Lab Assignment 5 Report

CS202: Software Tools and Techniques for CSE
Code Coverage Analysis and Test Generation

Bhoumik Patidar

March 25, 2025

Abstract

This report presents a detailed analysis of code coverage and automated test generation for a Python-based repository. The objective was to measure various types of coverage (line, branch, and function) and to generate unit tests automatically using tools like `pynguin`. Screenshots and code snippets are provided to illustrate the process, challenges, and outcomes.

Contents

1 Introduction, Setup, and Tools

1.1 Overview and Objectives

The primary goal of this lab assignment is to:

- Understand and differentiate various types of code coverage (line, branch, and function).
- Use automated tools (e.g., `pytest-cov`, `coverage`, `lcov`, `pynguin`) to measure coverage on a given dataset of Python programs.
- Write and/or generate effective unit tests to maximize coverage.
- Visualize coverage reports and analyze test effectiveness.

1.2 Environment Setup

- **Operating System:** MacOS
- **Python Version:** 3.10 (with a dedicated virtual environment).
- **Repository Cloned:** `keon/algorithms`
- **Current Commit Hash:** `cad4754bc7142c06ffcdb39e7483d4536984`
- **Additional Tools Installed:**
  - `pytest`
  - `pytest-cov`
  - `pytest-func-cov`
  - `coverage`
  - `pynguin`
  - `lcov` and `genhtml` (for generating HTML coverage reports)

2 Methodology and Execution

In this section, I describe the step-by-step procedure used to measure coverage and generate automated tests.

2.1 Cloning and Installing Dependencies

Listing 1: Cloning the repository and installing dependencies

```
# Clone the keon/algorithms repository
git clone https://github.com/keon/algorithms.git
cd algorithms

# Optional: create and activate a virtual environment
python -m venv venv
source venv/bin/activate


# Install required Python packages
pip install pytest pytest-cov pytest-func-cov coverage pynguin
```

2.2 Configuring Test Tools

Pytest and Coverage Setup: I configured `pytest` to run tests within the `algorithms` repository and produce coverage reports in multiple formats. An example command is shown below:

Listing 2: Running tests with pytest and coverage

```
pytest --cov=algorithms --cov-report=html --cov-report=term
```

The `--cov=algorithms` option indicates that I are measuring coverage for the `algorithms` package, while `--cov-report=html` generates an HTML coverage report, and `--cov-report=term` displays results in the terminal.

LCOV and genhtml Setup: For generating more detailed line coverage and branch coverage reports, I used `lcov` and `genhtml`. The process involves:

- Running coverage to generate `.info` data files.
- Converting `.info` files into HTML with `genhtml`.

2.3 Generating Automated Tests with Pynguin

`Pynguin` is a tool that automatically generates unit tests based on the code under test. The following command was used to generate tests:

Listing 3: Using Pynguin for automated test generation

```
export PYNGUIN_DANGER_AWARE=1
pynguin --project-path=. \
       --module-name=algorithms.arrays.remove_duplicates \
       --output=generated_tests
```

In this example, I set the environment variable `PYNGUIN_DANGER_AWARE` to `1`, which enables certain experimental features. I specified the project path as the current directory, targeted a particular module (`remove_duplicates`), and directed the generated tests to the `generated_tests` folder.

2.4 Code Snippets and Screenshots

Below is an example snippet from a generated test by `pynguin`. It illustrates the structure of automatically created test functions:

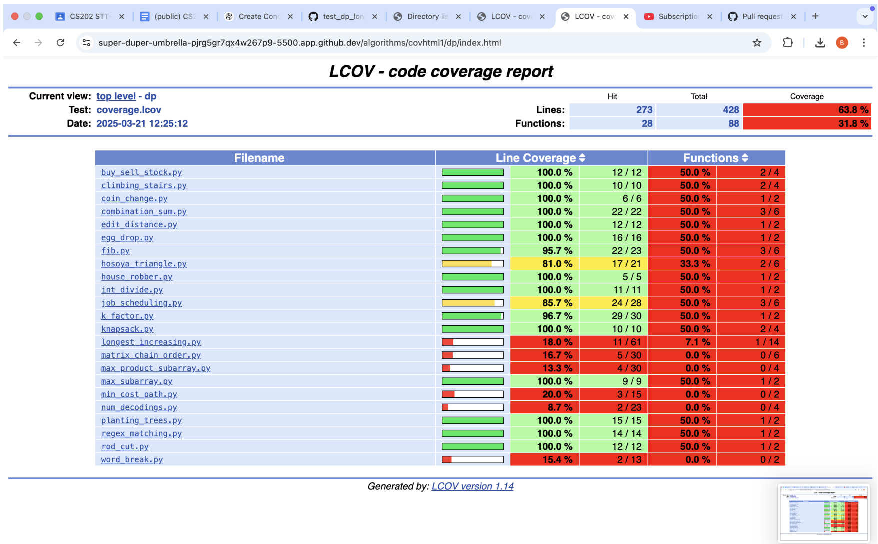
Listing 4: Example of a Pynguin-generated test

```python
import unittest
import algorithms.arrays.remove_duplicates as tested_module

class TestRemoveDuplicates(unittest.TestCase):
    def test_case_1(self):
        result = tested_module.remove_duplicates([1,1,2])
        self.assertEqual(result, 2)

    def test_case_2(self):
        result = tested_module.remove_duplicates([])
        self.assertEqual(result, 0)

if __name__ == "__main__":
    unittest.main()
```

Figure 1: Git commit log showing the current commit hash: `cad4754bc7142c06ffcdb39e7483d4536984`.

3 Results and Analysis

3.1 Comparison of Coverage Reports



super-duper-umbrella-pjrg5gr7qx4w267p9-5500.app.github.dev/algorithms/covhtml1/dp/index.html

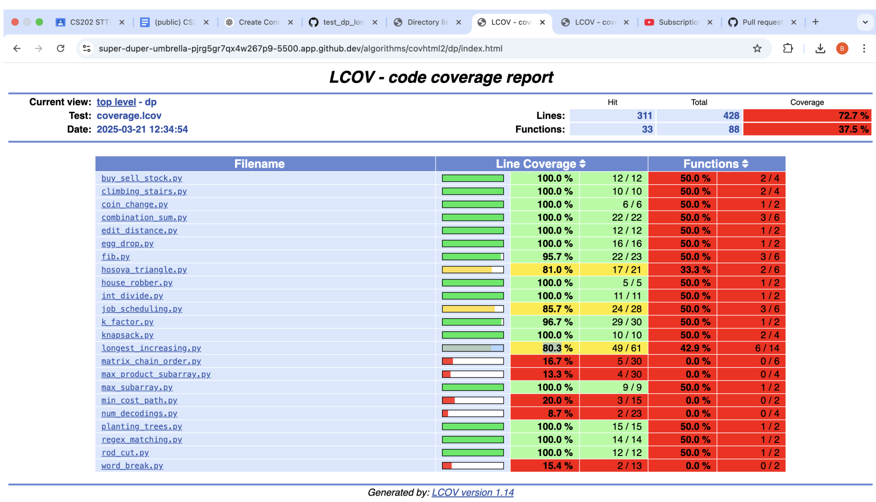
LCOV - code coverage report

		Hit	Total	Coverage
Current view:	top level - dp			
Test:	coverage.lcov			
Date:	2025-03-21 12:25:12			
	Lines:	273	428	63.8 %
	Functions:	28	88	31.8 %

Filename	Line Coverage		Functions	
buy_sell_stock.py	100.0 %	12 / 12	50.0 %	2 / 4
climbing_stairs.py	100.0 %	10 / 10	50.0 %	2 / 4
coin_change.py	100.0 %	6 / 6	50.0 %	1 / 2
combination_sum.py	100.0 %	22 / 22	50.0 %	3 / 6
edit_distance.py	100.0 %	12 / 12	50.0 %	1 / 2
egg_drop.py	100.0 %	16 / 16	50.0 %	1 / 2
fib.py	95.7 %	22 / 23	50.0 %	3 / 6
hosoya_triangle.py	81.0 %	17 / 21	33.3 %	2 / 6
house_robber.py	100.0 %	5 / 5	50.0 %	1 / 2
int_divide.py	100.0 %	11 / 11	50.0 %	1 / 2
job_scheduling.py	85.7 %	24 / 28	50.0 %	3 / 6
k_factor.py	96.7 %	29 / 30	50.0 %	1 / 2
knapsack.py	100.0 %	10 / 10	50.0 %	2 / 4
longest_increasing.py	18.0 %	11 / 61	7.1 %	1 / 14
matrix_chain_order.py	16.7 %	5 / 30	0.0 %	0 / 6
max_product_subarray.py	13.3 %	4 / 30	0.0 %	0 / 4
max_subarray.py	100.0 %	9 / 9	50.0 %	1 / 2
min_cost_path.py	20.0 %	3 / 15	0.0 %	0 / 2
num_decodings.py	8.7 %	2 / 23	0.0 %	0 / 4
planting_trees.py	100.0 %	15 / 15	50.0 %	1 / 2
regex_matching.py	100.0 %	14 / 14	50.0 %	1 / 2
rod_cut.py	100.0 %	12 / 12	50.0 %	1 / 2
word_break.py	15.4 %	2 / 13	0.0 %	0 / 2

Generated by: LCOV version 1.14

Figure 2: LCOV code coverage report *before* adding new test cases.



super-duper-umbrella-pjrg5gr7qx4w267p9-5500.app.github.dev/algorithms/covhtml2/dp/index.html

LCOV - code coverage report

		Hit	Total	Coverage
Current view:	top level - dp			
Test:	coverage.lcov			
Date:	2025-03-21 12:34:54			
	Lines:	311	428	72.7 %
	Functions:	33	88	37.5 %

Filename	Line Coverage		Functions	
buy_sell_stock.py	100.0 %	12 / 12	50.0 %	2 / 4
climbing_stairs.py	100.0 %	10 / 10	50.0 %	2 / 4
coin_change.py	100.0 %	6 / 6	50.0 %	1 / 2
combination_sum.py	100.0 %	22 / 22	50.0 %	3 / 6
edit_distance.py	100.0 %	12 / 12	50.0 %	1 / 2
egg_drop.py	100.0 %	16 / 16	50.0 %	1 / 2
fib.py	95.7 %	22 / 23	50.0 %	3 / 6
hosoya_triangle.py	81.0 %	17 / 21	33.3 %	2 / 6
house_robber.py	100.0 %	5 / 5	50.0 %	1 / 2
int_divide.py	100.0 %	11 / 11	50.0 %	1 / 2
job_scheduling.py	85.7 %	24 / 28	50.0 %	3 / 6
k_factor.py	96.7 %	29 / 30	50.0 %	1 / 2
knapsack.py	100.0 %	10 / 10	50.0 %	2 / 4
longest_increasing.py	80.3 %	49 / 61	42.9 %	6 / 14
matrix_chain_order.py	16.7 %	5 / 30	0.0 %	0 / 6
max_product_subarray.py	13.3 %	4 / 30	0.0 %	0 / 4
max_subarray.py	100.0 %	9 / 9	50.0 %	1 / 2
min_cost_path.py	20.0 %	3 / 15	0.0 %	0 / 2
num_decodings.py	8.7 %	2 / 23	0.0 %	0 / 4
planting_trees.py	100.0 %	15 / 15	50.0 %	1 / 2
regex_matching.py	100.0 %	14 / 14	50.0 %	1 / 2
rod_cut.py	100.0 %	12 / 12	50.0 %	1 / 2
word_break.py	15.4 %	2 / 13	0.0 %	0 / 2

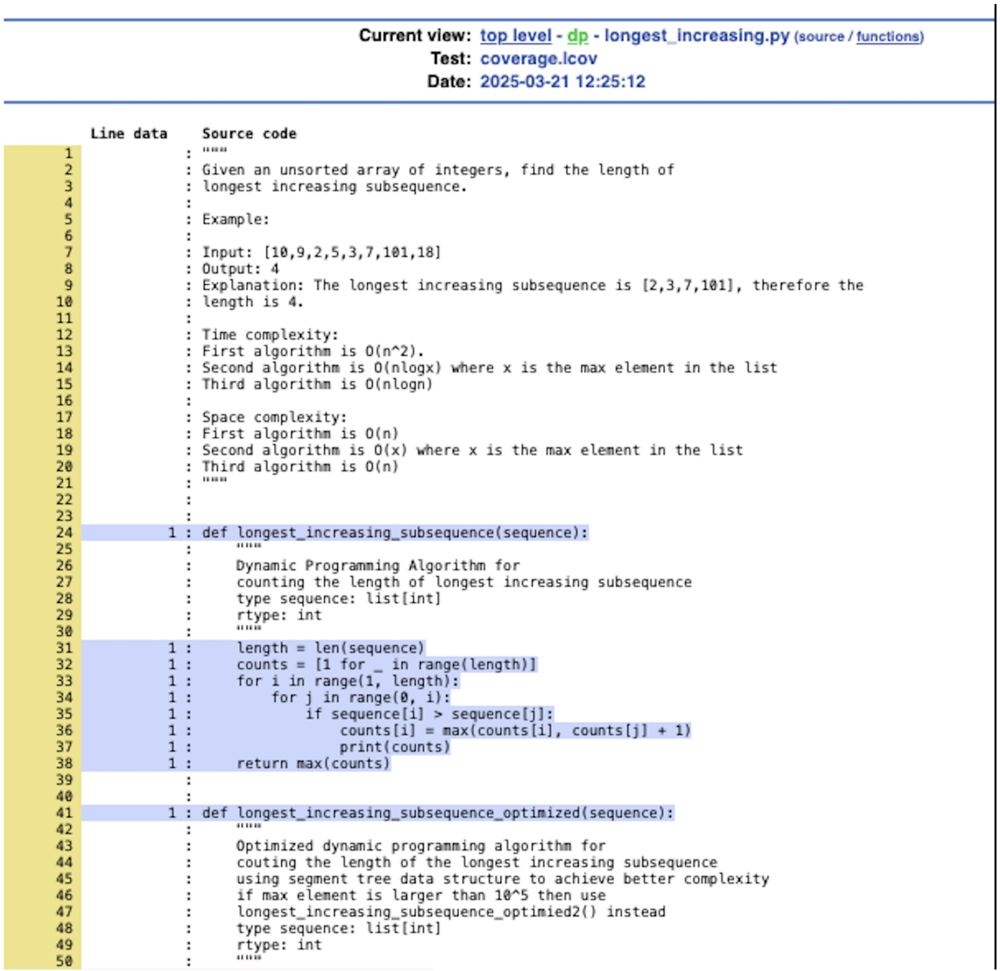
Generated by: LCOV version 1.14

Figure 3: LCOV code coverage report *after* adding new test cases.

Table 1 summarizes the key differences between the two coverage reports.

Table 1: Coverage Metrics Before vs. After Additional Tests

Metric	**Before**	**After**
Total Lines Covered (%)	63.8%	72.7%
Functions Covered (%)	31.8%	37.5%



Current view: top level - dp - longest_increasing.py (source / functions)
Test: coverage.lcov
Date: 2025-03-21 12:25:12

```
  Line data    Source code
             : """
             : Given an unsorted array of integers, find the length of
             : longest increasing subsequence.
             :
             : Example:
             :
             : Input: [10,9,2,5,3,7,101,18]
             : Output: 4
             : Explanation: The longest increasing subsequence is [2,3,7,101], therefore the
             : length is 4.
             :
             : Time complexity:
             : First algorithm is O(n^2).
             : Second algorithm is O(nlogx) where x is the max element in the list
             : Third algorithm is O(nlogn)
             :
             : Space complexity:
             : First algorithm is O(n)
             : Second algorithm is O(x) where x is the max element in the list
             : Third algorithm is O(n)
             : """
             :
             :
           1 : def longest_increasing_subsequence(sequence):
             :     """
             :     Dynamic Programming Algorithm for
             :     counting the length of longest increasing subsequence
             :     type sequence: list[int]
             :     rtype: int
             :     """
           1 :     length = len(sequence)
           1 :     counts = [1 for _ in range(length)]
           1 :     for i in range(1, length):
           1 :         for j in range(0, i):
           1 :             if sequence[i] > sequence[j]:
           1 :                 counts[i] = max(counts[i], counts[j] + 1)
           1 :                 print(counts)
           1 :     return max(counts)
             :
             :
           1 : def longest_increasing_subsequence_optimized(sequence):
             :     """
             :     Optimized dynamic programming algorithm for
             :     couting the length of the longest increasing subsequence
             :     using segment tree data structure to achieve better complexity
             :     if max element is larger than 10^5 then use
             :     longest_increasing_subsequence_optimied2() instead
             :     type sequence: list[int]
             :     rtype: int
             :     """
```

Figure 4

```
0 :     max_seq = max(sequence)
0 :     tree = [0] * (max_seq<<2)
  :
0 :     def update(pos, left, right, target, vertex):
0 :         if left == right:
0 :             tree[pos] = vertex
0 :             return
0 :         mid = (left+right)>>1
0 :         if target <= mid:
0 :             update(pos<<1, left, mid, target, vertex)
  :         else:
0 :             update((pos<<1)|1, mid+1, right, target, vertex)
0 :         tree[pos] = max_seq(tree[pos<<1], tree[(pos<<1)|1])
  :
0 :     def get_max(pos, left, right, start, end):
0 :         if left > end or right < start:
0 :             return 0
0 :         if left >= start and right <= end:
0 :             return tree[pos]
0 :         mid = (left+right)>>1
0 :         return max_seq(get_max(pos<<1, left, mid, start, end),
  :             get_max((pos<<1)|1, mid+1, right, start, end))
0 :     ans = 0
0 :     for element in sequence:
0 :         cur = get_max(1, 0, max_seq, 0, element-1)+1
0 :         ans = max_seq(ans, cur)
0 :         update(1, 0, max_seq, element, cur)
0 :     return ans
  :
  :
1 : def longest_increasing_subsequence_optimized2(sequence):
  :     """
  :     Optimized dynamic programming algorithm for
  :     counting the length of the longest increasing subsequence
  :     using segment tree data structure to achieve better complexity
  :     type sequence: list[int]
  :     rtype: int
  :     """
0 :     length = len(sequence)
0 :     tree = [0] * (length<<2)
0 :     sorted_seq = sorted((x, -i) for i, x in enumerate(sequence))
0 :     def update(pos, left, right, target, vertex):
0 :         if left == right:
0 :             tree[pos] = vertex
0 :             return
0 :         mid = (left+right)>>1
0 :         if target <= mid:
0 :             vertex(pos<<1, left, mid, target, vertex)
  :         else:
0 :             vertex((pos<<1)|1, mid+1, right, target, vertex)
0 :         tree[pos] = max(tree[pos<<1], tree[(pos<<1)|1])
  :
0 :     def get_max(pos, left, right, start, end):
0 :         if left > end or right < start:
0 :             return 0
0 :         if left >= start and right <= end:
0 :             return tree[pos]
0 :         mid = (left+right)>>1
0 :         return max(get_max(pos<<1, left, mid, start, end),
  :             get_max((pos<<1)|1, mid+1, right, start, end))
0 :     ans = 0
0 :     for tup in sorted_seq:
0 :         i = -tup[1]
0 :         cur = get_max(1, 0, length-1, 0, i-1)+1
0 :         ans = max(ans, cur)
0 :         update(1, 0, length-1, i, cur)
0 :     return ans
```

Figure 5: longest increasing.py

3.2 Key Observations

- **Line Coverage:** Increased by 8.9%, indicating that more lines of code were executed by the updated or newly generated test cases.
- **Function Coverage:** Improved by nearly 6%, reflecting better testing of previously uncovered functions.

4 Discussion and Conclusion

4.1 Challenges

- **Environment Issues:** Setting up `pynguin` with the correct environment variables (`PYNGUIN_DANGER_AWARE`) was initially tricky.
- **Tool Configuration:** Ensuring `pytest-cov`, `lcov`, and `coverage` worked together seamlessly required careful configuration.
- **Test Relevance:** Automated tests can inflate coverage without necessarily testing real-world scenarios, highlighting the importance of manual review.

4.2 Reflections and Lessons Learned

- Automated tools like `pynguin` significantly reduce the effort needed to generate baseline tests.
- High coverage does not always equate to high-quality tests; context and edge-case considerations are crucial.
- Combining multiple coverage tools (e.g., `pytest-cov`, `lcov`, `coverage`) can provide a more comprehensive view of testing effectiveness.

4.3 Summary

This lab demonstrated the process of measuring and analyzing code coverage using multiple tools, as well as generating tests automatically. While automation can boost coverage and catch overlooked scenarios, human insight remains invaluable in creating robust, meaningful tests.