

ASSIGNMENT 2

CS 202: SOFTWARE TOOLS AND TECHNIQUES FOR CSE

Shubham Agrawal

22110249

Department of Computer Science and Engineering

Supervisor:

Prof. Shouvick Mondal

Indian Institute of Technology Gandhinagar

Monday 24th March, 2025

Contents

1	Introduction	1
2	LAB-5 <i>dated 06/02/25</i>	2
2.1	Lab Topic	2
2.2	Overview	2
2.3	Objectives	2
2.4	Set-up and Tools	3
2.5	Methodology and Execution	3
2.6	Results	6
2.7	Discussion and Conclusion	15
2.8	Files and Code Location	16
3	LAB-6 <i>dated 13/02/25</i>	17
3.1	Lab Topic	17
3.2	Overview	17
3.3	Objectives	17
3.4	Set-up and Tools	18
3.5	Methodology and Execution	18
3.6	Results and Analysis	23
3.7	Discussion and Conclusion	27
3.8	Files and Code Location	28
4	LAB-7 and LAB-8 <i>dated 20/02/25 and 27/02/25</i>	29
4.1	Lab Topic	29
4.2	Overview	29
4.3	Objectives	29
4.4	Set-up and Tools	29

4.5	Methodology and Execution	30
4.6	Results and Analysis	33
4.7	Research Questions	38
4.8	Discussion and Conclusion	40
4.9	Files and Code Location	42

1 Introduction

This assignment contains a detailed and structured report for the Software Tools and Techniques for CSE (CS202) course labs carried out in Febraury 2025. There were a total of four labs which were performed on 6th, 13th, 20th, and 27th February.

2 LAB-5 dated 06/02/25

2.1 Lab Topic

Code Coverage Analysis and Test Generation.

2.2 Overview

In this laboratory, I performed a code coverage analysis and test generation methodologies to ascertain the efficacy of unit testing. I used a collection of Python packages and applied numerous automated testing tools to quantify various kinds of code coverage, such as line coverage, branch coverage, and function coverage. The main objective was to determine the extent to which the given test suite tested the provided codebase and enhance test coverage by creating more test cases.

To obtain this, I cloned the keon/algorithms repository and set up tools like pytest, pytest-cov, coverage, and pynguin for dynamic analysis. I ran the existing test suite (Test Suite A) and measured its coverage statistics, then created new test cases (Test Suite B) to enhance coverage. Lastly, I compared the success of both test suites and examined if generated tests uncovered uncovered situations.

2.3 Objectives

The primary objectives of this lab were:

- To understand and differentiate various types of code coverage, including line, branch, and function coverage.
- To measure and analyze code coverage using automated tools on a given dataset of Python programs.
- To execute an existing test suite and evaluate its coverage metrics.
- To generate additional unit test cases using Pynguin to maximize code coverage.
- To compare the effectiveness of the generated test cases against the original test suite.

- To visualize coverage reports and analyze whether the generated tests identified any previously uncovered scenarios.

2.4 Set-up and Tools

The following tools and set-ups were used to complete the lab objectives:

- **Operating System:** Windows 11
- **Python 3.10:** Coding Language Used.
- **Lightening AI:** Remote development platform with GPU availability along with an IDE similar to VS-code. Used for the lab as a VM.
- **Tools:** pytest, pytest-cov, coverage, pynguin genhtml, lcov

2.5 Methodology and Execution

2.5.1 Cloning Repository:

keon/algorithms was cloned using:

```
git clone https://github.com/keon/algorithms.git
```

The current commit hash of the repository is: cad4754bc71742c2d6fcbd3b92ae74834d359844

Then, inside the algorithms directory, the requirements of the repository were installed using:

```
pip install -e .
```

2.5.2 Installing Dependencies:

The pytest, pytest-cov, coverage, pynguin tools were installed using:

```
pip install pytest pytest-cov coverage pytest-func-cov pynguin
```

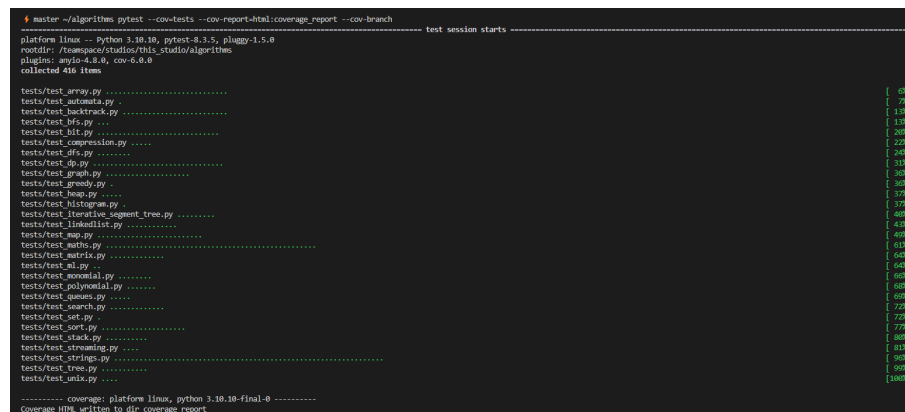
Additionally, `lcov` was installed for better visualisation using:

```
sudo apt update  
sudo apt install lcov
```

2.5.3 Running the Existing Test Suite (Test Suite A)

I executed the provided test suite inside the `tests` folder using:

```
pytest --cov=algorithms --cov-branch --cov-report=html:coverage_report
```



```
master ~/algorithms pytest --cov=tests --cov-report=html:coverage_report --cov-branch test session starts
platform linux -- Python 3.10.10, pytest-8.3.5, pluggy-1.5.0
rootdir: /workspace/studios/this_studio/algorithms
plugins: anyio-4.2.0, cov-6.0.0
collected 416 items

tests/test_array.py ..... [ 68%]
tests/test_automata.py ..... [ 75%]
tests/test_backtrack.py ..... [ 100%]
tests/test_bfs.py ..... [ 135%]
tests/test_bit.py ..... [ 200%]
tests/test_compression.py ..... [ 220%]
tests/test_dp.py ..... [ 240%]
tests/test_dp.py ..... [ 315%]
tests/test_graph.py ..... [ 360%]
tests/test_heap.py ..... [ 375%]
tests/test_histogram.py ..... [ 375%]
tests/test_iterative_segment_tree.py ..... [ 400%]
tests/test_linkedlist.py ..... [ 430%]
tests/test_map.py ..... [ 495%]
tests/test_maths.py ..... [ 615%]
tests/test_matrix.py ..... [ 640%]
tests/test_queue.py ..... [ 645%]
tests/test_monomial.py ..... [ 655%]
tests/test_polynomial.py ..... [ 655%]
tests/test_queues.py ..... [ 655%]
tests/test_search.py ..... [ 655%]
tests/test_set.py ..... [ 655%]
tests/test_sort.py ..... [ 655%]
tests/test_stack.py ..... [ 655%]
tests/test_streaming.py ..... [ 655%]
tests/test_strings.py ..... [ 655%]
tests/test_tree.py ..... [ 655%]
tests/test_union.py ..... [ 655%]

----- coverage: platform linux, python 3.10.10-final-0 -----
Coverage HTML written to dir coverage_report
```

Figure 1: Running Pytest on test suit A

Now, for better visualisation, I ran the similar command for generating `.lcov` report and using `coverage` and `lcov` for generating visualizations.

```
pytest --cov=algorithms --cov-branch --cov-report=lcov  
coverage lcov -o coverage.lcov  
genhtml coverage.lcov --output-directory coverage_html
```

2.5.4 Generating new Test Suite (Test Suite B)

I created a python script called `pynguin.py` that uses `pynguin` to generate new tests cases for 5 modules inside the `dp` module that had the lowest coverage. These five modules were:

- `longest_common_subsequence`

- matrix_chain_order
- min_cost_path
- num_decodings
- word_break

```

algorithms > pynguin.py ...
1  import os
2  # Set required environment variable
3  os.environ["PYNGUIN_DANGER_AWARE"] = "YES"
4  # Base path of your project
5  project_path = "."
6  # Directory containing all 'dp' modules
7  dp_dir: LiteralString = os.path.join(project_path, "algorithms", "dp")
8  # Output directory for generated tests
9  output_dir = "generated_tests"
10
11 # List of low-coverage files (without extensions)
12 low_coverage: list[str] = [
13     "longest_common_subsequence",
14     "matrix_chain_order",
15     "min_cost",
16     "num_decodings",
17     "word_break"
18 ]
19
20 # Find and run Pynguin only for low-coverage modules
21 for file in low_coverage:
22     module: str = f"algorithms.dp.{file}"
23     print(f"Generating tests for: {module}")
24     os.system(command=f"pynguin --project-path={project_path} --output-path={output_dir} --module-name={module}")
25

```

Figure 2: Script for Generating New test cases [pynguin.py]

These new tests cases generated were stored inside the generated_tests folder. This folder along with the tests folder is now our **test suite B**.

2.5.5 Running the new Test Suite (Test Suite B)

I executed the test suite inside the tests folder and the generated_tests folder using:

```
pytest --cov=algorithms --cov-branch --cov-report=html:coverage_report
```



```

$ master ~/algorithms pytest --cov=algorithms --cov-report=html:coverage_report --cov-branch
===== test session starts =====
platform linux -- Python 3.10.10, pytest-8.3.5, pluggy-1.5.0
rootdir: /teamspace/studios/this_studio/algorithms
plugins: anyio-4.6.0, cov-6.0.0
collected 445 items

generated_tests/test_algorithms_dp_longest_common_subsequence.py xxx [ 0%]
generated_tests/test_algorithms_dp_matrix_chain_order.py x..x.. [ 2%]
generated_tests/test_algorithms_dp_min_cost_path.py x.. [ 2%]
generated_tests/test_algorithms_dp_num_decodings.py x..xxxxxxxxxxxx [ 5%]
generated_tests/test_algorithms_dp_word_break.py x.. [ 6%]
tests/test_array.py ..... [ 13%]
tests/test_automata.py . [ 13%]
tests/test_backtrack.py ..... [ 18%]
tests/test_bfs.py ... [ 19%]
tests/test_bst.py ..... [ 20%]
tests/test_compression.py ..... [ 27%]
tests/test_dfs.py ..... [ 28%]
tests/test_dp.py ..... [ 35%]
tests/test_graph.py ..... [ 40%]
tests/test_greedy.py ..... [ 48%]
tests/test_heap.py ..... [ 41%]
tests/test_histogram.py ..... [ 42%]
tests/test_iterative_segment_tree.py ..... [ 44%]
tests/test_linkedlist.py ..... [ 46%]
tests/test_map.py ..... [ 52%]
tests/test_math.py ..... [ 53%]
tests/test_matrix.py ..... [ 60%]
tests/test_ml.py .. [ 60%]
tests/test_monomial.py ..... [ 68%]
tests/test_polynomial.py ..... [ 70%]
tests/test_queues.py ..... [ 71%]
tests/test_search.py ..... [ 74%]
tests/test_set.py . [ 74%]
tests/test_sort.py ..... [ 78%]
tests/test_stack.py ..... [ 81%]
tests/test_streaming.py ..... [ 82%]
tests/test_strings.py ..... [ 90%]
tests/test_trees.py ..... [ 90%]
tests/test_unix.py .... [ 100%]

----- coverage: platform linux, python 3.10.10-final-0 -----
Coverage HTML written to dir coverage_report

424 passed, 21 xfailed in 9.90s

```

Figure 3: Running Pytest on test suit B

Now, for better visualisation, I ran the similar command for generating .lcov report and using coverage and lcov for generating visualizations.

```

pytest --cov=algorithms --cov-branch --cov-report=lcov
coverage lcov -o coverage.lcov
genhtml coverage.lcov --output-directory coverage_html

```

2.6 Results

2.6.1 Outputs

After executing the original test suite (Test Suite A), I collected code coverage metrics using pytest-cov. The initial results showed that while some functions had decent coverage, several modules—particularly in dp, set, linkedlist, trees and graphs had low coverage.

Test Suit A results:

Coverage report: 68%

Files Functions Classes

coverage.py v7.0.0, created at 2020-03-17 10:51 +0000

File	statements	missing	excluded	branches	partial	coverage
algorithms/array/delete_nth.py	15	0	0	0	0	100%
algorithms/array/flatten.py	14	0	0	10	0	100%
algorithms/array/parag.py	18	0	0	0	1	100%
algorithms/array/josephus.py	8	0	0	2	0	100%
algorithms/array/limit.py	8	1	0	6	1	85%
algorithms/array/longest_non_repeat.py	61	14	0	32	4	77%
algorithms/array/max_plus.py	16	0	0	0	0	100%
algorithms/array/merge_intervals.py	48	16	0	18	2	64%
algorithms/array/missing_ranges.py	13	0	0	0	1	95%
algorithms/array/move_zeros.py	10	0	0	4	0	100%
algorithms/array/n_sum.py	64	0	0	28	1	99%
algorithms/array/plus_one.py	30	0	0	14	0	100%
algorithms/array/remove_duplicates.py	16	0	0	4	0	100%
algorithms/array/rotate.py	20	1	0	0	1	100%
algorithms/array/summarize_ranges.py	14	1	0	6	1	90%
algorithms/array/three_sum.py	21	1	0	14	1	94%
algorithms/array/top_1.py	14	0	0	0	0	100%
algorithms/array/two_sum.py	9	0	0	2	0	100%
algorithms/array/two_sum.py	7	0	0	0	0	100%
algorithms/array/two_sum.py	12	1	0	0	1	90%
algorithms/backtrack/perm_operations.py	20	1	0	12	1	94%

Figure 4: Result Test suit A

algorithms/tree/construct_btree_postorder_preorder.py	40	7	0	18	3	61%
algorithms/tree/deepest_left.py	25	23	0	0	0	0%
algorithms/tree/ferwick_tree/ferwick_tree.py	21	0	0	6	0	100%
algorithms/tree/insert_delete.py	8	0	0	4	0	100%
algorithms/tree/is_balanced.py	12	12	0	4	0	0%
algorithms/tree/is_subtree.py	19	19	0	0	0	0%
algorithms/tree/is_symmetric.py	25	25	0	16	0	0%
algorithms/tree/longest_consecutive.py	15	15	0	6	0	0%
algorithms/tree/lowest_common_ancestor.py	6	0	0	4	0	100%
algorithms/tree/max_height.py	25	23	0	14	0	0%
algorithms/tree/max_path_sum.py	11	11	0	2	0	0%
algorithms/tree/min_height.py	40	40	0	20	0	0%
algorithms/tree/path_sum.py	35	35	0	28	0	0%
algorithms/tree/path_sum2.py	42	42	0	28	0	0%
algorithms/tree/print.py	10	10	0	6	0	0%
algorithms/tree/segment_tree/segment_tree.py	15	0	0	10	0	100%
algorithms/tree/traversal/border.py	40	14	0	12	2	65%
algorithms/tree/traversal/level_order.py	17	17	0	10	0	0%
algorithms/tree/traversal/postorder.py	11	4	0	14	1	61%
algorithms/tree/traversal/preorder.py	28	4	0	12	1	85%
algorithms/tree/traversal/serialize.py	10	10	0	10	0	0%
algorithms/tree/traversal.py	1	1	0	0	0	0%
algorithms/tree/path/dfs.py	3	0	0	0	0	100%
algorithms/tree/path/dfs.py	6	0	0	0	0	100%
algorithms/tree/path/dfs.py	11	1	0	0	1	85%
algorithms/tree/path/dfs.py	7	0	0	0	0	100%
Total	794	248	0	178	21	68%

coverage.py v7.0.0, created at 2020-03-17 10:51 +0000

Figure 5: Result Test suit A

Coverage report: 68%

Files Functions Classes

coverage.py v7.0.0, created at 2020-03-17 10:51 +0000

File	function	statements	missing	excluded	branches	partial	coverage
algorithms/array/delete_nth.py	delete_nth	15	0	0	4	0	100%
algorithms/array/delete_nth.py	delete_nth	7	0	0	4	0	100%
algorithms/array/delete_nth.py	(no function)	1	0	0	0	0	100%
algorithms/array/flatten.py	flatten	7	0	0	6	0	100%
algorithms/array/flatten.py	flatten_iter	4	0	0	4	0	100%
algorithms/array/flatten.py	(no function)	1	0	0	0	0	100%
algorithms/array/parag.py	parag	18	0	0	2	1	100%
algorithms/array/parag.py	(no function)	2	0	0	0	0	100%
algorithms/array/josephus.py	josephus	7	0	0	2	0	100%
algorithms/array/josephus.py	(no function)	1	0	0	0	0	100%
algorithms/array/limit.py	limit	7	1	0	6	1	85%
algorithms/array/limit.py	(no function)	1	0	0	0	0	100%
algorithms/array/longest_non_repeat.py	longest_non_repeat_v1	12	1	0	6	1	85%
algorithms/array/longest_non_repeat.py	longest_non_repeat_v2	10	1	0	0	1	90%
algorithms/array/longest_non_repeat.py	get_longest_non_repeat_v1	14	1	0	0	1	93%
algorithms/array/longest_non_repeat.py	get_longest_non_repeat_v2	13	1	0	0	1	96%
algorithms/array/longest_non_repeat.py	get_longest_non_repeat_v3	10	10	0	4	0	0%
algorithms/array/longest_non_repeat.py	(no function)	5	0	0	0	0	100%
algorithms/array/max_one_index.py	max_one_index	15	0	0	0	0	100%
algorithms/array/max_one_index.py	(no function)	1	0	0	0	0	100%
algorithms/array/merge_intervals.py	Interval__init__	2	0	0	0	0	100%
algorithms/array/merge_intervals.py	Interval__repr__	1	1	0	0	0	0%
algorithms/array/merge_intervals.py	Interval__str__	1	1	0	0	0	0%
algorithms/array/merge_intervals.py	Interval__getitem__	3	3	0	2	0	0%
algorithms/array/merge_intervals.py	Interval__len__	1	1	0	0	0	0%

Figure 6: Result Test suit A

algorithms/tree/insert_node_iterative_segment_tree.py	(no function)	5	0	0	0	100%	
algorithms/tree/traversal/number.py	Node__init__	1	3	0	0	6%	
algorithms/tree/traversal/number.py	inorder	12	1	0	6	81%	
algorithms/tree/traversal/number.py	inorder_rec	8	0	0	4	100%	
algorithms/tree/traversal/number.py	(no function)	17	10	0	7	71%	
algorithms/tree/traversal/level_order.py	level_order	16	10	0	10	6%	
algorithms/tree/traversal/level_order.py	(no function)	1	1	0	0	6%	
algorithms/tree/traversal/postorder.py	Node__init__	1	3	0	0	6%	
algorithms/tree/traversal/postorder.py	postorder	16	1	0	10	92%	
algorithms/tree/traversal/postorder.py	postorder_rec	8	0	0	4	100%	
algorithms/tree/traversal/postorder.py	(no function)	4	0	0	0	100%	
algorithms/tree/traversal/preorder.py	Node__init__	1	3	0	0	6%	
algorithms/tree/traversal/preorder.py	preorder	13	1	0	1	96%	
algorithms/tree/traversal/preorder.py	preorder_rec	8	0	0	4	100%	
algorithms/tree/traversal/preorder.py	(no function)	4	0	0	0	100%	
algorithms/tree/traversal/rigzigzag.py	(no function)	18	10	0	10	6%	
algorithms/tree/traversal/rigzigzag.py	(no function)	1	1	0	0	6%	
algorithms/tree/tree.py	Tree.__init__	3	3	0	0	6%	
algorithms/tree/tree.py	(no function)	2	2	0	0	6%	
algorithms/union/path/full_path.py	full_path	1	0	0	0	100%	
algorithms/union/path/full_path.py	(no function)	2	0	0	0	100%	
algorithms/union/path/union_with_slash.py	join_with_slash	4	0	0	0	100%	
algorithms/union/path/union_with_slash.py	(no function)	2	0	0	0	100%	
algorithms/union/path/simplify_path.py	simplify_path_v1	1	0	0	0	100%	
algorithms/union/path/simplify_path.py	simplify_path_v2	7	1	0	1	83%	
algorithms/union/path/simplify_path.py	(no function)	3	0	0	0	100%	
algorithms/union/path/right.py	split	5	0	0	0	100%	
algorithms/union/path/right.py	(no function)	2	0	0	0	100%	
Total		7964	2468	0	3780	256	44%

coverage.py v7.7.0, created at 2025-09-17 10:51 -0000

LCOV - code coverage report									
Current view: top level				Lines:		5526	Total	7994	Coverage
Test: coverage.bov				Functions:		615		1757	88.1
Date: 2025-03-17 11:13:58									
Directory				Line Coverage %		Functions %			
src				91.6	371 / 405	81.9	38 / 44	38.9	91.6
src/asm				91.7	11 / 12	90.9	1 / 2	50.0	91.7
src/backtrack				96.1	274 / 285	98.8	1 / 1	50.0	96.1
src/bfs				77.1	84 / 109	84.8	1 / 1	50.0	77.1
src/bil				99.5	203 / 204	99.0	1 / 1	50.0	99.5
src/boundedset				92.2	237 / 257	94.9	1 / 1	50.0	92.2
src/bst				90.9	160 / 176	94.4	1 / 1	50.0	90.9
src/distribution				100.0	6 / 6	99.7	1 / 1	50.0	100.0
src/enum				93.2	213 / 228	25.1	1 / 1	20.0	93.2
src/eval				98.8	408 / 413	23.3	1 / 1	42.0	98.8
src/eval2				91.7	11 / 12	90.8	1 / 1	10.0	91.7
src/eval3				77.9	84 / 107	27.6	1 / 1	10.0	77.9
src/eval4				77.6	94 / 121	17.6	1 / 1	17.0	77.6
src/eval5				94.8	111 / 235	34.8	1 / 1	23.0	94.8
src/eval6				73.3	104 / 141	44.8	1 / 1	42.0	73.3
src/eval7				99.6	20 / 20	99.9	1 / 1	20.0	99.6
src/eval8				100.0	20 / 20	99.9	1 / 1	4.4	100.0
src/eval9				94.9	164 / 172	31.8	1 / 1	21.0	94.9
src/eval10				94.7	198 / 209	30.4	1 / 1	20.4	94.7
src/eval11				7.9	8 / 100	8.9	1 / 1	11.0	7.9
src/eval12				78.1	104 / 133	43.8	1 / 1	36.0	78.1
src/eval13				83.2	237 / 273	42.1	1 / 1	32.0	83.2
src/eval14				100.0	9 / 9	90.3	1 / 1	5.0	100.0
src/eval15				94.2	947 / 708	94.9	1 / 1	70.0	94.2
src/eval16				99.9	198 / 198	99.9	1 / 1	99.0	99.9
src/eval17				9.0	8 / 7	9.0	1 / 1	17.0	9.0
src/eval18				100.0	21 / 21	99.7	1 / 1	4.0	100.0
src/eval19				100.0	70 / 70	99.8	1 / 1	4.0	100.0
src/eval20				99.8	70 / 70	97.2	1 / 1	6.2	99.8
src/eval21				99.3	26 / 27	99.8	1 / 1	6.0	99.3

LCOV - code coverage report									
Current view: top level - 0p		Test: coverage-test		Lines: 18		Total: 465		Coverage	
Date: 2025-03-17 11:13:58				Functions: 28		90		92.5	
Filename		Line Coverage %		Functions %					
src/api/etcdapi.go	<div><div></div></div>	100.0%	12 / 12			100.0%	2 / 4		2 / 4
src/api/status.go	<div><div></div></div>	100.0%	10 / 10			100.0%	2 / 4		2 / 4
src/client.go	<div><div></div></div>	100.0%	8 / 8			100.0%	1 / 2		1 / 2
src/collector/api.go	<div><div></div></div>	100.0%	22 / 22			100.0%	2 / 2		2 / 2
src/collector/api_test.go	<div><div></div></div>	100.0%	12 / 12			100.0%	3 / 3		3 / 3
src/etcdapi.go	<div><div></div></div>	100.0%	18 / 18			100.0%	1 / 2		1 / 2
file.go	<div><div></div></div>	100.0%	22 / 22			100.0%	2 / 2		2 / 2
metrics/collector.go	<div><div></div></div>	81.0%	17 / 21			100.0%	2 / 6		2 / 6
metrics/collector_test.go	<div><div></div></div>	100.0%	5 / 5			100.0%	1 / 2		1 / 2
src/etcdapi_test.go	<div><div></div></div>	100.0%	11 / 11			100.0%	1 / 2		1 / 2
src/instrumentation.go	<div><div></div></div>	88.7%	24 / 28			100.0%	3 / 3		3 / 3
test/collector.go	<div><div></div></div>	100.0%	20 / 20			100.0%	1 / 2		1 / 2
test/collector_test.go	<div><div></div></div>	100.0%	10 / 10			100.0%	2 / 4		2 / 4
test/collector_test_suite.go	<div><div></div></div>	100.0%	1 / 1			100.0%	0 / 1		0 / 1
test/collector_test_suite_test.go	<div><div></div></div>	100.0%	11 / 11			100.0%	1 / 16		1 / 16
test/collector_test_suite_test_test.go	<div><div></div></div>	100.0%	5 / 5			100.0%	0 / 6		0 / 6
test/collector_test_suite_test_test_test.go	<div><div></div></div>	100.0%	1 / 1			100.0%	0 / 4		0 / 4
test/collector_test_suite_test_test_test_test.go	<div><div></div></div>	100.0%	0 / 0			100.0%	0 / 2		0 / 2
test/collector_test_suite_test_test_test_test.go	<div><div></div></div>	8.3%	2 / 25			100.0%	0 / 4		0 / 4
test/collector_test_suite_test_test_test_test_test.go	<div><div></div></div>	100.0%	15 / 15			100.0%	0 / 2		0 / 2
test/collector_test_suite_test_test_test_test_test_test.go	<div><div></div></div>	100.0%	14 / 14			100.0%	1 / 2		1 / 2
test/collector_test_suite_test_test_test_test_test_test_test.go	<div><div></div></div>	100.0%	12 / 12			100.0%	0 / 2		0 / 2
test/collector_test_suite_test_test_test_test_test_test_test_test.go	<div><div></div></div>	100.0%	2 / 2			100.0%	0 / 2		0 / 2

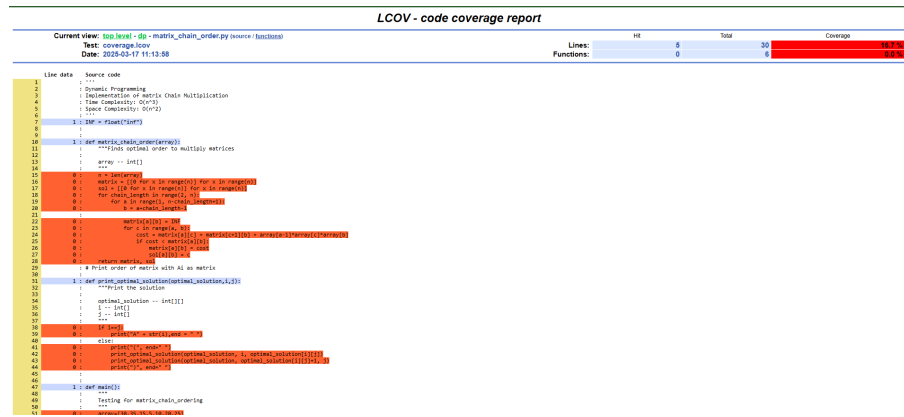


Figure 10: Result Test suit A - Matrix_chain_order

Finally, after executing the newly generated test suite using pynguin (Test Suite B), I collected code coverage metrics using pytest-cov again. The results showed a mix of passed (.) and expected failures (x, XFailed). The presence of XFailed tests indicated cases where failures were anticipated due to specific constraints in the tested functions. However, there was greater improvement in the test coverage of the modules whose tests cases were generated using pynguin. Overall coverage also improved.

Sample Test file from Test Suit B:

```

algorithms > generated_tests > test_algorithms_dp_matrix_chain_order.py > test_case_5
1  # Test cases automatically generated by PyPenguin (https://www.pyPenguin.eu).
2  # Please check them before you use them.
3  import pytest
4  import builtins as module_0
5  from algorithms.dp.matrix_chain_order import matrix_chain_order, print_optimal_solution
6
7
8
9  @pytest.mark.xfail(strict=True)
10 def test_case_0() -> None:
11     object_0: object = module_0.object()
12     matrix_chain_order(array=object_0)
13
14
15 @pytest.mark.xfail(strict=True)
16 def test_case_1() -> None:
17     str_0 = "J(PeU(t"
18     matrix_chain_order(array=str_0)
19
20
21 def test_case_2() -> None:
22     tuple_0: tuple[()] = ()
23     var_0: tuple[list[list[int]], list[list[int]]] = matrix_chain_order(ar=tuple_0)
24
25
26 def test_case_3() -> None:
27     str_0 = "JRCe(K"
28     var_0: None = print_optimal_solution(optimal_solution=str_0, i=str_0, j=str_0)
29
30
31 @pytest.mark.xfail(strict=True)
32 def test_case_4() -> None:
33     bool_0 = True
34     none_type_0: None = None
35     print_optimal_solution(optimal_solution=none_type_0, i=none_type_0, j=bool_0)
36
37
38 def test_case_5() -> None:
39     bytes_0 = b"\x8e\xba\x9c\xab"
40     var_0: tuple[list[list[int]], list[list[int]]] = matrix_chain_order(ar=bytes_0)
41

```

Figure 11: test_algorithms_dp_matrix_chain_order.py

Test Suit B results:

Coverage report: 69%

Files Functions Classes

coverage.py v7.6, created at 2025-03-20 22:14 +0000

File	statements	missing	excluded	branches	partial	coverage
algorithms/array/delete_nth.py	15	0	0	0	0	100%
algorithms/array/flatten.py	14	0	0	10	0	100%
algorithms/array/generate.py	10	0	0	0	1	94%
algorithms/array/josephus.py	9	0	0	2	0	100%
algorithms/array/limit.py	9	1	0	6	1	88%
algorithms/array/longest_non_repeat.py	63	14	0	32	4	77%
algorithms/array/max_index.py	16	0	0	0	0	100%
algorithms/array/merge_intervals.py	46	10	0	38	2	64%
algorithms/array/missing_ranges.py	12	0	0	0	1	92%
algorithms/array/move_zeros.py	30	0	0	4	0	100%
algorithms/array/po_sum.py	44	0	0	20	1	95%
algorithms/array/plus_one.py	30	0	0	14	0	100%
algorithms/array/remove_duplicates.py	6	0	0	4	0	100%
algorithms/array/rotate.py	28	1	0	0	1	94%
algorithms/array/summarize_ranges.py	14	1	0	0	1	90%
algorithms/array/two_sum.py	23	1	0	14	1	94%
algorithms/array/two_sum_2.py	14	0	0	0	0	100%
algorithms/array/two_sum_3.py	9	0	0	2	0	100%
algorithms/array/two_sum_4.py	7	0	0	4	0	100%
algorithms/array/two_sum_5.py	12	1	0	0	1	90%
algorithms/backtrack/add_operations.py	20	1	0	12	1	94%
algorithms/backtrack/analyze.py	10	0	0	4	0	100%
algorithms/backtrack/array_sum_combinations.py	47	0	0	22	0	100%
algorithms/backtrack/combinations_sum.py	13	0	0	6	0	100%
algorithms/backtrack/factor_combinations.py	10	0	0	10	0	100%

Figure 12: Result Test suit B

algorithms/tree/binary_tree_path.py	11	11	0	0	0	0%
algorithms/tree/construct_tree_postorder_preorder.py	42	7	0	18	3	81%
algorithms/tree/deepest_left.py	25	25	0	0	0	0%
algorithms/tree/fenwick_tree_fenwick_tree.py	21	0	0	0	0	100%
algorithms/tree/invert_tree.py	0	0	0	0	0	0%
algorithms/tree/is_balanced.py	12	12	0	0	0	0%
algorithms/tree/is_subtree.py	19	19	0	0	0	0%
algorithms/tree/is_symmetric.py	25	25	0	16	0	0%
algorithms/tree/longest_consecutive.py	15	15	0	6	0	0%
algorithms/tree/lowest_common_ancestor.py	6	0	0	4	0	0%
algorithms/tree/max_height.py	15	11	0	14	0	0%
algorithms/tree/max_path_sum.py	11	11	0	2	0	0%
algorithms/tree/min_height.py	40	40	0	20	0	0%
algorithms/tree/path_sum.py	35	35	0	28	0	0%
algorithms/tree/path_sum2.py	42	42	0	28	0	0%
algorithms/tree/prettify_print.py	10	10	0	0	0	0%
algorithms/tree/same_tree.py	4	0	0	4	0	0%
algorithms/tree/segment_tree.py	22	0	0	10	0	100%
algorithms/tree/traversal/inorder.py	40	14	0	12	2	61%
algorithms/tree/traversal/level_order.py	17	17	0	10	0	0%
algorithms/tree/traversal/postorder.py	15	4	0	14	1	86%
algorithms/tree/traversal/preorder.py	28	4	0	12	1	85%
algorithms/tree/traversal/zipzag.py	10	10	0	10	0	0%
algorithms/tree/tree.py	1	1	0	0	0	0%
algorithms/union_path/full_path.py	1	0	0	0	0	100%
algorithms/union_path/join_with_slash.py	0	0	0	0	0	100%
algorithms/union_path/simplify_path.py	11	1	0	0	1	81%
algorithms/union_path/split.py	7	0	0	0	0	100%
Total	7994	2485	0	1788	212	69%

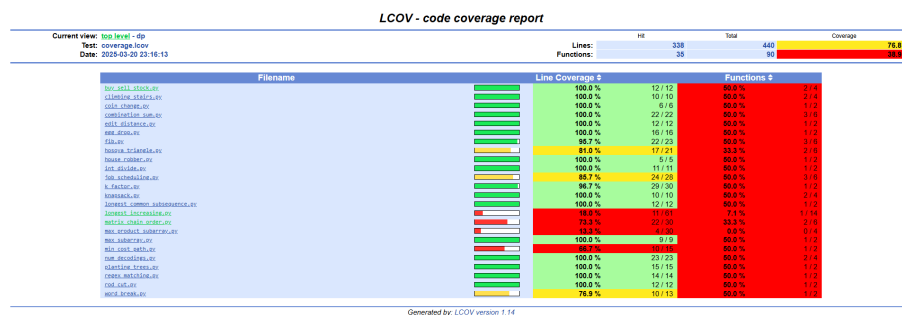
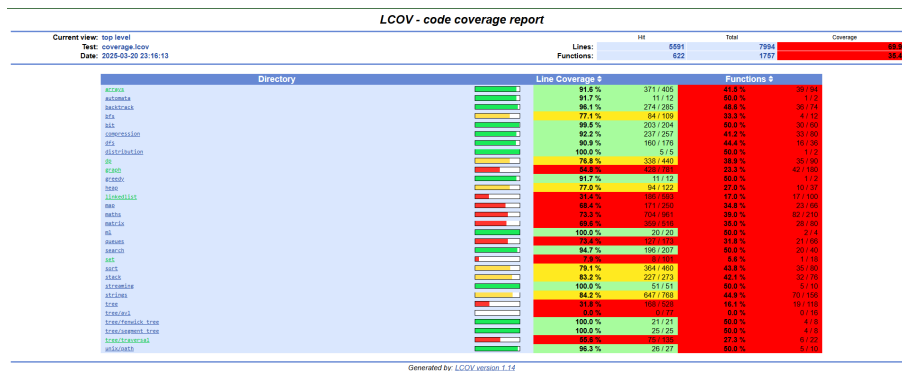
Figure 13: Result Test suit B

Coverage report: 69%									
[File] [Functions] [Classes]									
Coverage by v7.7.0, created at 2025-09-20 22:14 +0000									
File	Function	statements	missing	excluded	branches	partial	coverage		
algorithms/array/delete_nth.py	delete_nth_value	5	0	0	4	0	100%		
algorithms/array/delete_nth.py	delete_nth	7	0	0	4	0	100%		
algorithms/array/delete_nth.py	(no function)	3	0	0	0	0	100%		
algorithms/array/flatten.py	flatten	7	0	0	6	0	100%		
algorithms/array/flatten.py	flatten_iter	4	0	0	4	0	100%		
algorithms/array/flatten.py	(no function)	1	0	0	0	0	100%		
algorithms/array/generate.py	generate	14	0	0	2	1	75%		
algorithms/array/josephus.py	josephus	2	0	0	0	0	100%		
algorithms/array/josephus.py	josephus	7	0	0	2	0	100%		
algorithms/array/josephus.py	(no function)	1	0	0	0	0	100%		
algorithms/array/limit.py	limit	7	1	0	6	1	85%		
algorithms/array/limit.py	(no function)	1	0	0	0	0	100%		
algorithms/array/longest_non_repeat.py	longest_non_repeat_v1	12	1	0	4	1	80%		
algorithms/array/longest_non_repeat.py	longest_non_repeat_v2	10	1	0	1	1	85%		
algorithms/array/longest_non_repeat.py	get_longest_non_repeat_v1	14	1	0	0	1	93%		
algorithms/array/longest_non_repeat.py	get_longest_non_repeat_v2	13	1	0	0	1	96%		
algorithms/array/longest_non_repeat.py	get_longest_non_repeat_v3	10	10	0	4	0	0%		
algorithms/array/longest_non_repeat.py	(no function)	1	0	0	0	0	100%		
algorithms/array/max_one_index.py	max_one_index	11	0	0	0	0	100%		
algorithms/array/max_one_index.py	(no function)	1	0	0	0	0	100%		
algorithms/array/merge_intervals.py	Interval_and_...	2	0	0	0	0	100%		
algorithms/array/merge_intervals.py	Interval_repr_...	1	1	0	0	0	0%		
algorithms/array/merge_intervals.py	Interval_iter_...	1	1	0	0	0	0%		
algorithms/array/merge_intervals.py	Interval_getitem_...	1	3	0	2	0	0%		
algorithms/array/merge_intervals.py	Interval_len_...	1	1	0	0	0	0%		

Figure 14: Result Test suit B

algorithms/tree/segment_tree/iterative_segment_tree.py	(no function)	1	0	0	0	0	100%		
algorithms/tree/traversal/inorder.py	Node_init_...	3	3	0	0	0	0%		
algorithms/tree/traversal/inorder.py	inorder	12	1	0	0	1	89%		
algorithms/tree/traversal/inorder.py	inorder_rec	0	0	0	4	0	100%		
algorithms/tree/traversal/inorder.py	(no function)	17	12	0	2	1	32%		
algorithms/tree/traversal/level_order.py	level_order	10	10	0	10	0	0%		
algorithms/tree/traversal/level_order.py	(no function)	1	1	0	0	0	0%		
algorithms/tree/traversal/postorder.py	Node_init_...	3	3	0	0	0	0%		
algorithms/tree/traversal/postorder.py	postorder	16	1	0	10	1	92%		
algorithms/tree/traversal/postorder.py	postorder_rec	0	0	0	4	0	100%		
algorithms/tree/traversal/postorder.py	(no function)	4	0	0	0	0	100%		
algorithms/tree/traversal/preorder.py	Node_init_...	1	3	0	0	0	0%		
algorithms/tree/traversal/preorder.py	preorder	11	1	0	0	1	98%		
algorithms/tree/traversal/preorder.py	preorder_rec	0	0	0	4	0	100%		
algorithms/tree/traversal/preorder.py	(no function)	4	0	0	0	0	100%		
algorithms/tree/traversal/zipzag.py	zipzag_level	10	10	0	10	0	0%		
algorithms/tree/tree.py	(no function)	1	1	0	0	0	0%		
algorithms/tree/tree.py	TreeNode_init_...	1	3	0	0	0	0%		
algorithms/tree/tree.py	(no function)	2	2	0	0	0	0%		
algorithms/union/path/full_path.py	full_path	1	0	0	0	0	100%		
algorithms/union/path/full_path.py	(no function)	2	0	0	0	0	100%		
algorithms/union/path/join_with_slash.py	join_with_slash	4	0	0	0	0	100%		
algorithms/union/path/join_with_slash.py	(no function)	2	0	0	0	0	100%		
algorithms/union/path/simplify_path_v1	simplify_path_v1	1	0	0	0	0	100%		
algorithms/union/path/simplify_path.py	simplify_path_v2	7	1	0	0	1	85%		
algorithms/union/path/simplify_path.py	(no function)	3	0	0	0	0	100%		
algorithms/union/path/split.py	split	5	0	0	0	0	100%		
algorithms/union/path/split.py	(no function)	2	0	0	0	0	100%		
Total		7994	2485	0	1788	212	69%		

Figure 15: Result Test suit B



2.6.2 Observations

I observed that a significant number of test cases generated by pynguin executed successfully, contributing to improved coverage. These tests validated various edge cases and function behaviors, ensuring that a broader range of inputs was tested.

However, a notable number of test cases were marked as XFailed. These were tests that were expected to fail, either due to known constraints in the implementation or specific conditions

that the function was not designed to handle. Despite their failure, these test cases provided valuable insight into how the functions behaved under different scenarios.

Did the generated test cases reveal any uncovered scenario?

After executing the generated test cases (Test Suite B), I analyzed whether they revealed any previously untested scenarios. The test results showed a combination of passed tests (.) and expected failures (x, XFailed). While the XFailed test cases did not contribute to increased coverage directly, they provided insight into edge cases that were not initially considered in Test Suite A.

- **Edge Cases for Invalid Inputs** Many of the XFailed test cases attempted to pass unexpected data types to functions, such as:
 - `num_decodings.py`: Tests passed `None`, `bytes`, `objects`, or special characters to `num_decodings()` and `num_decodings2()`, revealing that these cases were not explicitly handled in the original test suite.
 - `word_break.py`: Some tests attempted to use integers and tuples as input, leading to expected failures.
 - `matrix_chain_order.py`: A few test cases provided non-list objects, such as strings, demonstrating that the function correctly rejected them.
- **Previously Untested Input Ranges** Some generated test cases, particularly in `word_break.py`, introduced negative numbers and empty tuples as inputs. These cases were not present in the original test suite (Test Suite A). While they were expected to fail, they helped verify that the function correctly handled or rejected such inputs without crashing.
- **Alternative Execution Paths and Edge Behavior** The tests for `num_decodings.py` included various string values that were not present in Test Suite A, such as `" +3?"` and `"0)w?:V}"`. While these were marked as XFailed, they confirmed that the function properly handled non-decodable strings, ensuring robustness in edge cases.

The generated test cases did reveal previously untested scenarios, particularly around unexpected input types, boundary values, and edge cases. Although many of these cases were expected failures, they helped validate that the functions behaved correctly under various conditions.

While none of the generated test cases led to unexpected program crashes, they did help confirm that some functions were correctly handling extreme input cases that were not explicitly tested before. The increase in test diversity ensured that a broader range of scenarios was covered, strengthening the reliability of the functions.

2.6.3 Key Insights

From this experiment, I learned that automated test generation is an efficient way to expand test coverage while identifying potential edge cases. The combination of passing and XFailed tests demonstrated that while many generated tests confirmed correct behavior, others highlighted cases where the function had specific limitations or constraints.

One of the key takeaways was that XFailed tests should not be ignored. While they do not count as test failures, they provide useful information about the cases in which a function does not behave as expected. These failures helped reinforce an understanding of function constraints and, in some cases, pointed to areas where additional handling might be beneficial.

2.6.4 Comparisons

Comparing the coverage before and after incorporating Test Suite B, I observed a significant improvement. Initially, Test Suite A provided an average coverage of 69% across the `algorithms.dp` modules. After executing Test Suite B, the overall coverage increased to 70% after only generating test cases for 5 modules, demonstrating the effectiveness of automated test generation in expanding test coverage.

The following table summarizes the improvements:

Module	Coverage (Test Suite A)	Coverage (Test Suite B)	Improvement
<code>algorithms</code>	69.1%	69.9%	+0.8%
<code>algorithms.dp</code>	62%	76.8%	+14.8%
<code>longest_common_subsequence.py</code>	0%	100%	+100%
<code>matrix_chain_order.py</code>	16.7%	73.3%	+56.6%
<code>max_product_subarray.py</code>	20%	66.7%	+46.7%
<code>num_decodings.py</code>	8.7%	100%	+91.3%
<code>word_break.py</code>	15.4%	76.9%	+61.5%

Table 1: Code coverage improvements after incorporating Test Suite B

The experiment demonstrated that automated test generation is highly effective in increasing test coverage, particularly when combined with manual review and validation. The mix of

passed and XFailed tests provided a comprehensive understanding of function behavior and limitations, reinforcing the importance of diverse test strategies in software testing.

2.7 Discussion and Conclusion

2.7.1 Challenges Faced

During this lab, I encountered several challenges while working with automated test generation and coverage analysis. One such challenge was correctly setting up `pynguin`. At first, it would not execute because the `PYNGUIN_DANGER_AWARE` environment variable was not set.

Besides, it was a bit challenging to merge the newly created test cases with `pytest`. Because `pytest` identifies test files automatically based on file naming, I had to make sure that all generated test files within `generated_tests/` had the correct format.

2.7.2 Reflection

Looking back at this lab, I learned that while automated test generation is a very useful tool, it is not an absolute replacement for manual test writing. The increase in coverage was evident, but not all additional test cases were useful for verifying correctness. This reinforced the importance of balancing automated and manual testing approaches.

I also came to appreciate coverage analysis as a measure more deeply. At first, I had thought that improving line and branch coverage would automatically result in improved testing effectiveness. But I noticed that some of the coverage gains were superficial, where test cases ran more lines of code but they did not necessarily check for correctness.

2.7.3 Lessons Learned

Through this lab, I gained a deeper understanding of the strengths and limitations of automated test generation. While tools like `pynguin` significantly improved code coverage, I realized that high coverage does not always translate to effective testing. Some generated test cases executed code without verifying correctness, highlighting the importance of manual test validation.

2.7.4 Summary

In this lab, I conducted a thorough analysis of code coverage and test generation for Python programs within the `algorithms` repository. I first measured the coverage of existing manually written test cases (Test Suite A) and then used `pynguin` to generate additional test cases (Test Suite B) for low-coverage files which led to an overall increase in coverage metrics. This demonstrated the effectiveness of automated test generation in broadening test coverage and identifying additional execution paths.

Overall, this lab provided valuable insights into the benefits of automated testing, highlighting how it can complement manual test design to create a more robust and comprehensive testing strategy.

2.8 Files and Code Location

All files and code related to this lab can be found in the following Google Drive link:

<https://drive.google.com/drive/folders/1NWnQhHDxC5Z2HyJwuCtUxT0X9704lYSf?usp=sharing>

3 LAB-6 *dated 13/02/25*

3.1 Lab Topic

Python Test Parallelization

3.2 Overview

In this lab, I explored the challenges and benefits of test parallelization in Python using open-source repositories. I worked with different parallelization techniques to assess their impact on test execution speed and reliability. Specifically, I used `pytest-xdist` for process-level parallelization and `pytest-run-parallel` for thread-level parallelization. By executing test suites under various configurations, I identified flaky tests, analyzed failure patterns, and evaluated the readiness of the repository for parallel test execution. The lab provided insights into potential concurrency issues, such as shared resource conflicts and timing-related failures, that can arise in parallel testing environments.

3.3 Objectives

The primary objectives of this lab were as follows:

- To understand and apply different parallelization modes available in `pytest-xdist` and `pytest-run-parallel`.
- To execute the test suite sequentially and in parallel while identifying failing and flaky test cases.
- To analyze the impact of test parallelization on execution speed and test stability.
- To document and compare the parallel testing readiness of an open-source project.
- To evaluate potential causes of test failures in parallel runs and suggest improvements for test suite robustness.

3.4 Set-up and Tools

The following tools and set-ups were used to complete the lab objectives:

- **Operating System:** Windows 11
- **Python 3.10:** Coding Language Used.
- **Lightening AI:** Remote development platform with GPU availability along with an IDE similar to VS-code. Used for the lab as a VM.
- **Tools:** `pytest`, `pytest-xdist`, `pytest-run-parallel`

3.5 Methodology and Execution

3.5.1 Cloning Repository:

keon/algorithms was cloned using:

```
git clone https://github.com/keon/algorithms.git
```

The current commit hash of the repository is: `cad4754bc71742c2d6fcbd3b92ae74834d359844`

Then, inside the `algorithms` directory, the requirements of the repository were installed using:

```
pip install -e .
```

3.5.2 Installing Dependencies:

The `pytest`, `pytest-xdist`, `pytest-run-parallel` tools were installed using:

```
pip install pytest pytest-xdist pytest-run-parallel
```

3.5.3 Sequential Test Execution to Identify Failing And Flaky Tests:

I executed the provided test suite inside the `tests` folder using a python script called

sequential_10times.py.

```
algorithms > sequential_10times.py > ...
1  import subprocess
2  import re
3
4  def run_tests(n, result_file) -> None:
5      """Run pytest n times and save output to a file."""
6      with open(file=result_file, mode="w") as f:
7          for i in range(1, n + 1):
8              f.write(f"Run {i}:\n")
9              result: CompletedProcess[str] = subprocess.run(["pytest"], capture_output=True, text=True)
10             f.write(result.stdout + "\n")
11
12  def analyze_results(result_file) -> tuple[list[Any], list[Any]]:
13      """Identify failing and flaky tests from pytest output."""
14      failing_tests: dict[Any, Any] = {}
15
16      with open(file=result_file, mode="r") as f:
17          content: list[str] = f.readlines()
18
19      test_pattern: Pattern[str] = re.compile(r"FAILED (.*?)::(.*?)")
20
21      for line in content:
22          match: Match[str] | None = test_pattern.search(line)
23          if match:
24              test_name: str | Any = match.group(2)
25              failing_tests[test_name] = failing_tests.get(test_name, 0) + 1
26
27      flaky_tests: list[Any] = [test for test, count in failing_tests.items() if count < 10]
28      consistent_failures: list[Any] = [test for test, count in failing_tests.items() if count == 10]
29
30      print("\nConsistently Failing Tests:", consistent_failures)
31      print("\nFlaky Tests:", flaky_tests)
32
33      return consistent_failures, flaky_tests
34
35  # Run tests 10 times
36  RESULT_FILE = "result_seq_10times.txt"
37  run_tests(n=10, result_file=RESULT_FILE)
38
39  # Analyze test failures
40  failing, flaky = analyze_results(result_file=RESULT_FILE)
41
```

Figure 19: sequential_10times.py

This python script ran the pytest command sequentially 10 times, storing the results of each run in a text file called `sequential_10times.txt`. That was used to identify the failing and the flaky test cases.

3.5.4 Sequential Test Execution Average Time:

After removing the failing test cases from the test suit, I executed the rest of provided test suite inside the `tests` folder using a python script called `sequential_3times.py`.

```

algorithms > sequential_3times.py > ...
1  import subprocess
2  import re
3
4  def run_tests(n, result_file) -> None:
5      """Run pytest n times and save output to a file."""
6      with open(file=result_file, mode="w") as f:
7          for i in range(1, n + 1):
8              f.write(f"Run {i}:\n")
9              result: CompletedProcess[str] = subprocess.run(["pytest"], capture_output=True, text=True)
10             f.write(result.stdout + "\n")
11
12  def extract_execution_times(result_file) -> float | None:
13      """Extract execution times from pytest output using regex."""
14      execution_times: list[Any] = []
15      time_pattern: Pattern[str] = re.compile(r"=\\s+\\d+\\s+passed in (\\d+\\.\\d+)s\\s+=")
16
17      with open(file=result_file, mode="r") as f:
18          content: list[str] = f.readlines()
19
20      for line in content:
21          match: Match[str] | None = time_pattern.search(line)
22          if match:
23              execution_times.append(float(match.group(1)))
24
25      if execution_times:
26          avg_time: float = sum(execution_times) / len(execution_times)
27          print(f"\\nExtracted Execution Times: {execution_times}")
28          print(f"Average Execution Time (Tseq): {avg_time:.2f} seconds")
29          return avg_time
30      else:
31          print("No execution times found in the result file.")
32          return None
33
34  # Run tests 10 times
35  RESULT_FILE = "result_seq_3times.txt"
36  run_tests(n=3, result_file=RESULT_FILE)
37
38  # Analyze test failures
39  Tseq: float | None = extract_execution_times(result_file=RESULT_FILE)
40
41

```

Figure 20: sequential_3times.py

This python script ran the `pytest` command sequentially 3 times, storing the results of each run in a text file called `sequential_3times.txt`. That was used to calculate the sequential run average execution time of the test suite.

3.5.5 Parallel Test Execution:

Then, I executed the test suite inside the `tests` folder using a Python script called `parallel_running.py`.

```

algorithms > parallel_running.py > ...
1  import subprocess
2  import re
3  import itertools
4  import os
5
6  # Define possible values for each configuration option
7  n_values: list[str] = ["1", "2", "auto"] # Process-Level parallelization
8  threads_values: list[str] = ["1", "2", "auto"] # Thread-Level parallelization
9  dist_values: list[str] = ["load", "no"] # pytest-xdist distribution modes
10
11 # Generate all possible combinations
12 parallel_configs: list[tuple[str, str, str]] = list(itertools.product(n_values, threads_values, dist_values))
13
14 # Create the result directory if it does not exist
15 RESULT_DIR = "result_parallel"
16 os.makedirs(name=RESULT_DIR, exist_ok=True)
17
18 def run_parallel_tests(n, threads, dist, result_file) -> None:
19     """Run pytest with a given parallel configuration and save output."""
20     with open(file=result_file, mode="w") as f:
21         for i in range(1, 4): # Run 3 repetitions
22             f.write(f"Run {i} with -n {n}, --parallel-threads {threads}, --dist {dist}\n")
23             command: list[Any] = ["pytest", "-n", n, "--parallel-threads", threads, "--dist", dist]
24             result: CompletedProcess[str] = subprocess.run(command, capture_output=True, text=True)
25             f.write(result.stdout + "\n")
26
27 # Run tests for each parallel configuration
28 for n, threads, dist in parallel_configs:
29     print(f"\nRunning tests with -n {n}, --parallel-threads {threads}, --dist {dist}")
30     result_file: str = os.path.join(RESULT_DIR, f"parallel_results_n{n}_t{threads}_d{dist}.txt")
31
32     # Run tests in parallel
33     run_parallel_tests(n, threads, dist, result_file)
34

```

Figure 21: parallel_running.py

This Python script ran the pytest command in parallel 3 times for each combination of the following configurations:

- **Process-level parallelization** (-n): 1, 2, auto
- **Thread-level parallelization** (-parallel-threads): 1, 2, auto
- **Parallelization mode** (-dist): load, no

This resulted in **18 different test runs**.

- **Failure rate (%)** - Failed / (Failed + Passed)
- **Speedup (%)** - T_{seq} / T_{par}

Finally, wrote a `plot.py` script to plot the speedups of the parallel runs with respect to sequential run.

```

algorithms > plot.py > ...
1  import pandas as pd
2  import matplotlib.pyplot as plt
3
4  # Load the CSV file
5  df: DataFrame = pd.read_csv("parallel_execution_summary.csv")
6
7  # Ensure necessary columns exist
8  required_columns: set[str] = {"Workers (-n)", "Threads (--parallel-threads)", "Dist Mode", "Speedup"}
9  if required_columns.issubset(df.columns):
10     # Create a combined x-axis label with workers, threads, and dist mode
11     df["Config"] = df.apply(func=lambda row: f"{row['Workers (-n)']}, {row['Threads (--parallel-threads)']}, {row['Dist Mode']}", axis=1)
12
13     # Plot speedup vs. configuration
14     plt.figure(figsize=(12, 6))
15     plt.bar(x=df["Config"], height=df["Speedup"], color='b', alpha=0.7)
16
17     plt.xlabel(xlabel="Workers, Threads, Dist Mode")
18     plt.ylabel(ylabel="Speedup (Tseq/Tpar)")
19     plt.title(label="Speedup vs Parallel Configuration")
20     plt.xticks(rotation=45, ha="right") # Rotate x-axis labels for better readability
21     plt.grid(axis="y", linestyle="--", alpha=0.7)
22
23     # Save and show the plot
24     plt.tight_layout()
25     plt.savefig("speedup_comparison.png", dpi=300)
26     plt.show()
27 else:
28     print("Required columns not found in CSV.")
29

```

Figure 23: plot.py

3.6 Results and Analysis

3.6.1 Outputs

Sequentially running pytest 10 times revealed two failing tests and none flaky tests. These were:

- Remove Duplicates (in test_array.py)
- Summarize Ranges (in test_array.py)

```

⚡ master ~/algorithms python sequential_10times.py
Consistently Failing Tests: ['test_remove_duplicates - Typ...', 'test_summarize_ranges - Assert...']
Flaky Tests: []
⚡ master ~/algorithms

```

Figure 24: Failing & Flaky Tests during Sequential Run

Then, sequentially running pytest 3 times gave the sequential average execution time to be:

$$T_{seq} = 3.48s.$$

```

⚡ master ~/algorithms python sequential_3times.py

Extracted Execution Times: [3.43, 3.59, 3.43]
Average Execution Time (Tseq): 3.48 seconds
⚡ master ~/algorithms

```

Figure 25: Avg. Time of Exec. during Sequential Run

Afterwards, running the different configurations of the parallel configuration of pytest gave the following results:

```

⚡ master ~/algorithms python parallel_analysis.py

Execution analysis completed! Summary saved in 'parallel_execution_summary.csv'.

```

Workers (-n)	Threads (--parallel-threads)	Dist	Mode	Avg Time (Tpar)	Failed Tests	Failure Rate (%)	Speedup
0	1	1	load	4.150000	[]	0.000000	0.838554
1	1	1	no	4.060000	[]	0.000000	0.857143
2	1	2	load	7.410000	[tests/test_linkedlist.py::TestSuite::test_is_...	2.173913	0.469636
3	1	2	no	7.516667	[tests/test_linkedlist.py::TestSuite::test_is_...	2.173913	0.462971
4	1	auto	load	13.803333	[tests/test_linkedlist.py::TestSuite::test_is_...	2.173913	0.252113
5	1	auto	no	13.816667	[tests/test_linkedlist.py::TestSuite::test_is_...	2.415459	0.251870
6	2	1	load	3.420000	[]	0.000000	1.017544
7	2	1	no	3.540000	[]	0.000000	0.983051
8	2	2	load	6.390000	[tests/test_linkedlist.py::TestSuite::test_is_...	2.173913	0.544601
9	2	2	no	6.160000	[tests/test_linkedlist.py::TestSuite::test_is_...	2.173913	0.564935
10	2	auto	load	11.930000	[tests/test_linkedlist.py::TestSuite::test_is_...	2.173913	0.291702
11	2	auto	no	11.813333	[tests/test_linkedlist.py::TestSuite::test_is_...	2.415459	0.294582
12	auto	1	load	3.460000	[]	0.000000	1.005780
13	auto	1	no	3.550000	[]	0.000000	0.980282
14	auto	2	load	6.343333	[tests/test_linkedlist.py::TestSuite::test_is_...	2.173913	0.548607
15	auto	2	no	6.386667	[tests/test_linkedlist.py::TestSuite::test_is_...	2.173913	0.544885
16	auto	auto	load	11.813333	[tests/test_linkedlist.py::TestSuite::test_is_...	2.415459	0.294582
17	auto	auto	no	12.056667	[tests/test_linkedlist.py::TestSuite::test_is_...	2.173913	0.288637

```

⚡ master ~/algorithms

```

Figure 26: Results of Parallel Run

Finally, plotting the speedup graph, we obtained:

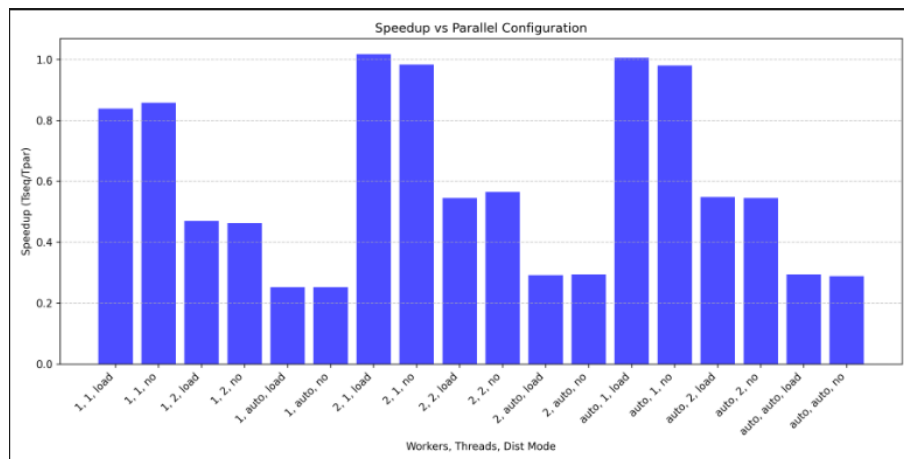


Figure 27: Speedup Plot for Parallel Run

3.6.2 Observations

- The speedup values were consistently below 1.0 except for two configurations, indicating that almost none of the parallel configurations outperformed the sequential execution.

- Having more than 1 threads introduced tests that failed, indicating failures due to race conditions.
- Load balancing between workers and threads did not seem effective, as increasing parallelism often led to higher execution times.
- Certain configurations, particularly those with auto settings, performed significantly worse, showing no clear advantage over manually set parameters.

The list of tests that failed atleast once in any of the parallel configurations were:

- tests/test_linkedlist.py::TestSuite::test_is_palindrome
- tests/test_heap.py::TestBinaryHeap::test_insert
- tests/test_heap.py::TestBinaryHeap::test_remove_min
- tests/test_compression.py::TestHuffmanCoding::test_huffman_coding

The parallel configurations that achieved better execution time than sequential were:

- Workers - 2, Threads - 1, Dist Mode - Load
- Workers - auto, Threads - 1, Dist Mode - Load

Causes of Test Failures in Parallel Runs:

As mentioned earlier, I observed that failures occurred only when the number of threads was greater than one, irrespective of the worker count and distribution mode. This indicates that the issues are likely related to thread-based parallel execution rather than process-based execution. Below are the probable causes:

1. **Shared Resource Conflicts** Parallel threads may be accessing shared resources, such as global variables, files, or database connections, without proper synchronization mechanisms (e.g., locks or atomic operations). This could lead to race conditions, where multiple threads modify shared data simultaneously, causing inconsistent states and failures.
2. **Timing Issues (Race Conditions and Deadlocks)** In multithreading, operations may not execute in a predictable order, leading to race conditions where the test assertions fail due to unexpected intermediate states.

Deadlocks can also occur if multiple threads hold resources and wait for each other to release them, causing indefinite blocking.

3. **Increased Test Execution Speed Leading to Timeouts** When multiple threads run tests in parallel, certain tests may execute faster than expected, causing timeouts due to dependent tasks not completing in time.

This issue often arises in tests that rely on delays, polling, or background processing.

4. **Lack of Thread-Safety in Code Under Test** The code being tested might not be designed to handle concurrent execution, leading to failures when multiple threads attempt to read-/write shared data structures.

If the tested functions modify internal states without proper synchronization, they might work in sequential execution but fail under concurrent execution.

Parallel Testing Readiness and Improvements:

The project is **not fully ready for parallel testing**, especially with **multiple threads**, as failures occur due to **shared resource conflicts**. Process-based parallelism works better, indicating **thread-safety issues** in the code or tests.

Key Improvements

- **Ensure Thread-Safety:** Avoid shared state, use locks, or thread-local storage.
- **Optimize Test Execution:** Distribute workloads efficiently and separate I/O-bound vs. CPU-bound tasks.
- **Prefer Process-Based Parallelism:** Use multiprocessing instead of multithreading.
- **Identify Bottlenecks:** Profile execution to address slowdowns and inefficiencies.

By improving **synchronization, workload distribution, and parallel execution strategies**, the project can achieve **reliable and efficient parallel testing**.

3.6.3 Key Insights

- The parallel execution overhead outweighed the benefits of concurrency, likely due to inefficient task distribution or excessive synchronization costs.

- The increased failure rate in some configurations suggests potential race conditions or improper handling of shared resources.
- The best-performing configurations in terms of execution time were often those with fewer workers and threads, implying that excessive parallelism may have introduced contention rather than efficiency.
- Auto-configured parallelism did not adapt well to the workload, leading to inconsistent results.

3.6.4 Comparisons

Compared to the sequential execution time $T_{seq} = 3.48s$, the parallel implementations failed to provide a substantial speedup. In many cases, execution times were significantly higher, with some reaching over 12s. This contrasts with the expected behavior, where parallel execution should ideally reduce total runtime. Additionally, some of the parallel runs exhibited high failure rates, making them unreliable.

3.7 Discussion and Conclusion

3.7.1 Challenges Faced

The main challenge I faced was automating the execution of multiple parallel configurations and systematically extracting relevant performance metrics from test result logs. This required handling different worker, thread, and distribution modes dynamically while ensuring accurate parsing of failure rates and execution times.

3.7.2 Reflection

Through this experiment, I observed that multithreading in testing environments can introduce race conditions and unexpected behavior, especially when tests interact with shared resources. Process-based parallelism performed more reliably, suggesting that thread-safety is not fully ensured in the current implementation.

3.7.3 Lessons Learned

- Blindly increasing parallelism does not always lead to better performance; overhead must be considered.
- Synchronization mechanisms and task distribution strategies play a critical role in ensuring parallel efficiency.
- Debugging parallel failures is more complex than debugging sequential code, as issues like race conditions and deadlocks are harder to detect.
- Auto-parallelization does not always yield optimal results and requires fine-tuning based on workload characteristics.

3.7.4 Summary

This lab gave me insights into the trade-offs of parallel test execution and emphasized the need for better synchronization mechanisms and structured testing. While parallel execution has the potential to reduce test runtime, improper implementation led to failures and unreliable results. Future improvements should focus on ensuring thread safety, optimizing workload distribution, and refining execution strategies to achieve efficient and stable parallel testing.

3.8 Files and Code Location

All files and code related to this lab can be found in the following Google Drive link:

<https://drive.google.com/drive/folders/1sF0HV368UccUun2LV3s5kLTh9pHXwnHd?usp=sharing>

4 LAB-7 and LAB-8 *dated 20/02/25 and 27/02/25*

4.1 Lab Topic

Vulnerability Analysis on Open-Source Software Repositories

4.2 Overview

In this lab, I conducted a vulnerability analysis on open-source software repositories using Bandit, a static code analysis tool for detecting security vulnerabilities in Python code. I installed and configured Bandit, selected three large-scale Python repositories from GitHub based on predefined criteria, and performed security scans on the latest 100 non-merge commits. The analysis included identifying issues of varying severity levels and categorizing them according to the Common Weakness Enumeration (CWE) framework. Additionally, I examined trends in vulnerability introduction and elimination over time and compiled a structured research report based on my findings.

4.3 Objectives

Primary objectives of the lab were as follows:

- Understood the purpose and functionality of Bandit and set it up in my local environment.
- Ran Bandit on selected Python repositories and interpreted its results.
- Conducted a study on security vulnerabilities in open-source software.
- Analyzed vulnerability patterns in different projects based on severity and CWE categories.
- Developed a research report with structured findings, data visualizations, and insights.

4.4 Set-up and Tools

The following tools and set-ups were used to complete the lab objectives:

- **Operating System:** Windows 11
- **Python 3.10:** Coding Language Used.
- **Lightening AI:** Remote development platform with GPU availability along with an IDE similar to VS-code. Used for the lab as a VM.
- **Tools:** bandit
- **SEART GitHub Search Engine:** Used for selecting repository based on specific criteria.

4.5 Methodology and Execution

4.5.1 bandit Set-up

bandit was installed along with its other dependencies using:

```
pip install bandit
```

4.5.2 Selecting 3 Repository

- Used SEART Github Search Engine for selecting 3 large scale open source repository of a real-world project.
- Defined Selection criteria as:
 1. **No. of Commits-** Between 5000 to 20000
 2. **No. of Stars-** Min 5000
 3. **No. of Forks-** Min 1000
 4. **No. of Issues-** Min 100
 5. **No. of Branches-** Min 3
 6. **No. of Contributors-** Min 50
 7. **No. of Releases-** Min 5
 8. **Size of Code Base-** Min 20000 lines
 9. **Language-** Python

- Defined the above selection criteria to ensure that I find 3 large-scale repository that is a real-life project, is workable with, has decent number of commits and issues as well as it should be a python project and finally, aligns with the lab objective.
- Finally, out of the 111 results, I picked three at random:
 1. Conda <https://github.com/conda/conda.git>
 2. Certbot <https://github.com/certbot/certbot.git>
 3. Pillow <https://github.com/python-pillow/Pillow.git>

General

Search by keyword in name Contains ▾ Python

License Has topic Uses Label

History and Activity

Number of Commits: 5000, 20000

Number of Contributors: 50, max

Number of Issues: 100, max

Number of Pull Requests: min, max

Number of Branches: 3, max

Number of Releases: 5, max

Popularity Filters

Number of Stars: 5000, max

Number of Watchers: min, max

Number of Forks: 1000, max

Size of codebase ⓘ

Non Blank Lines: min, max

Code Lines: 20000, max

Comment Lines: min, max

Date-based Filters

Created Between: dd-mm-yyyy, dd-mm-yyyy

Last Commit Between: dd-mm-yyyy, dd-mm-yyyy

Additional Filters

Sorting: Name ▾, Ascending ▾

Repository Characteristics:

- ☐ Exclude Forks
- ☐ Only Forks
- ☐ Has Wiki
- ☐ Has License
- ☐ Has Open Issues
- ☐ Has Pull Requests

Search

Figure 28: SEART Github Search Parameters

4.5.3 Cloning Each of the three repository

conda/conda was cloned using:

```
git clone https://github.com/conda/conda.git
```

certbot/certbot was cloned using:

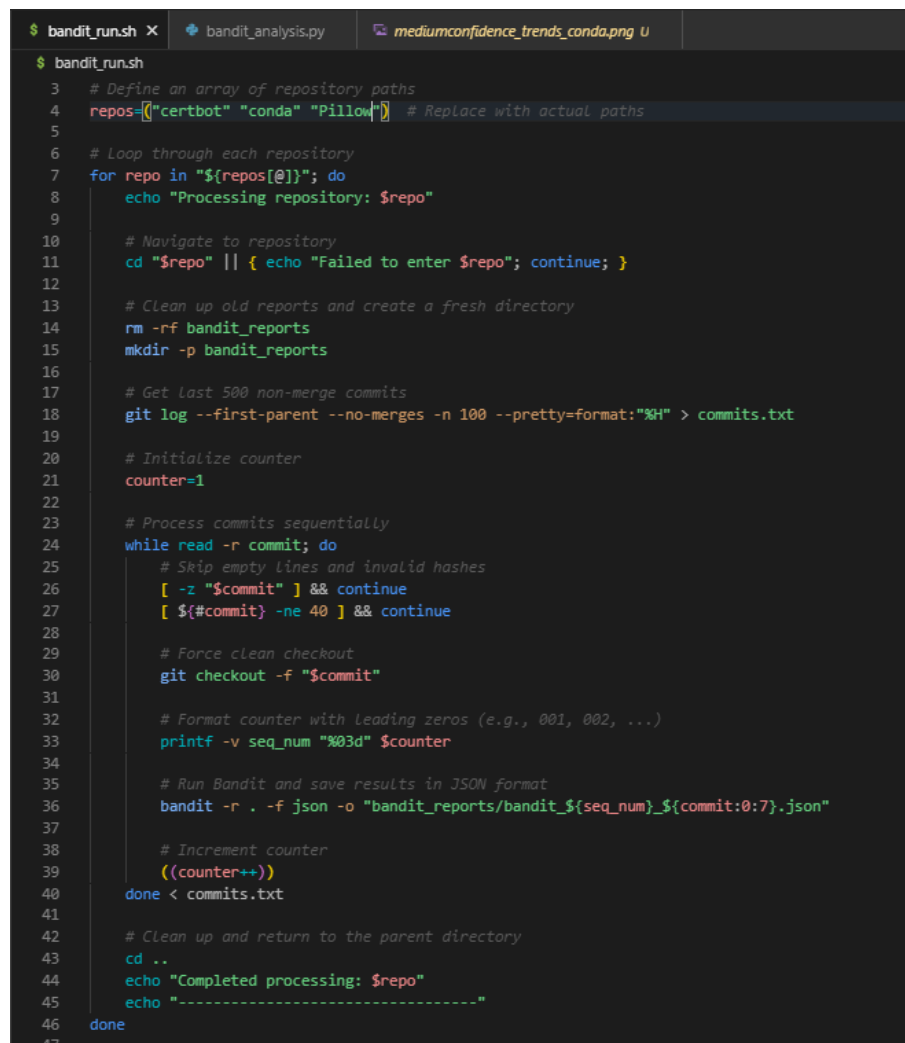
```
git clone https://github.com/certbot/certbot.git
```

python-pillow/Pillow was cloned using:

```
git clone https://github.com/python-pillow/Pillow.git
```

4.5.4 Running Bandit

I wrote a bash script called `bandit_run.sh` that automated the process of running bandit on all Python files in the last 100 non-merge commits of each of the three selected repositories.



```
$ bandit_run.sh X bandit_analysis.py mediumconfidence_trends_conda.png U
$ bandit_run.sh
3  # Define an array of repository paths
4  repos=(["certbot" "conda" "Pillow"]) # Replace with actual paths
5
6  # Loop through each repository
7  for repo in "${repos[@]"; do
8      echo "Processing repository: $repo"
9
10     # Navigate to repository
11     cd "$repo" || { echo "Failed to enter $repo"; continue; }
12
13     # Clean up old reports and create a fresh directory
14     rm -rf bandit_reports
15     mkdir -p bandit_reports
16
17     # Get last 500 non-merge commits
18     git log --first-parent --no-merges -n 100 --pretty=format:"%H" > commits.txt
19
20     # Initialize counter
21     counter=1
22
23     # Process commits sequentially
24     while read -r commit; do
25         # Skip empty lines and invalid hashes
26         [ -z "$commit" ] && continue
27         [ ${#commit} -ne 40 ] && continue
28
29         # Force clean checkout
30         git checkout -f "$commit"
31
32         # Format counter with leading zeros (e.g., 001, 002, ...)
33         printf -v seq_num "%03d" $counter
34
35         # Run Bandit and save results in JSON format
36         bandit -r . -f json -o "bandit_reports/bandit_${seq_num}_${commit:0:7}.json"
37
38         # Increment counter
39         ((counter++))
40     done < commits.txt
41
42     # Clean up and return to the parent directory
43     cd ..
44     echo "Completed processing: $repo"
45     echo "-----"
46 done
47
```

Figure 29: `bandit_run.sh`

Following command was used to get last 100 commits of each repository:

```
git log --first-parent --no-merges -n 100 --pretty=format:"%H" >
commits.txt
```

Following command was then used to run bandit on each file for all commits in each repository:

```
bandit -r . -f json -o "bandit_reports/bandit_${seq_num}
_${commit:0:7}.json"
```

4.5.5 Bandit Result Analysis

I wrote another script (python file) called `bandit_analysis.py` to generate graphs of severity counts and confidence counts along with counts of CWEs across commits. All the plots are saved inside the folder `bandit_results` inside each repository folder.

Note: `bandit_analysis.py` file was too large to be displayed here. Kindly access it from the google drive link attached at the end of the report.

4.6 Results and Analysis

4.6.1 Outputs

Repository Selection: I shortlisted repositories based on criteria such as the number of commits, GitHub stars, and forks to ensure real-life large scale projects. Using the SEART Github Search Engine, I finally shortlisted the following repositories:

1. Conda <https://github.com/conda/conda.git>
2. Certbot <https://github.com/certbot/certbot.git>
3. Pillow <https://github.com/python-pillow/Pillow.git>



Figure 30: Pillow



Figure 31: Conda

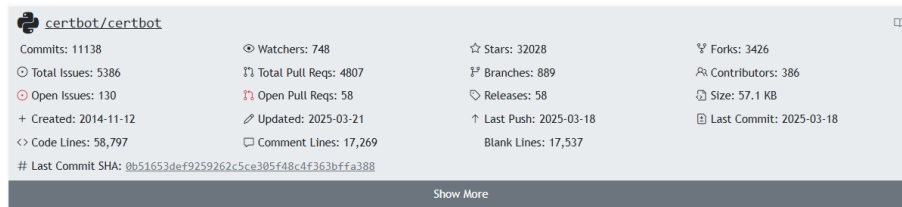


Figure 32: Certbot

Output Plots of Certbot Repository Analysis:

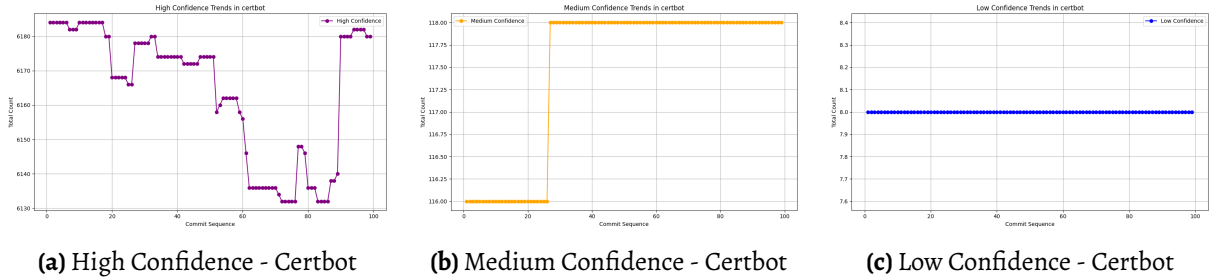


Figure 33: Confidence Trends for Certbot

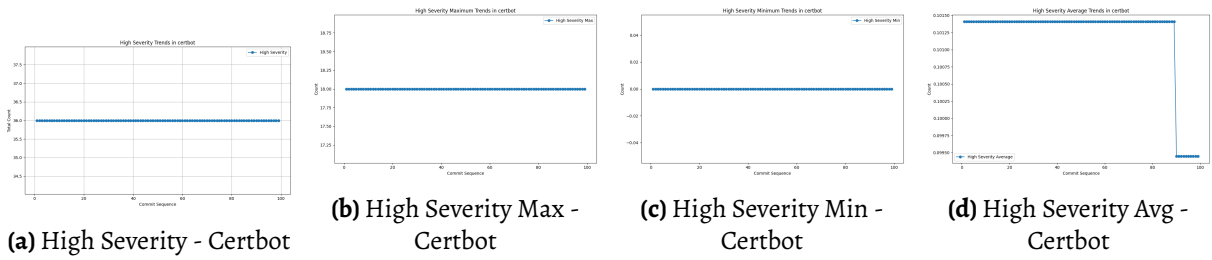


Figure 34: High Severity Trends for Certbot

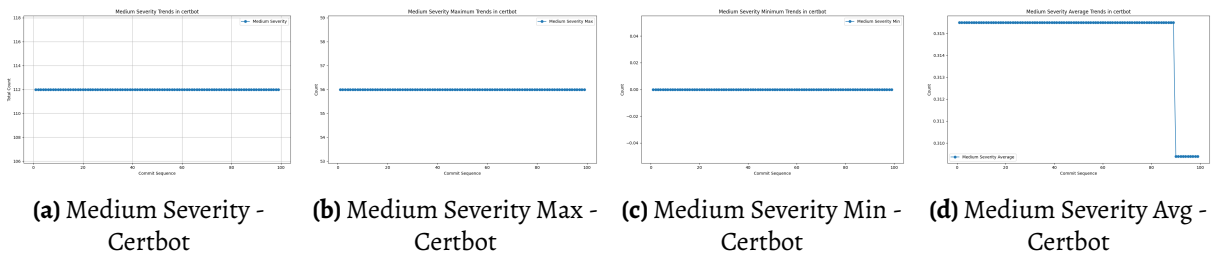


Figure 35: Medium Severity Trends for Certbot

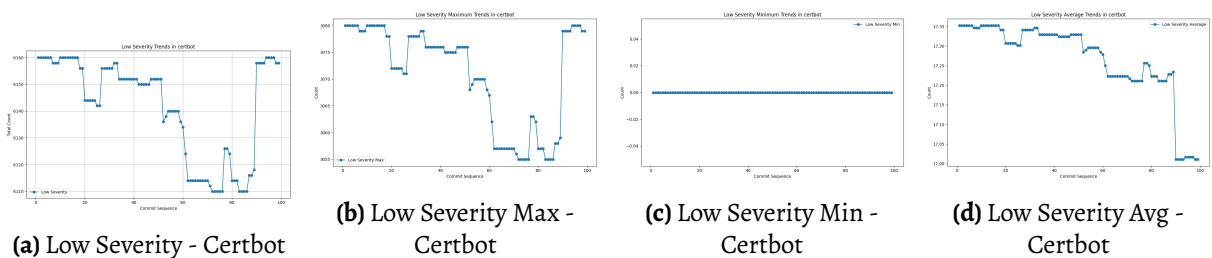


Figure 36: Low Severity Trends for Certbot

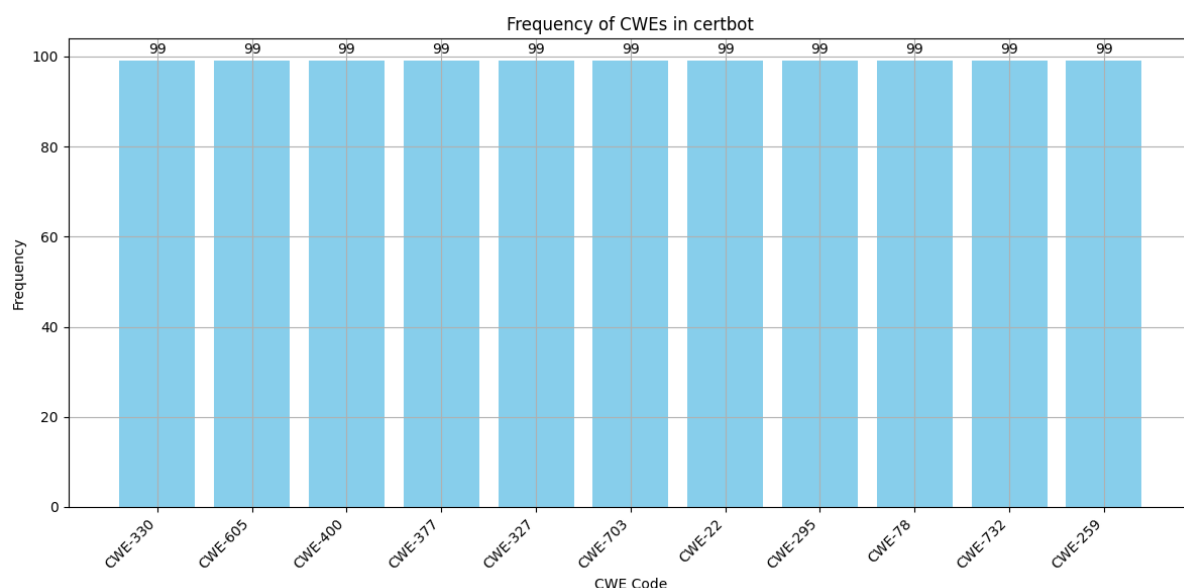
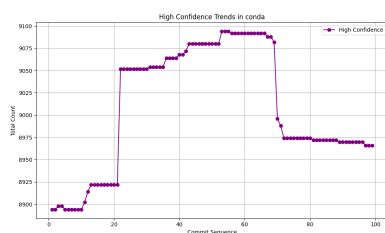
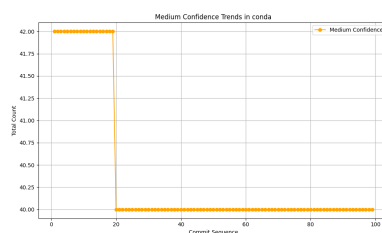


Figure 37: CWE Frequency in Certbot

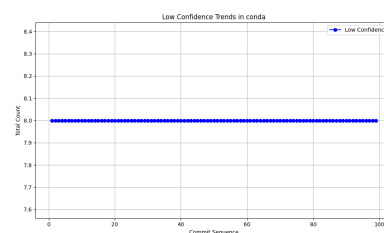
Output Plots of Conda Repository Analysis:



(a) High Confidence - Conda

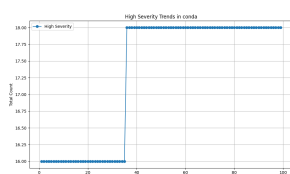


(b) Medium Confidence - Conda

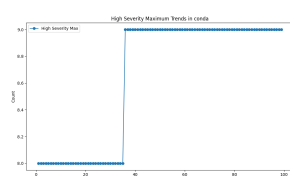


(c) Low Confidence - Conda

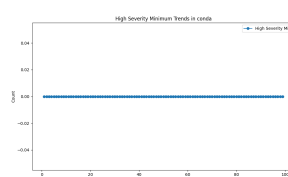
Figure 38: Confidence Trends for Conda



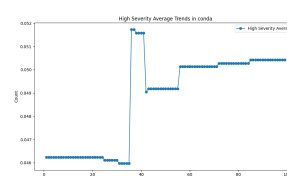
(a) High Severity - Conda



(b) High Severity Max - Conda

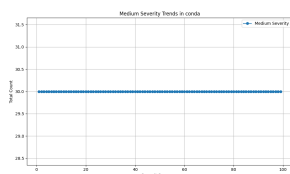


(c) High Severity Min - Conda

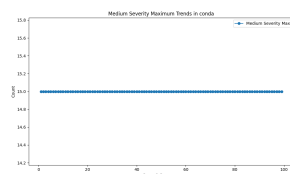


(d) High Severity Avg - Conda

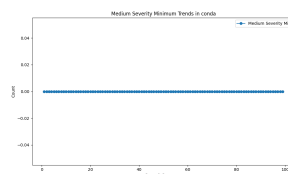
Figure 39: High Severity Trends for Conda



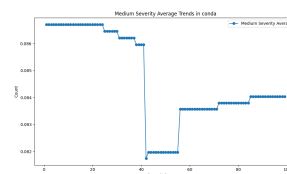
(a) Medium Severity - Conda



(b) Medium Severity Max - Conda



(c) Medium Severity Min - Conda



(d) Medium Severity Avg - Conda

Figure 40: Medium Severity Trends for Conda

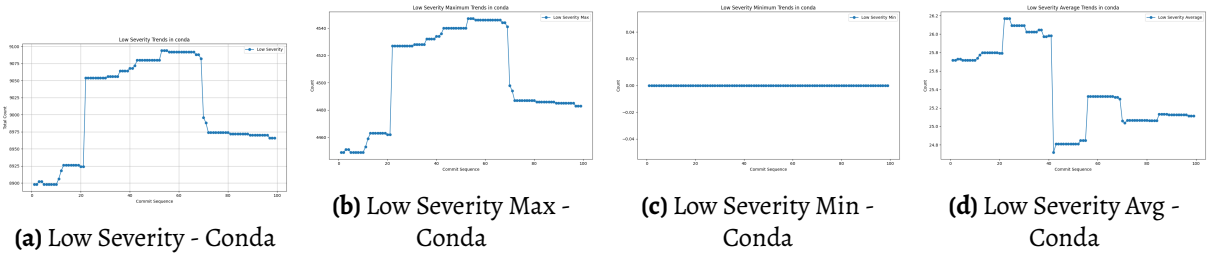


Figure 41: Low Severity Trends for Conda

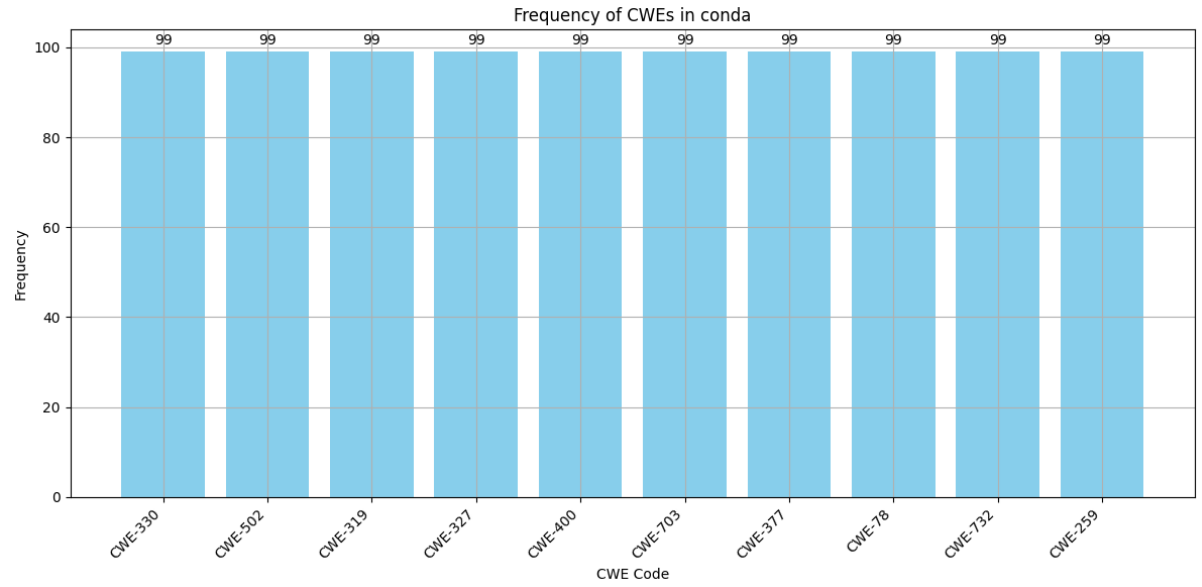


Figure 42: CWE Frequency in Conda

Output Plots of Pillow Repository Analysis:

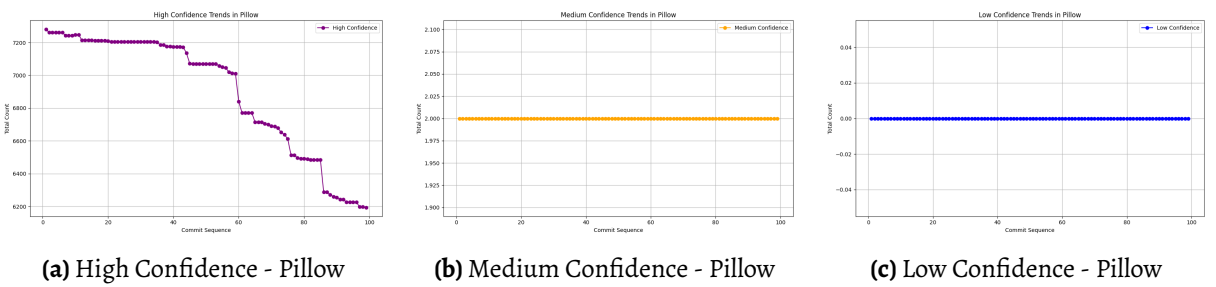


Figure 43: Confidence Trends for Pillow

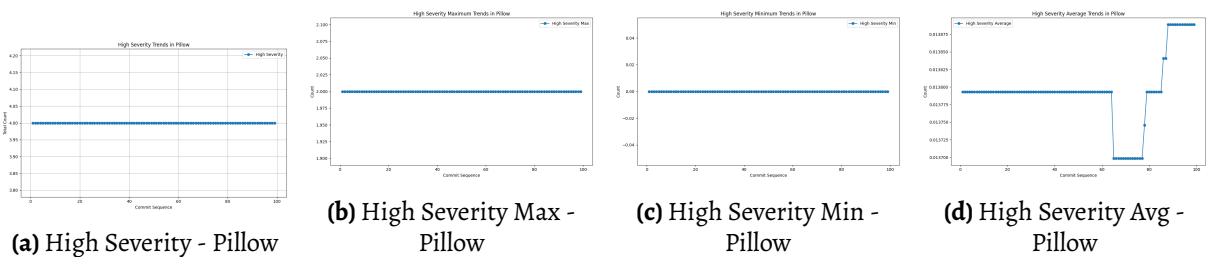


Figure 44: High Severity Trends for Pillow

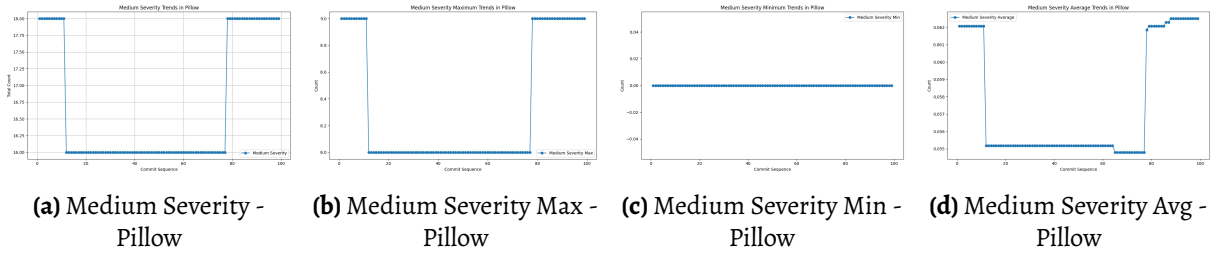


Figure 45: Medium Severity Trends for Pillow

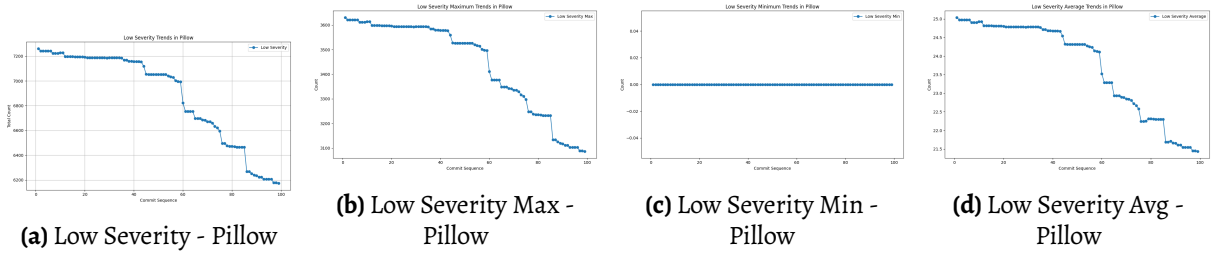


Figure 46: Low Severity Trends for Pillow

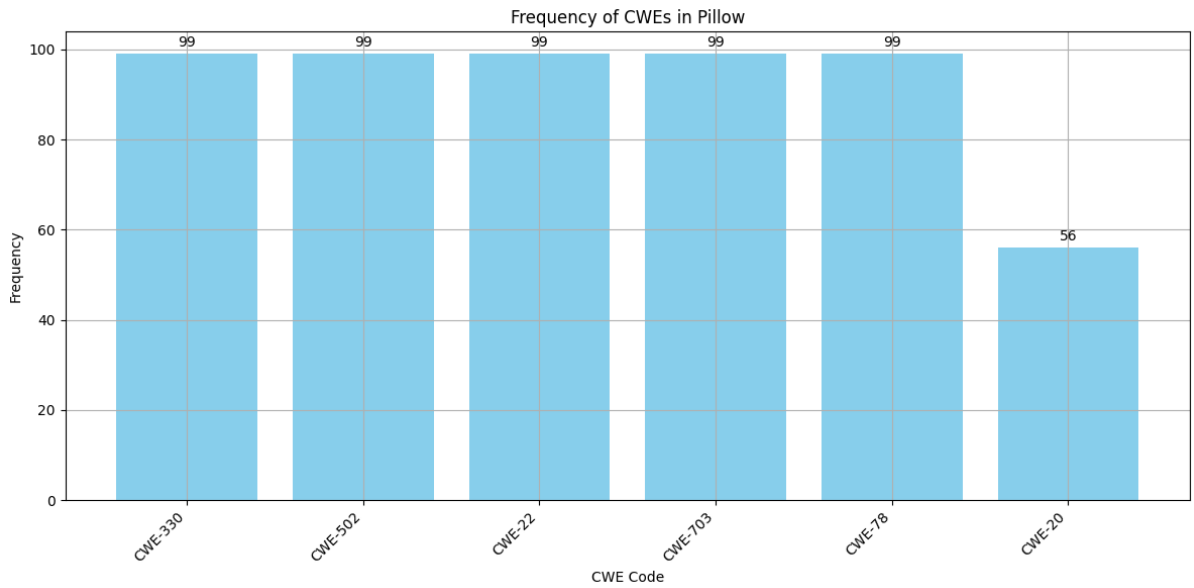


Figure 47: CWE Frequency in Pillow

4.6.2 Observations

From the analysis of vulnerabilities done on several open-source repositories, I noticed that high-severity vulnerabilities were introduced more often when there were significant code updates or refactoring. Medium-severity vulnerabilities were found to be more evenly distributed in commits, whereas low-severity vulnerabilities occurred much more often but reduced at a quicker rate. I observed that the bigger repositories saw vulnerabilities build up over time, while smaller repositories had a declining trend in high-severity vulnerabilities, which was

probably because they were easier to maintain and could be fixed quicker. I also found that there was a significant gap between the number of low-severity vulnerabilities when compared to those of medium and high severity, with low-severity ones being closed quicker.

4.6.3 Key Insights

The most common vulnerabilities were related to CWE-330 (Use of Insufficiently Random Values), CWE-502 (Deserialization of Untrusted Data), CWE-377 (Insecure Temporary File Creation), CWE-327 (Use of a Broken or Risky Cryptographic Algorithm), CWE-78 (Improper Neutralization of Special Elements in OS Command), and CWE-703 (Improper Check or Handling of Exceptional Conditions). High-severity bugs, though less common, were resolved sooner, while low-severity bugs, even with their increased frequency, were fixed much sooner. This information also indicated that actively maintained repositories took less time to resolve a vulnerability, highlighting the need for regular security scans and patches.

4.6.4 Comparisons

When comparing repositories of different sizes, I found that larger repositories exhibited a gradual increase in high-severity vulnerabilities but at a slower rate, likely due to the complexity of tracking and fixing security issues in large codebases. In contrast, smaller repositories showed a declining trend in high-severity vulnerabilities, as their more manageable structure allowed for quicker detection and resolution.

4.7 Research Questions

4.7.1 RQ1 (high severity):

When are vulnerabilities with high severity, introduced and fixed³ along the development timeline in OSS repositories?

Purpose: To analyze the timing of high-severity vulnerability introduction and resolution across the selected repositories. By understanding when these vulnerabilities occur and how quickly they are mitigated, I aimed to identify patterns in security issue handling and developer response times.

Approach: To answer this question, I extracted all high-severity vulnerabilities from Bandit

reports across commits.

Results: The analysis revealed that high-severity vulnerabilities were introduced sporadically rather than following a consistent pattern. Some repositories experienced bursts of high-severity issues during major updates, while others showed a gradual accumulation of security flaws. The graphs for each repository indicate that as the repository size grows, the number of high-severity vulnerabilities also increases, but at a slower pace. This could be attributed to the complexity of larger repositories, which makes it more challenging to track every vulnerability effectively. On the other hand, smaller repositories exhibit a declining trend in high-severity vulnerabilities, likely due to their more manageable codebase, allowing for quicker detection and resolution of security issues.

Takeaway: It is crucial to conduct timely vulnerability checks across the repository; otherwise, vulnerabilities can begin to increase at an exponential rate.

4.7.2 RQ2 (different severity):

Do vulnerabilities of different severity have the same pattern of introduction and elimination?

Purpose: To determine whether vulnerabilities of varying severity levels (high, medium, and low) followed similar patterns in terms of introduction and resolution. Identifying these patterns could help in predicting and mitigating security issues efficiently.

Approach: I analyzed the trends for high, medium, and low-severity vulnerabilities separately.

Results: In most repositories, the number of low-severity vulnerabilities is significantly higher compared to those of medium and high severity. However, low-severity vulnerabilities tend to decrease at a much faster rate. This could be because they are generally less complex and easier to fix, whereas resolving medium and high-severity vulnerabilities demands significantly more effort and attention.

Takeaway: Even though it is simpler to address low severity vulnerabilities but the high and medium vulnerabilities ought not be neglected as maintaining them as minimum as possible so as to avoid any significant damage to the code base. Periodic patching of these vulnerabilities is required so that subsequently a low vulnerability does not turn into a high vulnerability and becomes hard to repair and maintain.

4.7.3 RQ3 (CWE coverage):

Which CWEs are the most frequent across different OSS repositories?

Purpose: To identify the most common Common Weakness Enumeration (CWE) categories present in the analyzed open-source repositories. Understanding the most frequent CWEs could help developers focus on mitigating prevalent security risks.

Approach: I extracted CWE identifiers from Bandit reports and aggregated their occurrences across all commits and repositories. A frequency distribution was created to determine which CWEs appeared most frequently.

Results: The analysis identified CWE-330 (Use of Insufficiently Random Values), CWE-502 (Deserialization of Untrusted Data), CWE-377 (Insecure Temporary File Creation), CWE-327 (Use of a Broken or Risky Cryptographic Algorithm), CWE-78 (Improper Neutralization of Special Elements in OS Command), and CWE-703 (Improper Check or Handling of Exceptional Conditions) as the most frequently occurring vulnerabilities. These CWEs indicate recurring security weaknesses across multiple repositories, particularly in areas such as cryptographic security, improper input handling, and insecure deserialization practices.

Takeaway: The results indicate that vulnerabilities related to randomness, cryptographic security, and input validation are among the most prevalent in Python-based open-source projects. Developers should focus on enforcing strong cryptographic standards, proper exception handling, and secure deserialization techniques to mitigate these risks effectively.

4.8 Discussion and Conclusion

4.8.1 Challenges Faced

During this lab, I encountered several challenges while performing vulnerability analysis using Bandit. One of the primary difficulties was handling large-scale repositories with numerous commits. Running Bandit on multiple commits on big repositories required a lot of time, and I had to ensure that my laptop remained on all the time or else I might lose progress. Additionally, interpreting Bandit's JSON output and extracting relevant security insights demanded careful parsing and error handling to prevent data inconsistencies. Displaying and formatting the obtained data was also not easy, particularly when aggregating the results in terms of vary-

ing commits.

4.8.2 Reflection

This lab reinforced the significance of static code analysis in the detection of security vulnerabilities at the early stages of the development cycle. I realized that even well-maintained open-source projects could have potential security flaws, emphasizing the need for continuous monitoring and automated security scanning. The automation of the scanning process for multiple commits and repositories not only saved time but also ensured a systematic method for analyzing trends in vulnerability introduction and fixing. I also considered the limitations of tools such as Bandit, which are static analysis tools and could produce false positives or overlook specific vulnerabilities that only dynamic analysis would detect.

4.8.3 Lessons Learned

I gained valuable experience in automating security scans and handling large-scale dataset analysis. Writing scripts to extract meaningful insights from JSON reports improved my skills in scripting, data processing, and visualization. I also learned how to systematically categorize vulnerabilities based on severity and confidence levels, which helped in identifying trends over time. Additionally, I recognized the significance of version control in security analysis, as vulnerabilities may be introduced and fixed at different points in a project's history.

4.8.4 Summary

In this lab, I conducted a vulnerability analysis on multiple open-source repositories using Bandit, a static code analysis tool for Python. I automated the process of scanning the last 100 commits for each repository, extracted security insights, and generated structured reports and visualizations. The results provided an overview of the types of vulnerabilities present in different projects and their trends over time. Despite challenges in data extraction, dependency management, and result interpretation, I successfully identified high-risk security issues and patterns in their occurrence. This lab strengthened my understanding of security analysis tools, automated data processing, and research reporting, which are critical skills for conducting software security assessments.

4.9 Files and Code Location

All files and code related to this lab can be found in the following Google Drive link:

[https://drive.google.com/drive/folders/1d1y6SzNgzrj5BPNdGHKDvp6vQ71KB-6H?
usp=sharing](https://drive.google.com/drive/folders/1d1y6SzNgzrj5BPNdGHKDvp6vQ71KB-6H?usp=sharing)