_0.py       6      0      2      0   100%
generated_code/problem_5_iteration_1.py       6      0      2      0   100%
generated_code/problem_5_iteration_2.py       6      0      2      0   100%
generated_code/problem_5_iteration_3.py       5      0      0      0   100%
generated_code/problem_5_iteration_4.py       6      0      2      0   100%
generated_code/problem_6_iteration_0.py       4      0      0      0   100%
generated_code/problem_6_iteration_1.py       5      0      0      0   100%
generated_code/problem_6_iteration_2.py       5      0      0      0   100%
generated_code/problem_6_iteration_3.py       5      0      0      0   100%
generated_code/problem_6_iteration_4.py       5      0      0      0   100%
generated_code/problem_7_iteration_0.py       2      0      0      0   100%
generated_code/problem_7_iteration_1.py       2      0      0      0   100%
generated_code/problem_7_iteration_2.py       2      0      0      0   100%
generated_code/problem_7_iteration_3.py       2      0      0      0   100%
generated_code/problem_7_iteration_4.py       2      0      0      0   100%
generated_code/problem_8_iteration_0.py       8      0      4      2    83%   4->8, 5->4
generated_code/problem_8_iteration_1.py      11      3      6      3    65%   7->4, 9-11
generated_code/problem_8_iteration_2.py       4      0      0      0   100%
generated_code/problem_8_iteration_3.py      22     20     12      0     6%   2-14,
17-23
generated_code/problem_8_iteration_4.py       9      8      2      0     9%   3-20
generated_code/problem_9_iteration_0.py       2      0      0      0   100%
generated_code/problem_9_iteration_1.py       4      0      0      0   100%
generated_code/problem_9_iteration_2.py       4      0      0      0   100%
generated_code/problem_9_iteration_3.py       4      0      0      0   100%
generated_code/problem_9_iteration_4.py       4      0      0      0   100%
-------------------------------------------------------------------------------------
TOTAL                                       318     89    144      5    68%
======================================================================
```

Listing 3: Improved code coverage results

Exercise 2

The two most interesting problems are p_8_it_0 and p_8_it_1, as they pose some complexity with relatively low branch coverage. As shown in Listing 2 both of them fail the tests, so we can assume that the code is not correct. However there still might be the possibility to increase coverage with adding additional tests
Let's take a look at the generated code to analyze why the coverage is low

```python
def find_Rotations(s):
    n = len(s)
    min_rotations = 0
    for rotation in range(n):
        if s[n - rotation - 1:] == s[n - rotation - 1:]:
            min_rotations = rotation + 1
            break
    return min_rotations
```

Listing 4: problem_8_iteration_0.py

```python
def find_Rotations(s):
    count = 0
    same_string = False
    while not same_string:
        count += 1
        same_string = True
        for i in range(len(s)):
            if s[i] != s[i + count]:
                same_string = False
                break
    return count
```

Listing 5: problem_8_iteration_1.py

For the code in Listing 4 the if statement is always true no matter what input we set as parameters. In the html documentation we can also see that the for loop is never skipped, which indicates that there is no test where an empty string is provided. However it is a bit boring to generate new tests just to get this one case of an empty input.
The issues in Listing 5 can be broken down in a similar fashion. The while loop is never skipped as one line above the condition is set to False. In this case it is impossible to archive full branch coverage.
The second loop is never skipped, because the current tests do not include an empty string as input.
After that, the code breaks in the if statement because of an out of range exception.

At this point I decided that instead of trying to improve on this individual issue that can only be solved by adding this one specific assertion, that it would be more interesting to create new code with a better model and problems of higher complexity.
The complexity was measured by the length of the canonical solution. The new code was generated by gemini 2.5 PRO and source code and prompt can be viewed in the git repository in the folder generated_code_new.

```
======================================================================
Name                              Stmts   Miss Branch BrPart  Cover   Missing
-----------------------------------------------------------------------------
generated_code_new/problem_0.py      28      1     18      2    93%   33, 43->52
generated_code_new/problem_1.py      29      3     26      3    89%   33, 41, 43
-----------------------------------------------------------------------------
======================================================================
```
Listing 6: Coverage for new code

These problems both pass all tests but their coverage and even more importantly the branch coverage is not 100%. After manual examination I decided that additional tests could lead to better results.

```
        prompt = (
            f"Improve the branch coverage for the following code snippet\n"
            f"```python\n{code_snippet}\n```\n\n"
            f"by adding tests to these existing tests\n"
            f"```python\n{existing_tests}\n```"
        )
```
Listing 7: Prompt to increase coverage

By prompting the code shown in Listing 7 new tests were generated. Each Iteration the existing tests from the previous iteration are used.

```
==============================================================================
                     COVERAGE IMPROVEMENT SUMMARY (PER PROBLEM)
=====================================================================================
Iteration  | Problem      | Test Files | Total Cov.   | Branch Cov.  | Missing Br.  | Improvement
-------------------------------------------------------------------------------------
0          | problem_0.py | 1          | 96.43      % | 88.89      % | 2            | (Base)
0          | problem_1.py | 1          | 89.66      % | 88.46      % | 3            | (Base)
1          | problem_0.py | 2          | 100.00     % | 94.44      % | 1            | (+3.57%)
1          | problem_1.py | 2          | 100.00     % | 100.00     % | 0            | (+10.34%)
=====================================================================================
```
Listing 8: First iteration to show improvements

Addition of the generated tests resulted in improvements shown in Listing 8.

Problem 1 now archives a 100% line and branch coverage, so no further improvements are necessary. However in problem 0 there is still a loop which does not have total coverage. However, multiple iterations did not improve the total coverage as at this point the newly generated tests were redundant.
Only after I added which branch is not tested to the prompt (Listing 9) in hopes that this might hint the model in the right direction. This at last archived a coverage of 100% for both problems
The changes made for the test cases can be seen in the github repository under problem_X_iteration_Y. html documentation under htmlcov_iteration_Y.

```
        prompt = (
            f"Improve the branch coverage for the following code snippet\n"
            f"```python\n{code_snippet}\n```\n\n"
            f"by adding tests specifically to cover the branch coverage to jump from
line 15 to 24 to these existing tests\n"
            f"```python\n{existing_tests}\n```"
        )
```

Listing 9: Final prompt for test generation

```
====================================================================================
                        COVERAGE IMPROVEMENT SUMMARY (PER PROBLEM)
====================================================================================
Iteration  | Problem      | Test Files | Total Cov.    | Branch Cov.   | Missing Br.   | Improvement
------------------------------------------------------------------------------------
0          | problem_0.py | 1          | 96.43       % | 88.89       % | 2             | (Base)
0          | problem_1.py | 1          | 89.66       % | 88.46       % | 3             | (Base)
1          | problem_0.py | 2          | 100.00      % | 94.44       % | 1             | (+3.57%)
1          | problem_1.py | 2          | 100.00      % | 100.00      % | 0             | (+10.34%)
2          | problem_0.py | 3          | 100.00      % | 94.44       % | 1             | (+0.00%)
2          | problem_1.py | 2          | 100.00      % | 100.00      % | 0             | (+0.00%)
3          | problem_0.py | 4          | 100.00      % | 94.44       % | 1             | (+0.00%)
3          | problem_1.py | 2          | 100.00      % | 100.00      % | 0             | (+0.00%)
4          | problem_0.py | 5          | 100.00      % | 100.00      % | 0             | (+0.00%)
4          | problem_1.py | 2          | 100.00      % | 100.00      % | 0             | (+0.00%)
====================================================================================
```

Listing 10: Final iteration

Exercise 3

As stated in exercise 2, instead of working with my generated code from assignment 1, I decided to generate new code, which passes all the tests from the dataset.

When changing the code of problem 0 in one of the loops to an off by one error, shown in Listing 11 and 12, the tests fail because now the logic in the code is flawed. Line and branch coverage are still 100%. Report of the Listing 12 are shown in htmlcov_buggy_0

```
    for _ in range(k - 1):
      #to
    for _ in range(k): # BUG
```

Listing 11: Off-by-one 1

```
    if 0 <= nr < N and 0 <= nc < N:
      #to
    if 0 <= nr < N and 0 <= nc < N -1: # BUG
```

Listing 12: Off-by-one 2

For problem 1 instead I flipped the logic in the comparisons (I know there is no E grade in the US)

```
        elif gpa > 0.0:
            letter_grades.append('D-')
        else:
            letter_grades.append('E')
    #to
        elif gpa >= 0.0: # BUG
            letter_grades.append('D-')
        else:
            letter_grades.append('E')
```

Listing 13: Wrong logic

```
        elif gpa > 1.7:
            letter_grades.append('C')
    #to
        elif gpa > 1.7:
            letter_grades.append('C ') # BUG
```

Listing 14: Accidental space in a string

In both these cases these errors are catched again because the assertions fail with 100% line and branch coverage. These bugs are examples of common typos. While you could make the argument that Listing 14 is not a real bug and should be interpreted by the program, it is trivial why the tests fail. The documentation of Listing 14 can be viewed under htmcov_buggy_1.

All failing assertions can be seen in Table 2.

Listing 11	pytest.assume(min_path([[1, 2, 3], [4, 5, 6], [7, 8, 9]], 3) == [1, 2, 1])
	pytest.assume(min_path([[5, 9, 3], [4, 1, 6], [7, 8, 2]], 1) == [1])
	pytest.assume(min_path([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12], [13, 14, 15, 16]], 4) == [1, 2, 1, 2])
	pytest.assume(min_path([[6, 4, 13, 10], [5, 7, 12, 1], [3, 16, 11, 15], [8, 14, 9, 2]], 7) == [1, 10, 1, 10, 1, 10, 1])
	pytest.assume(min_path([[8, 14, 9, 2], [6, 4, 13, 15], [5, 7, 1, 12], [3, 10, 11, 16]], 5) == [1, 7, 1, 7, 1])
	pytest.assume(min_path([[11, 8, 7, 2], [5, 16, 14, 4], [9, 3, 15, 6], [12, 13, 10, 1]], 9) == [1, 6, 1, 6, 1, 6, 1, 6, 1])
	pytest.assume(min_path([[12, 13, 10, 1], [9, 3, 15, 6], [5, 16, 14, 4], [11, 8, 7, 2]], 12) == [1, 6, 1, 6, 1, 6, 1, 6, 1, 6, 1, 6])
	pytest.assume(min_path([[2, 7, 4], [3, 1, 5], [6, 8, 9]], 8) == [1, 3, 1, 3, 1, 3, 1, 3])
	pytest.assume(min_path([[6, 1, 5], [3, 8, 9], [2, 7, 4]], 8) == [1, 5, 1, 5, 1, 5, 1, 5])
	pytest.assume(min_path([[1, 2], [3, 4]], 10) == [1, 2, 1, 2, 1, 2, 1, 2, 1, 2])
	pytest.assume(min_path([[1, 3], [3, 2]], 10) == [1, 3, 1, 3, 1, 3, 1, 3, 1, 3])

	pytest.assume(min_path([[1]], 1) == [1])
	pytest.assume(min_path([[5, 2, 3], [4, 1, 6], [7, 8, 9]], 4) == [1, 2, 1, 2])
	pytest.assume(min_path([[1]], 5) == [1, float('inf'), float('inf'), float('inf'), float('inf')])
	pytest.assume(min_path([[2, 3], [4, 5]], 3) == [1, float('inf'), float('inf')])

Listing 12	pytest.assume(min_path([[6, 4, 13, 10], [5, 7, 12, 1], [3, 16, 11, 15], [8, 14, 9, 2]], 7) == [1, 10, 1, 10, 1, 10, 1])
	pytest.assume(min_path([[11, 8, 7, 2], [5, 16, 14, 4], [9, 3, 15, 6], [12, 13, 10, 1]], 9) == [1, 6, 1, 6, 1, 6, 1, 6, 1])
	pytest.assume(min_path([[12, 13, 10, 1], [9, 3, 15, 6], [5, 16, 14, 4], [11, 8, 7, 2]], 12) == [1, 6, 1, 6, 1, 6, 1, 6, 1, 6, 1, 6])
	pytest.assume(min_path([[6, 1, 5], [3, 8, 9], [2, 7, 4]], 8) == [1, 5, 1, 5, 1, 5, 1, 5])
	pytest.assume(min_path([[1, 2], [3, 4]], 10) == [1, 2, 1, 2, 1, 2, 1, 2, 1, 2])

Listing 13	pytest.assume(numerical_letter_grade([0.0]) == ['E'])
	pytest.assume(numerical_letter_grade([0, 0.7]) == ['E', 'D-'])

Listing 14	pytest.assume(numerical_letter_grade([4.0, 3, 1.7, 2, 3.5]) == ['A+', 'B', 'C-', 'C', 'A-'])

Table 2: Failing assertions for buggy code

This lets us conclude that the test with the addition of the generated ones yields exceptional coverage and fault detection.

GitRepository available under:
https://github.com/VMM2000/520-Prompting-Debugging-and-Innovation-for-Code/tree/main/assignment2