CS202: Software Tools and Techniques

Lab 5 Report: Code Coverage Analysis and Test Generation

Jiya Bhavin Desai | Roll No. 22110107 | 06.02.2025

---

1. Introduction

Software testing is a very important part of ensuring the reliability and correctness of programs. One of the key points of testing is **code coverage analysis**, which helps evaluate how much of the source code is exercised by a given test suite. The focus of lab 5 was on measuring coverage metrics such as line coverage, branch coverage, and function coverage using automated testing tools and improving it using generated tests.

In this lab, we got to work with a dataset of Python programs (present in the algorithms repository on GitHub) and use coverage analysis tools to assess how thoroughly the code has been tested. The aim was to identify untested portions of the code (the portions left untested by the given test suite) and improve test coverage. This process helps understand the effectiveness of existing test cases, identifying gaps in testing, and demonstrates the benefits and limitations of automated vs. manually written tests. The following were the targeted outcomes:

- Differentiate between various types of code coverage.
- Utilize automated tools to measure and visualize test coverage and compare their coverage with the coverage of the given tests.
- Generate effective unit tests to improve code coverage.
- Analyze coverage reports to assess test effectiveness.

1.1 Setup

- Using the SET-IITGN-VM to ensure version compatibility with pytest and pynguin
- Installing the required dependencies: pytest, pynguin, pytest-cov for test coverage analysis

1.2 Tools

- Windows Command Prompt (on my local machine)
- Algorithms repository on GitHub (contains python files to be tested as well as those used for testing)
- pytest (for running tests)
- pytest-cov (for line/branch coverage analysis)
- coverage (for detailed coverage metrics)
- pynguin (for automated unit test generation)

2. Methodology and Execution

Cloning the algorithms repository:

- I used the SET-IITGN-VM for executing this lab, to ensure version compatibility with pytest and pynguin
- Next, I cloned the repository using the command:
  ```
  git clone https://github.com/keon/algorithms
  ```
  There was some issue with the algorithms directory within the root directory of the cloned repository, which was fixed by running the following command within the root directory:
  ```
  pip install -e .
  ```

About the repository: The GitHub repository **keon/algorithms** provides a comprehensive collection of Python implementations for various data structures and algorithms. It includes a wide range of examples, from basic data structures like arrays, linked lists, and trees, to complex algorithms such as dynamic programming, graph traversal, and sorting techniques. The repository is designed to be minimal and clean, making it an excellent resource for learning and understanding the fundamentals of algorithms in Python. It also supports contributions and includes a robust testing framework using pytest for ensuring the reliability of the code. While primarily focused on providing implementations of various algorithms, this dataset of programs can also serve as a valuable resource for analyzing code coverage, as explored in this lab assignment.

2.1 Running the given test suite:

- The following command executes all the tests present in the tests folder on all the python programs present in the algorithms directory, and creates a report with all information about the code coverage of each file by the given tests:
  ```
  pytest --cov=algorithms --cov-branch
  --cov-report=term-missing
  --cov-report=xml:coverage.xml
  --cov-report=html:coverage_html
  --cov-report=annotate:coverage_annotate >
  coverage_report.txt
  ```
- The report generated is stored both as a text file and an interactive web page in the htmlcov folder generated, where the coverage results for each file can be viewed as separate html documents, with index.html summarising the results for all the files in a single webpage.
- ***On running the command, a few test cases were failing.*** When I checked the files that were throwing the errors, I found that there was one syntax error (a missing comma) in the file test_array.py and a logical error (the test for removing duplicates from an array was using assert, but only provided one argument instead of two, raising the error). I fixed the comma error before proceeding further.

2.2 Generating new tests:

Pynguin works by performing static analysis of a module to understand its structure, generating test cases iteratively, executing them, measuring coverage, and optimizing the test suite. The primary command for running Pynguin is:

```
pynguin --project-path <path_to_project> --module-name
<module_name> [options]
```

after setting the pynguin environment by running the command:

```
export PYNGUIN DANGER AWARE=1
```

Key arguments include --project-path (specifies the project directory), --module-name (target module for testing), --output-path (where tests are

saved), and --coverage-metric (defines coverage type such as BRANCH or LINE).
I identified the files with very less coverage by the given test suite, and generated tests for 3 of them, which also ended up covering some other files. The modules for which I generated the test cases are:

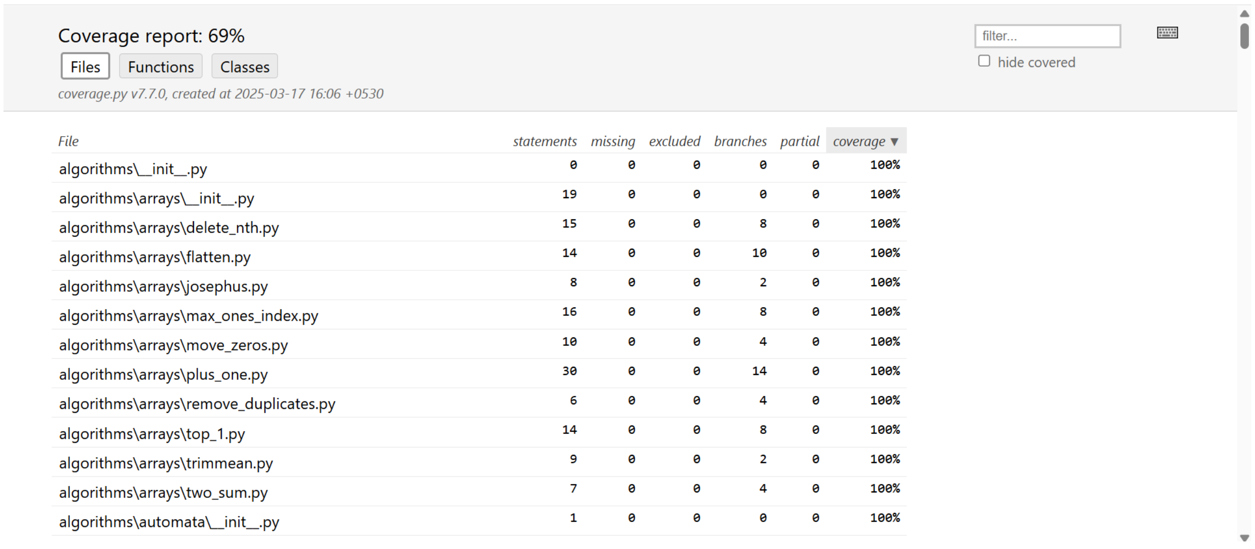
1. clone_graph.py (graph module)
   ```
   pynguin –project-path . –module-name
   algorithms.graph.clone_graph –output-path
   generated_tests
   ```
2. find_all_cliques.py (graph module)
   ```
   pynguin –project-path . –module-name
   algorithms.graph.find_all_cliques –output-path
   generated_tests
   ```
3. longest_common_subsequence.py (dynamic programming module)
   ```
   pynguin –project-path . –module-name
   algorithms.dp.longest_common_subsequence –output-path
   generated_tests
   ```

The coverage report of running those tests **individually** as well as **combined with the original test suite** have been presented in the results section.

3. Results and Analysis

3.1 Coverage report for the given test suite:

As mentioned above, the coverage report could be viewed in a browser as an interactive webpage. The files are shown in decreasing order of their coverage:



Coverage report: 69%

Files Functions Classes

coverage.py v7.7.0, created at 2025-03-17 16:06 +0530

filter...

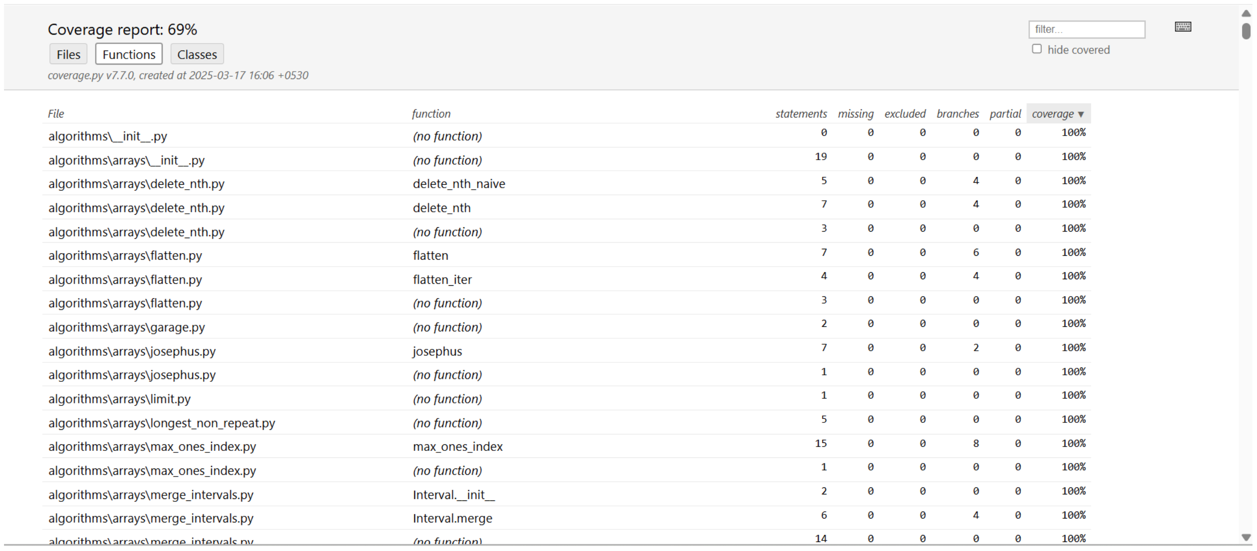
hide covered

File	statements	missing	excluded	branches	partial	coverage ▼
algorithms__init__.py	0	0	0	0	0	100%
algorithms\arrays__init__.py	19	0	0	0	0	100%
algorithms\arrays\delete_nth.py	15	0	0	8	0	100%
algorithms\arrays\flatten.py	14	0	0	10	0	100%
algorithms\arrays\josephus.py	8	0	0	2	0	100%
algorithms\arrays\max_ones_index.py	16	0	0	8	0	100%
algorithms\arrays\move_zeros.py	10	0	0	4	0	100%
algorithms\arrays\plus_one.py	30	0	0	14	0	100%
algorithms\arrays\remove_duplicates.py	6	0	0	4	0	100%
algorithms\arrays\top_1.py	14	0	0	8	0	100%
algorithms\arrays\trimmean.py	9	0	0	2	0	100%
algorithms\arrays\two_sum.py	7	0	0	4	0	100%
algorithms\automata__init__.py	1	0	0	0	0	100%

The following fields are present in the report:

1. File – The Python file being analyzed for code coverage.
2. Statements – The total number of executable statements in the file.
3. Missing – The number of statements that were not executed during testing.
4. Excluded – The number of statements explicitly excluded from coverage analysis.
5. Branches – The total number of conditional branches in the code.
6. Partial – The number of branches that were only partially executed.
7. Coverage – The percentage of statements executed, indicating how much of the file was covered by tests.

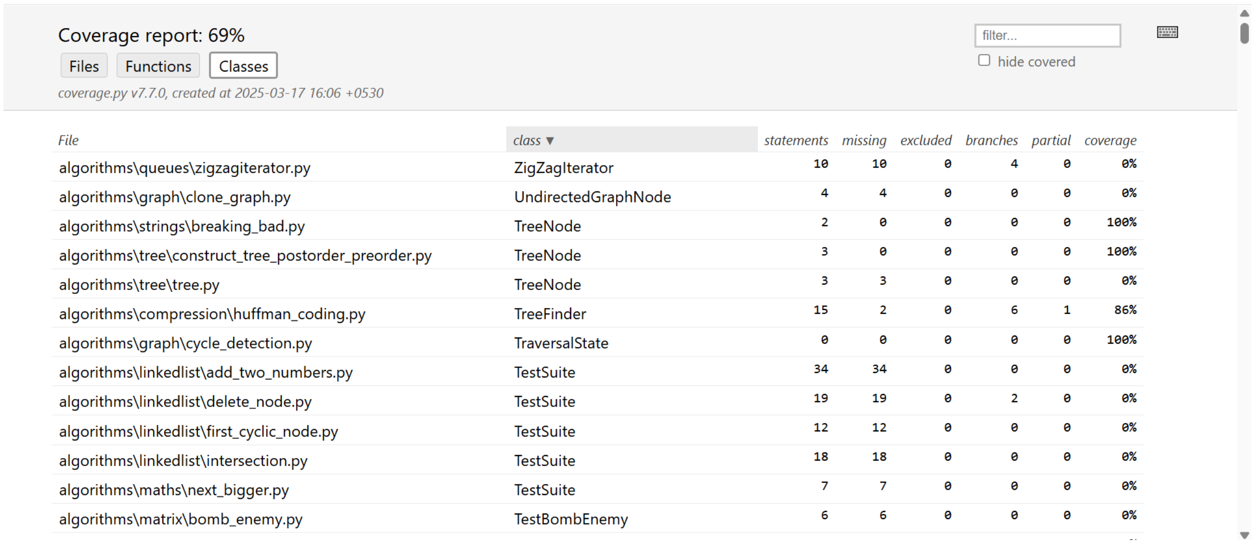
The interface also allows us to view the function and class-wise coverage of all the functions and classes present in the dataset of programs:



Coverage report: 69%

coverage.py v7.7.0, created at 2025-03-17 16:06 +0530

File	function	statements	missing	excluded	branches	partial	coverage ▼
algorithms__init__.py	*(no function)*	0	0	0	0	0	100%
algorithms\arrays__init__.py	*(no function)*	19	0	0	0	0	100%
algorithms\arrays\delete_nth.py	delete_nth_naive	5	0	0	4	0	100%
algorithms\arrays\delete_nth.py	delete_nth	7	0	0	4	0	100%
algorithms\arrays\delete_nth.py	*(no function)*	3	0	0	0	0	100%
algorithms\arrays\flatten.py	flatten	7	0	0	6	0	100%
algorithms\arrays\flatten.py	flatten_iter	4	0	0	4	0	100%
algorithms\arrays\flatten.py	*(no function)*	3	0	0	0	0	100%
algorithms\arrays\garage.py	*(no function)*	2	0	0	0	0	100%
algorithms\arrays\josephus.py	josephus	7	0	0	2	0	100%
algorithms\arrays\josephus.py	*(no function)*	1	0	0	0	0	100%
algorithms\arrays\limit.py	*(no function)*	1	0	0	0	0	100%
algorithms\arrays\longest_non_repeat.py	*(no function)*	5	0	0	0	0	100%
algorithms\arrays\max_ones_index.py	max_ones_index	15	0	0	8	0	100%
algorithms\arrays\max_ones_index.py	*(no function)*	1	0	0	0	0	100%
algorithms\arrays\merge_intervals.py	Interval.__init__	2	0	0	0	0	100%
algorithms\arrays\merge_intervals.py	Interval.merge	6	0	0	4	0	100%
algorithms\arrays\merge_intervals.py	*(no function)*	14	0	0	0	0	100%

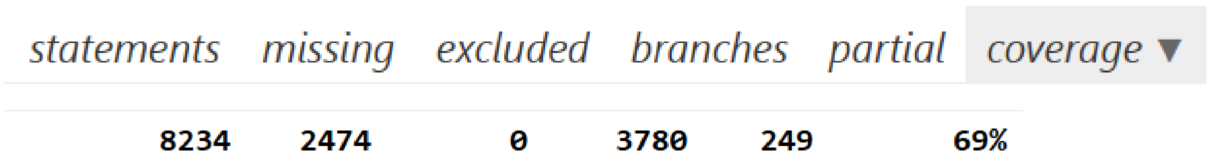


Coverage report: 69%

coverage.py v7.7.0, created at 2025-03-17 16:06 +0530

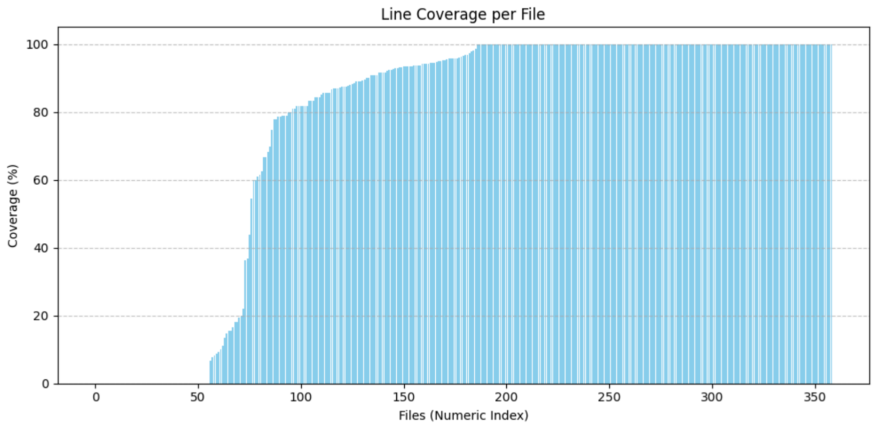
File	class ▼	statements	missing	excluded	branches	partial	coverage
algorithms\queues\zigzagiterator.py	ZigZagIterator	10	10	0	4	0	0%
algorithms\graph\clone_graph.py	UndirectedGraphNode	4	4	0	0	0	0%
algorithms\strings\breaking_bad.py	TreeNode	2	0	0	0	0	100%
algorithms\tree\construct_tree_postorder_preorder.py	TreeNode	3	0	0	0	0	100%
algorithms\tree\tree.py	TreeNode	3	3	0	0	0	0%
algorithms\compression\huffman_coding.py	TreeFinder	15	2	0	6	1	86%
algorithms\graph\cycle_detection.py	TraversalState	0	0	0	0	0	100%
algorithms\linkedlist\add_two_numbers.py	TestSuite	34	34	0	0	0	0%
algorithms\linkedlist\delete_node.py	TestSuite	19	19	0	2	0	0%
algorithms\linkedlist\first_cyclic_node.py	TestSuite	12	12	0	0	0	0%
algorithms\linkedlist\intersection.py	TestSuite	18	18	0	0	0	0%
algorithms\maths\next_bigger.py	TestSuite	7	7	0	0	0	0%
algorithms\matrix\bomb_enemy.py	TestBombEnemy	6	6	0	0	0	0%

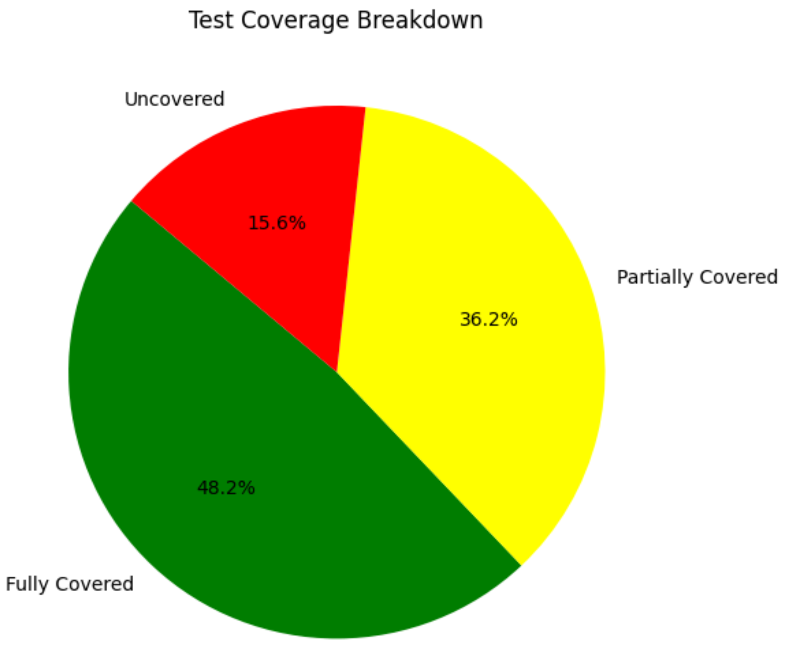
The result of coverage by the given test suite on the overall dataset is:

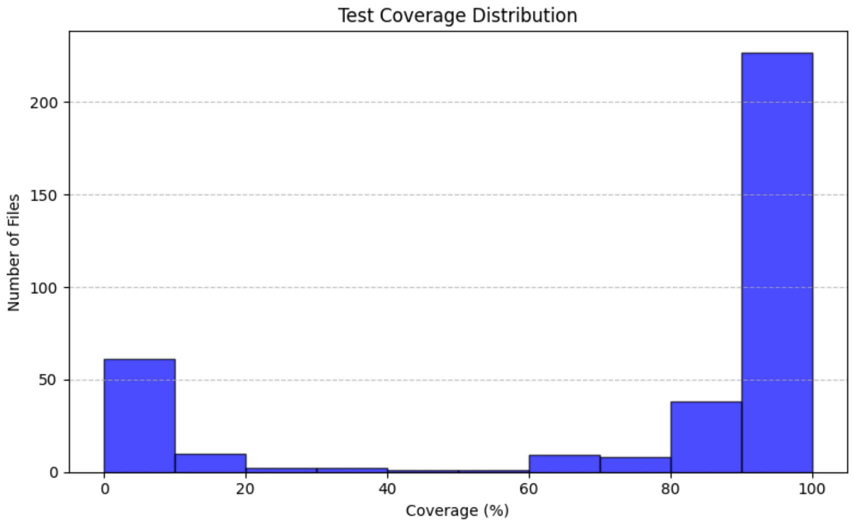


statements	missing	excluded	branches	partial	coverage ▼
8234	2474	0	3780	249	69%

I analyzed the coverage results by reading the generated XML file of the coverage report using a Python script and generating graphs to show the overall coverage % by the test suite, the line coverage per file and the test coverage distribution. The following visualizations were obtained:


Line Coverage per File
Coverage (%)
Files (Numeric Index)
0
20
40
60
80
100
0
50
100
150
200
250
300
350

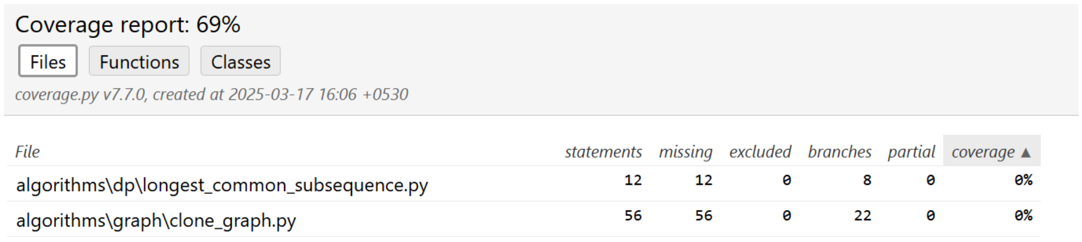

Test Coverage Breakdown
Uncovered
15.6%
Partially Covered
36.2%
Fully Covered
48.2%


Test Coverage Distribution
Number of Files
Coverage (%)
0
50
100
150
200
0
20
40
60
80
100

3.2 Coverage report for the generated test cases:

The individual coverage report of the 3 files for which I generated the test cases using pynguin are presented here. The generated tests were run individually first on all the files, and then combined with the original test suite. The generated tests for the specific files also ended up covering some other files when run on all the files, as can be seen below:
IMPORTANT NOTE: In all of the following cases, the first image shows the coverage of the file in question by running ALL the tests present in the original test suite, and the second image shows the coverage of the same file by ONLY THE GENERATED test case for that file. So, the overall coverage report percentage in the following snapshots is not to be considered, and only the particular file's coverage is to be considered.

1. clone_graph.py (graph module)
   Original test coverage: 0%

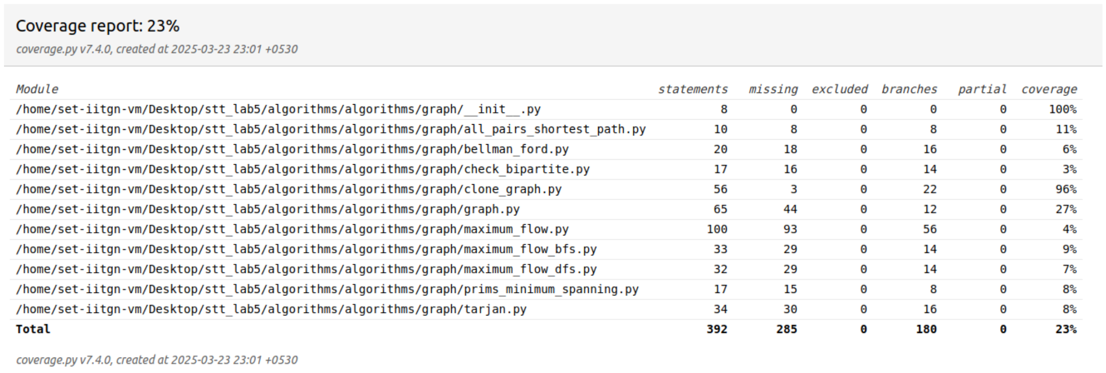


Coverage report: 69%

coverage.py v7.7.0, created at 2025-03-17 16:06 +0530

File	statements	missing	excluded	branches	partial	coverage ▲
algorithms\dp\longest_common_subsequence.py	12	12	0	8	0	0%
algorithms\graph\clone_graph.py	56	56	0	22	0	0%

Generated test coverage: 96%



Coverage report: 23%

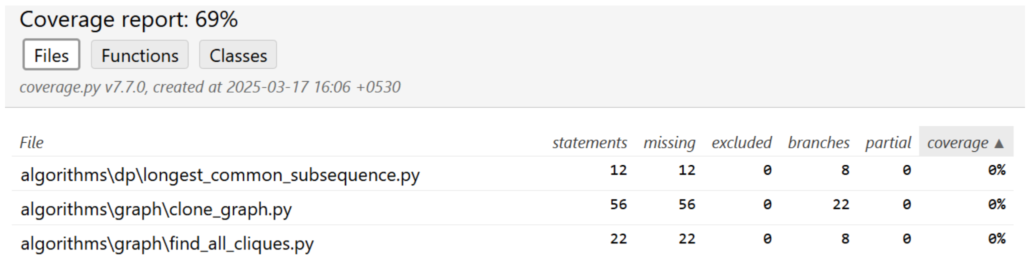
coverage.py v7.4.0, created at 2025-03-23 23:01 +0530

Module	statements	missing	excluded	branches	partial	coverage
/home/set-iitgn-vm/Desktop/stt_lab5/algorithms/algorithms/graph/__init__.py	8	0	0	0	0	100%
/home/set-iitgn-vm/Desktop/stt_lab5/algorithms/algorithms/graph/all_pairs_shortest_path.py	10	8	0	8	0	11%
/home/set-iitgn-vm/Desktop/stt_lab5/algorithms/algorithms/graph/bellman_ford.py	20	18	0	16	0	6%
/home/set-iitgn-vm/Desktop/stt_lab5/algorithms/algorithms/graph/check_bipartite.py	17	16	0	14	0	3%
/home/set-iitgn-vm/Desktop/stt_lab5/algorithms/algorithms/graph/clone_graph.py	56	3	0	22	0	96%
/home/set-iitgn-vm/Desktop/stt_lab5/algorithms/algorithms/graph/graph.py	65	44	0	12	0	27%
/home/set-iitgn-vm/Desktop/stt_lab5/algorithms/algorithms/graph/maximum_flow.py	100	93	0	56	0	4%
/home/set-iitgn-vm/Desktop/stt_lab5/algorithms/algorithms/graph/maximum_flow_bfs.py	33	29	0	14	0	9%
/home/set-iitgn-vm/Desktop/stt_lab5/algorithms/algorithms/graph/maximum_flow_dfs.py	32	29	0	14	0	7%
/home/set-iitgn-vm/Desktop/stt_lab5/algorithms/algorithms/graph/prims_minimum_spanning.py	17	15	0	8	0	8%
/home/set-iitgn-vm/Desktop/stt_lab5/algorithms/algorithms/graph/tarjan.py	34	30	0	16	0	8%
Total	**392**	**285**	**0**	**180**	**0**	**23%**

coverage.py v7.4.0, created at 2025-03-23 23:01 +0530

2. find_all_cliques.py (graph module)

Original test coverage: 0%



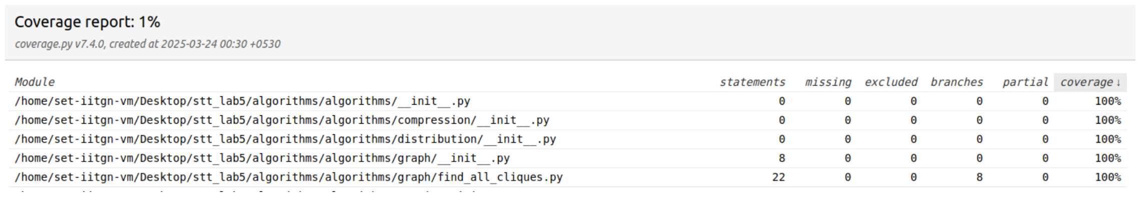
Coverage report: 69%

Files | Functions | Classes

coverage.py v7.7.0, created at 2025-03-17 16:06 +0530

File	statements	missing	excluded	branches	partial	coverage ▲
algorithms\dp\longest_common_subsequence.py	12	12	0	8	0	0%
algorithms\graph\clone_graph.py	56	56	0	22	0	0%
algorithms\graph\find_all_cliques.py	22	22	0	8	0	0%

Generated test coverage: 100%



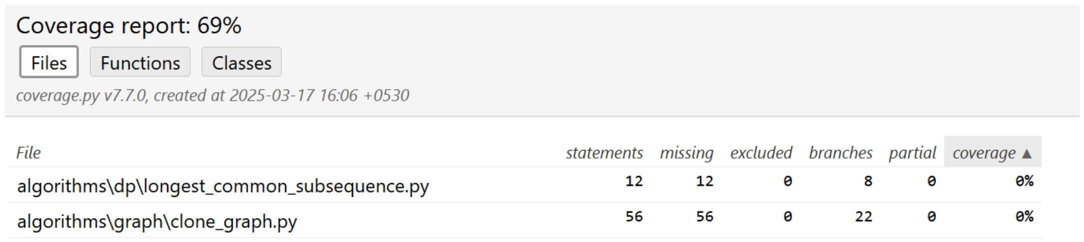
Coverage report: 1%

coverage.py v7.4.0, created at 2025-03-24 00:30 +0530

Module	statements	missing	excluded	branches	partial	coverage ↓
/home/set-iitgn-vm/Desktop/stt_lab5/algorithms/algorithms/__init__.py	0	0	0	0	0	100%
/home/set-iitgn-vm/Desktop/stt_lab5/algorithms/algorithms/compression/__init__.py	0	0	0	0	0	100%
/home/set-iitgn-vm/Desktop/stt_lab5/algorithms/algorithms/distribution/__init__.py	0	0	0	0	0	100%
/home/set-iitgn-vm/Desktop/stt_lab5/algorithms/algorithms/graph/__init__.py	8	0	0	0	0	100%
/home/set-iitgn-vm/Desktop/stt_lab5/algorithms/algorithms/graph/find_all_cliques.py	22	0	0	8	0	100%

3. longest_common_subsequence.py (dynamic programming module)

Original test coverage: 0%



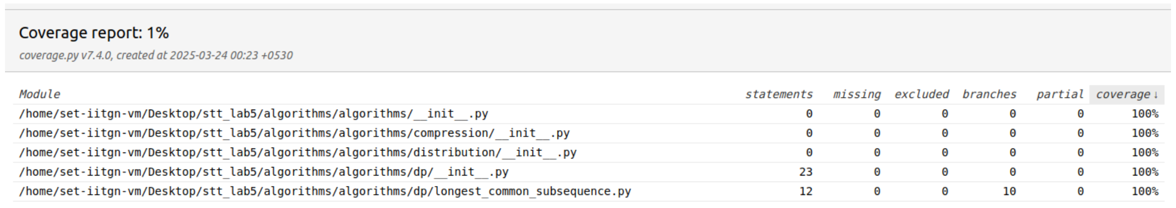
Coverage report: 69%

Files | Functions | Classes

coverage.py v7.7.0, created at 2025-03-17 16:06 +0530

File	statements	missing	excluded	branches	partial	coverage ▲
algorithms\dp\longest_common_subsequence.py	12	12	0	8	0	0%
algorithms\graph\clone_graph.py	56	56	0	22	0	0%

Generated test coverage: 100%



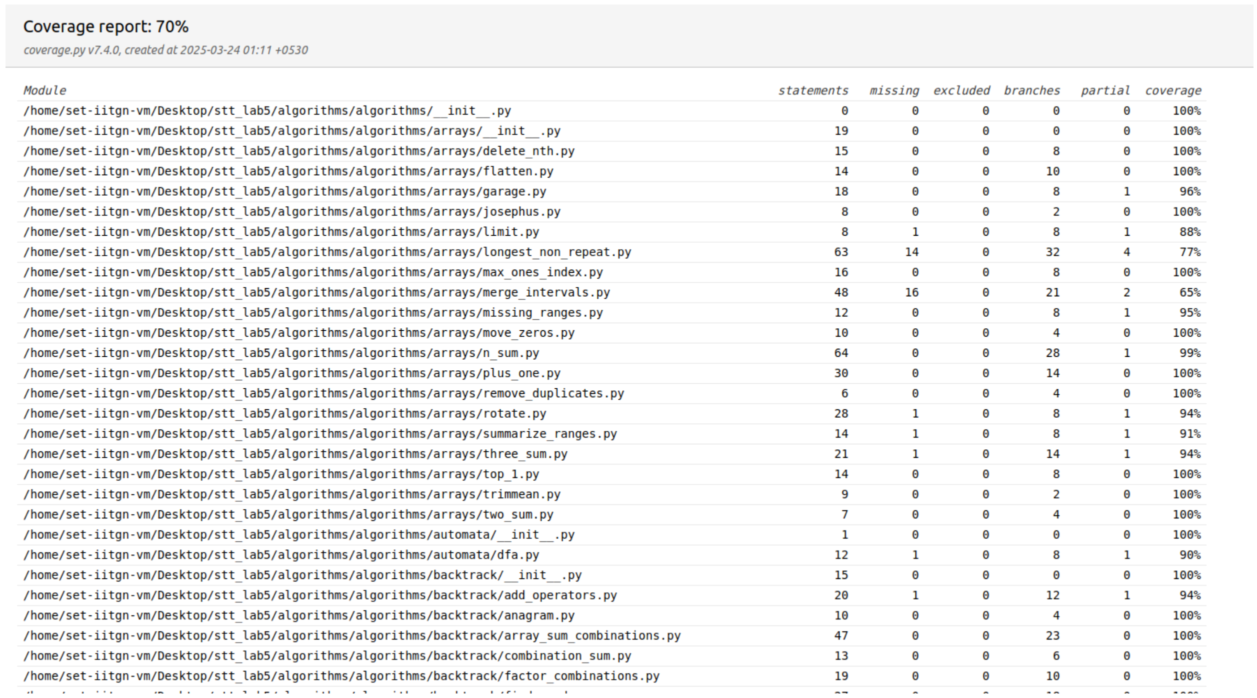
Coverage report: 1%

coverage.py v7.4.0, created at 2025-03-24 00:23 +0530

Module	statements	missing	excluded	branches	partial	coverage ↓
/home/set-iitgn-vm/Desktop/stt_lab5/algorithms/algorithms/__init__.py	0	0	0	0	0	100%
/home/set-iitgn-vm/Desktop/stt_lab5/algorithms/algorithms/compression/__init__.py	0	0	0	0	0	100%
/home/set-iitgn-vm/Desktop/stt_lab5/algorithms/algorithms/distribution/__init__.py	0	0	0	0	0	100%
/home/set-iitgn-vm/Desktop/stt_lab5/algorithms/algorithms/dp/__init__.py	23	0	0	0	0	100%
/home/set-iitgn-vm/Desktop/stt_lab5/algorithms/algorithms/dp/longest_common_subsequence.py	12	0	0	10	0	100%

3.3 Coverage analysis of the combined test suite (original + generated tests)

The coverage of the combined test suite (where I added the generated tests to the given tests folder and ran the folder on all the files again) is:

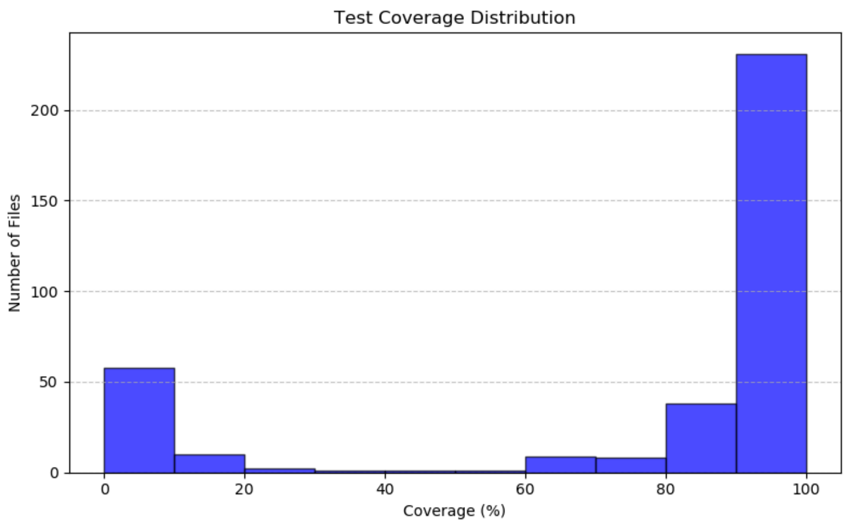


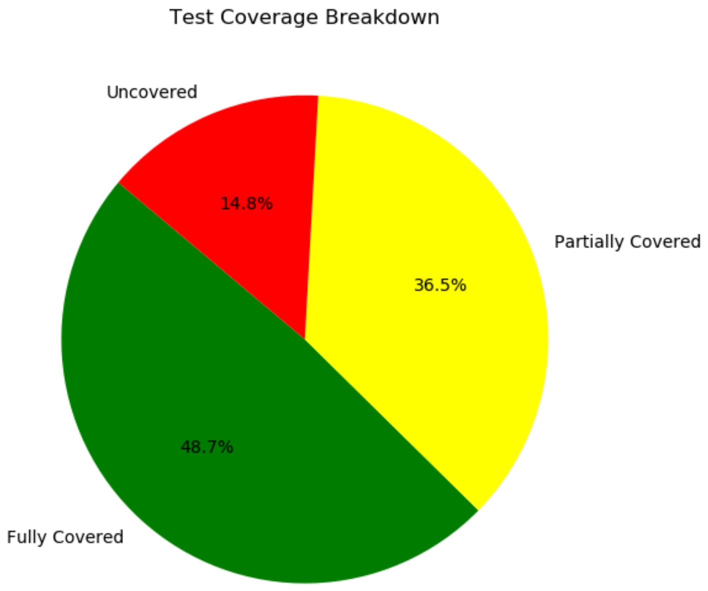
Coverage report: 70%

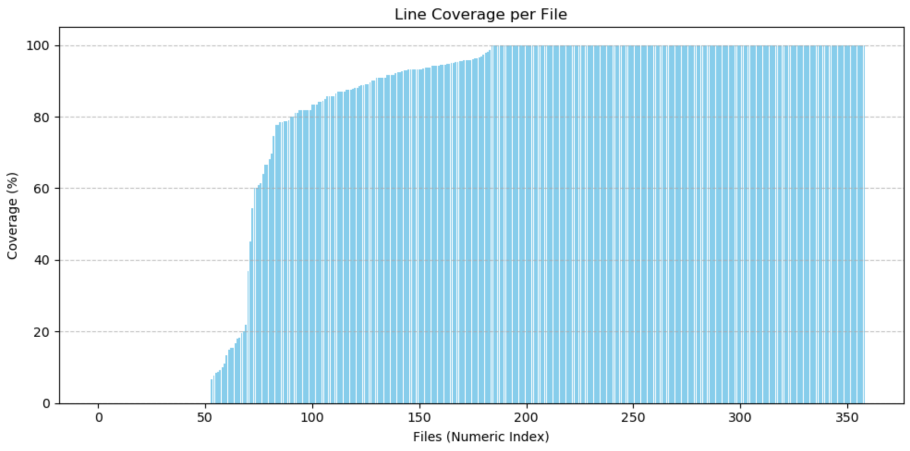
coverage.py v7.4.0, created at 2025-03-24 01:11 +0530

Module	*statements*	*missing*	*excluded*	*branches*	*partial*	*coverage*
/home/set-iitgn-vm/Desktop/stt_lab5/algorithms/algorithms/__init__.py	0	0	0	0	0	100%
/home/set-iitgn-vm/Desktop/stt_lab5/algorithms/algorithms/arrays/__init__.py	19	0	0	0	0	100%
/home/set-iitgn-vm/Desktop/stt_lab5/algorithms/algorithms/arrays/delete_nth.py	15	0	0	8	0	100%
/home/set-iitgn-vm/Desktop/stt_lab5/algorithms/algorithms/arrays/flatten.py	14	0	0	10	0	100%
/home/set-iitgn-vm/Desktop/stt_lab5/algorithms/algorithms/arrays/garage.py	18	0	0	8	1	96%
/home/set-iitgn-vm/Desktop/stt_lab5/algorithms/algorithms/arrays/josephus.py	8	0	0	2	0	100%
/home/set-iitgn-vm/Desktop/stt_lab5/algorithms/algorithms/arrays/limit.py	8	1	0	8	1	88%
/home/set-iitgn-vm/Desktop/stt_lab5/algorithms/algorithms/arrays/longest_non_repeat.py	63	14	0	32	4	77%
/home/set-iitgn-vm/Desktop/stt_lab5/algorithms/algorithms/arrays/max_ones_index.py	16	0	0	8	0	100%
/home/set-iitgn-vm/Desktop/stt_lab5/algorithms/algorithms/arrays/merge_intervals.py	48	16	0	21	2	65%
/home/set-iitgn-vm/Desktop/stt_lab5/algorithms/algorithms/arrays/missing_ranges.py	12	0	0	8	1	95%
/home/set-iitgn-vm/Desktop/stt_lab5/algorithms/algorithms/arrays/move_zeros.py	10	0	0	4	0	100%
/home/set-iitgn-vm/Desktop/stt_lab5/algorithms/algorithms/arrays/n_sum.py	64	0	0	28	1	99%
/home/set-iitgn-vm/Desktop/stt_lab5/algorithms/algorithms/arrays/plus_one.py	30	0	0	14	0	100%
/home/set-iitgn-vm/Desktop/stt_lab5/algorithms/algorithms/arrays/remove_duplicates.py	6	0	0	4	0	100%
/home/set-iitgn-vm/Desktop/stt_lab5/algorithms/algorithms/arrays/rotate.py	28	1	0	8	1	94%
/home/set-iitgn-vm/Desktop/stt_lab5/algorithms/algorithms/arrays/summarize_ranges.py	14	1	0	8	1	91%
/home/set-iitgn-vm/Desktop/stt_lab5/algorithms/algorithms/arrays/three_sum.py	21	1	0	14	1	94%
/home/set-iitgn-vm/Desktop/stt_lab5/algorithms/algorithms/arrays/top_1.py	14	0	0	8	0	100%
/home/set-iitgn-vm/Desktop/stt_lab5/algorithms/algorithms/arrays/trimmean.py	9	0	0	2	0	100%
/home/set-iitgn-vm/Desktop/stt_lab5/algorithms/algorithms/arrays/two_sum.py	7	0	0	4	0	100%
/home/set-iitgn-vm/Desktop/stt_lab5/algorithms/algorithms/automata/__init__.py	1	0	0	0	0	100%
/home/set-iitgn-vm/Desktop/stt_lab5/algorithms/algorithms/automata/dfa.py	12	1	0	8	1	90%
/home/set-iitgn-vm/Desktop/stt_lab5/algorithms/algorithms/backtrack/__init__.py	15	0	0	0	0	100%
/home/set-iitgn-vm/Desktop/stt_lab5/algorithms/algorithms/backtrack/add_operators.py	20	1	0	12	1	94%
/home/set-iitgn-vm/Desktop/stt_lab5/algorithms/algorithms/backtrack/anagram.py	10	0	0	4	0	100%
/home/set-iitgn-vm/Desktop/stt_lab5/algorithms/algorithms/backtrack/array_sum_combinations.py	47	0	0	23	0	100%
/home/set-iitgn-vm/Desktop/stt_lab5/algorithms/algorithms/backtrack/combination_sum.py	13	0	0	6	0	100%
/home/set-iitgn-vm/Desktop/stt_lab5/algorithms/algorithms/backtrack/factor_combinations.py	19	0	0	10	0	100%

The following visualisations help in representing the coverage results better, again generated by reading the XML coverage report file using a Python script:

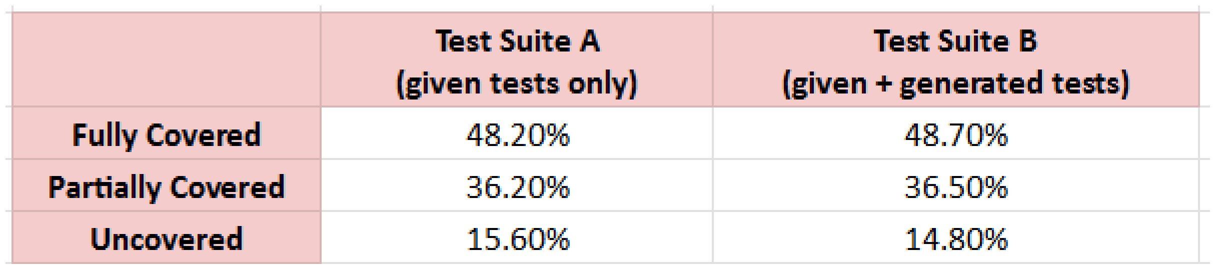

Test Coverage Distribution
Number of Files
Coverage (%)
0
20
40
60
80
100
0
50
100
150
200


Test Coverage Breakdown
Uncovered
14.8%
Partially Covered
36.5%
Fully Covered
48.7%


Line Coverage per File
Coverage (%)
Files (Numeric Index)
0
20
40
60
80
100
0
50
100
150
200
250
300
350

3.4 Comparing the original and combined (original + combined tests cases)

As we saw in the above two sub-sections, **the overall code coverage increased by 1% on adding three new test cases.** The following table logs the coverage percentage for both the test suites:



	Test Suite A (given tests only)	Test Suite B (given + generated tests)
Fully Covered	48.20%	48.70%
Partially Covered	36.20%	36.50%
Uncovered	15.60%	14.80%

4. Challenges Faced

- Initial error in cloning the repository: There was some issue while cloning the repository (common for all students) which led to the main 'algorithms' folder within the root directory to not be loaded successfully. This was fixed by running `pip install -e . in` in the root directory after cloning the repository.
- Syntax error in the file test_array.py: A missing comma in this file led to the failed execution of a lot of tests. This was fixed by adding a comma manually where it was missing.
- Python version incompatibility didn't allow me to run pynguin on my local machine, so I had to use SET-IITGN-VM to run that part.
- Test generation for certain uncovered files was taking too long, with certain processes failing eventually after running for several minutes (~20), so I had to halt the process midway for certain files and start all over again and experiment with different files.

5. Conclusion

The analysis of code coverage by the existing test case execution on all the files in the algorithms sub-directory within the root folder provided a baseline for assessing the effects of further test generation by presenting sections of the code that were entirely covered, half-covered, and uncovered. Although the provided test suite had already attained a full coverage of 48.20%, the existence of partially covered (36.20%) and uncovered (15.60%) code parts suggested that there was still opportunity for improvement.
In the second part, on trying Pynguin-based automated test case generation as a way to improve coverage, it was observed that because it took a lot of time, only a few test cases could be generated. A slight improvement in coverage was noted—fully covered code increased to 48.70% and uncovered code dropped to 14.80%—despite the small number of extra tests. Overall, the code coverage increased on adding the three generated test cases to the original test suite, and it could be further increased if the bottleneck of the time it takes for test case generation could be improved.

6. References

- Lecture 5 slides
- pynguin (https://www.pynguin.eu)
- coverage (https://coverage.readthedocs.io/en/latest)
- pytest (https://docs.pytest.org/en/7.4.x/index.html)
- pytest-cov (https://github.com/pytest-dev/pytest-cov)
- pytest-func-cov (https://pypi.org/project/pytest-func-cov)

CS202: Software Tools and Techniques

Lab 6 Report: Python Test Parallelization

Jiya Bhavin Desai | Roll No. 22110107 | 13.02.2025

1. Introduction

Parallel test execution is an essential technique for optimizing software testing workflows, enabling faster test suite execution by distributing tests across multiple processes or threads. In this lab, the focus was on understanding the challenges and effectiveness of test parallelization using Python's **pytest-xdist** (process-based parallelization) and **pytest-run-parallel** (thread-based parallelization). The following objectives were targeted to be achieved:

- Evaluating different parallelization strategies.
- Identifying flaky tests that behave inconsistently under parallel execution.
- Analyzing the speedup gains (comparing both sequential and parallel execution as well as different modes of parallel execution) and limitations of test parallelization.
- By performing parallelization on an open-source Python repository (keon/algorithms), to identify its readiness for parallel testing.

This report presents the observations and the subsequent analysis performed on them.

1.1 Setup

- Creating a virtual environment with Python (no version compatibility was specified - implying no conflicts with the latest version. I used Python 3.12.6)
- Installing the required dependencies: **pytest** (test execution), **pytest-xdist** (process level test parallelization) and **pytest-run-parallel** (thread level test parallelization)

1.2 Tools

- Windows Command Prompt (available on my local machine)
- Algorithms repository on GitHub (contains python files to be tested as well as those used for testing)
- pytest (for running tests)
- pytest-xdist (process level test parallelization)
- pytest-run-parallel (thread level test parallelization)

2. Methodology and Execution

Cloning the algorithms repository:

- I first created a virtual Python environment on my local device and activated it using the following commands:
  ```
  python -m venv venvLab6
  venvLab6\Scripts\activate
  ```
- Next, I cloned the repository using the command:
  ```
  git clone https://github.com/keon/algorithms
  ```

However, even after cloning the repository using the above command, there was some issue in loading the 'algorithms' folder (containing the programs to be tested) present in the root directory. To resolve this issue, the following command was run in the root directory of the cloned repo:

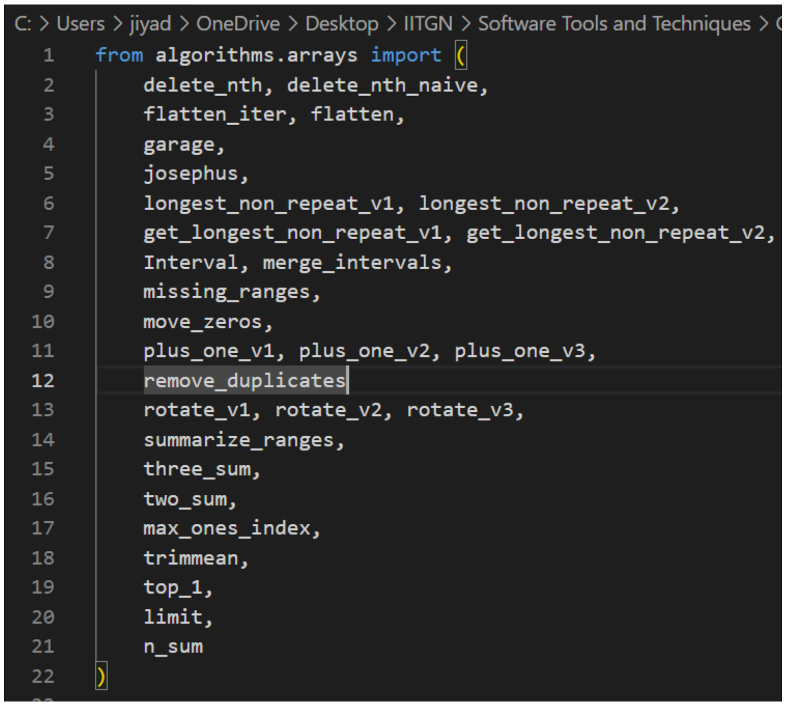
```
pip install -e .
```

About the repository:

The GitHub repository keon/algorithms provides a comprehensive collection of Python implementations for various data structures and algorithms. It includes a wide range of examples, from basic data structures like arrays, linked lists, and trees, to complex algorithms such as dynamic programming, graph traversal, and sorting techniques. The repository is a standard resource for learning and understanding the fundamentals of algorithms in Python. It also supports contributions and includes an almost-exhaustive testing framework using both unittest and pytest for ensuring the reliability of the code. While primarily focused on providing implementations of various algorithms, this dataset of programs can also serve as a valuable resource for analyzing code coverage, as explored in this lab assignment.

2.1 Running the given test suite for initial debugging and making minor syntax corrections:

- ***Initially, on running the given test suite as it is to assess the general accuracy and coverage , a few test cases were failing, and the initial test case coverage was unusually low - just 14%.*** When I checked the files that were throwing the errors, I found that there was one syntax error (a missing comma) in the file *test_array.py*. On fixing this one error, the coverage increased significantly - this was observed in lab 5 as well. **Even though this lab involves the identification and elimination of failing and flaky test cases, I fixed this one file before proceeding - this was discussed during the lab session with the TAs as well.**



```
C: > Users > jiyad > OneDrive > Desktop > IITGN > Software Tools and Techniques >
 1  from algorithms.arrays import (
 2      delete_nth, delete_nth_naive,
 3      flatten_iter, flatten,
 4      garage,
 5      josephus,
 6      longest_non_repeat_v1, longest_non_repeat_v2,
 7      get_longest_non_repeat_v1, get_longest_non_repeat_v2,
 8      Interval, merge_intervals,
 9      missing_ranges,
10      move_zeros,
11      plus_one_v1, plus_one_v2, plus_one_v3,
12      remove_duplicates
13      rotate_v1, rotate_v2, rotate_v3,
14      summarize_ranges,
15      three_sum,
16      two_sum,
17      max_ones_index,
18      trimmean,
19      top_1,
20      limit,
21      n_sum
22  )
```

Line 12 had a missing comma, causing the entire test_array.py to not run

2.2 Sequential Execution:

The first part involved executing the given tests sequentially, as is the default setting. I ran all the tests sequentially 10 times using the following command (***the outputs and analysis are presented in section 3***):

```
for ($i=1; $i -le 10; $i++) {pytest | Tee-Object -FilePath
"sequential_run_$i.log"}
```

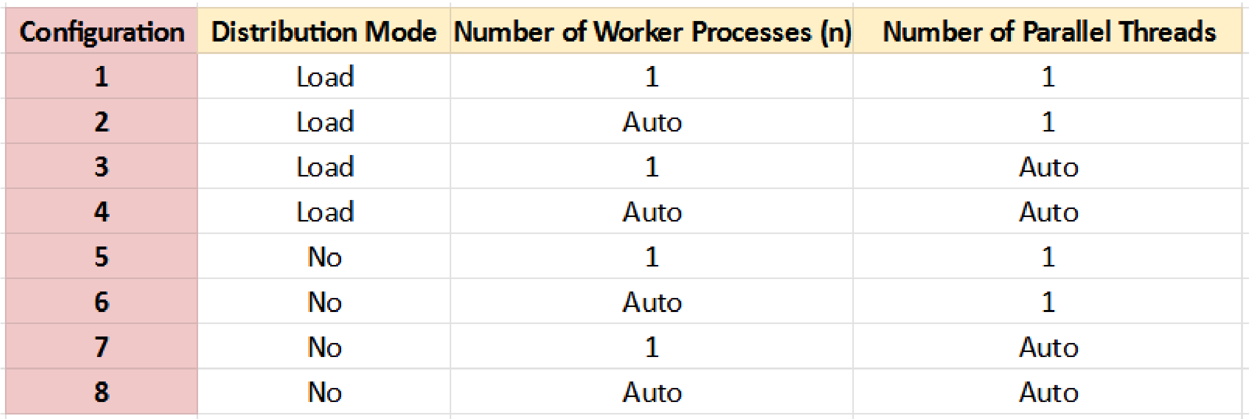
Next, I removed the failing test cases identified in the outputs of the above command by commenting out the related code section, and again ran the error-free test suite thrice using the same command (changing 10 to 3).

2.3 Parallel Execution:

Parallelization of the test suite included trying out various configurations of parallel execution. In the pytest command used for test execution, there are three parameters that can be varied:

1. **The distribution mode:** Determines how test cases are distributed among worker processes. When set to load, it dynamically distributes tests to workers based on their availability, helping balance execution time. When set to no, it statically assigns tests to workers without redistributing, meaning each worker gets a fixed set of tests.
2. **The number of worker processes:** Specifies how many worker processes should be created for parallel test execution. When set to 1, it runs all tests sequentially in a single worker process. When set to auto, it automatically assigns the number of workers based on the available CPU cores.
3. **The number of parallel threads:** Defines the number of threads each worker process can use for test execution. When set to 1, each worker runs tests in a single thread. When set to auto, it determines the optimal number of threads based on system resources.

All possible combinations of the above three parameters were tested, resulting into the following eight configurations:



Configuration	Distribution Mode	Number of Worker Processes (n)	Number of Parallel Threads
1	Load	1	1
2	Load	Auto	1
3	Load	1	Auto
4	Load	Auto	Auto
5	No	1	1
6	No	Auto	1
7	No	1	Auto
8	No	Auto	Auto

An example command for running the tests in parallel is:

```
pytest -n auto --dist load --parallel-threads auto
```

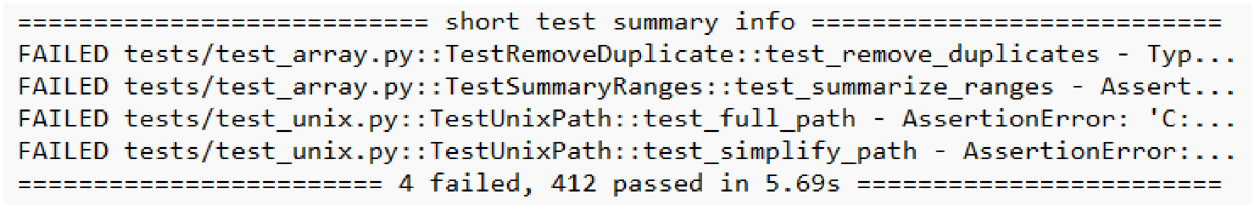
(for configuration number 4)

The detailed outputs of all configurations are mentioned in the next section.

3. Results and Analysis

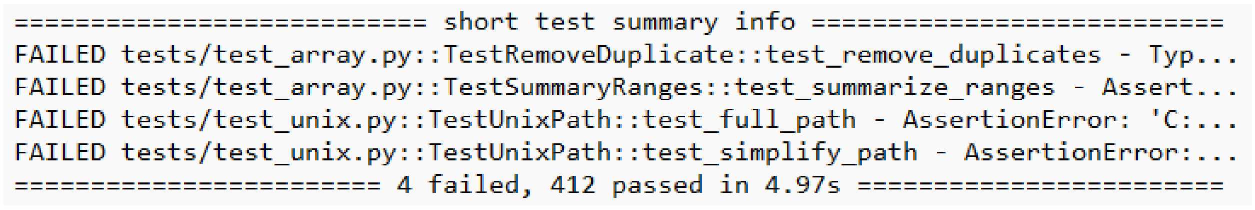
3.1 Results of Sequential Execution:

After running the test suite sequentially 10 times, I logged the output of each run in separate log files (the command used has been written in the previous section) and inspected it to identify failing and flaky test cases. All 10 outputs are shown below:



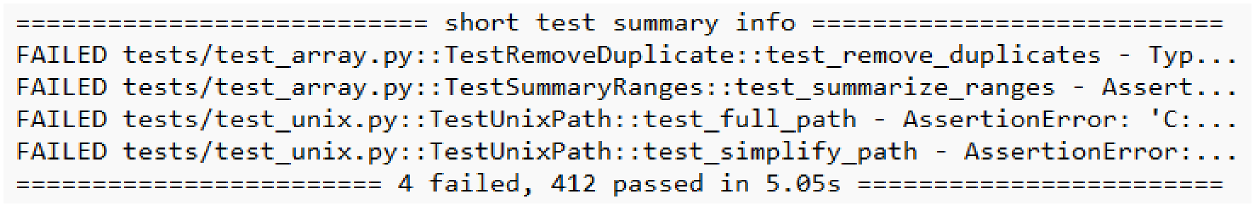
```
=========================== short test summary info ============================
FAILED tests/test_array.py::TestRemoveDuplicate::test_remove_duplicates - Typ...
FAILED tests/test_array.py::TestSummaryRanges::test_summarize_ranges - Assert...
FAILED tests/test_unix.py::TestUnixPath::test_full_path - AssertionError: 'C:...
FAILED tests/test_unix.py::TestUnixPath::test_simplify_path - AssertionError:...
======================== 4 failed, 412 passed in 5.69s ========================
```

Run 1



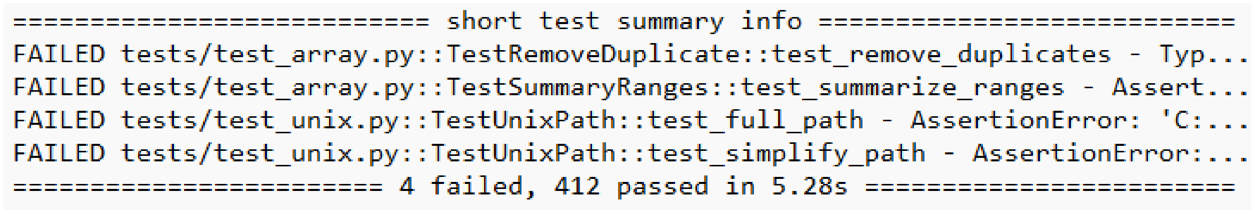
```
=========================== short test summary info ============================
FAILED tests/test_array.py::TestRemoveDuplicate::test_remove_duplicates - Typ...
FAILED tests/test_array.py::TestSummaryRanges::test_summarize_ranges - Assert...
FAILED tests/test_unix.py::TestUnixPath::test_full_path - AssertionError: 'C:...
FAILED tests/test_unix.py::TestUnixPath::test_simplify_path - AssertionError:...
======================== 4 failed, 412 passed in 4.97s ========================
```

Run 2



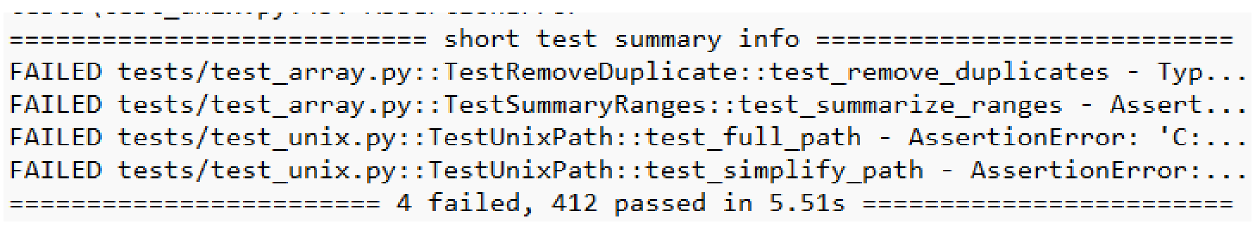
```
=========================== short test summary info ===========================
FAILED tests/test_array.py::TestRemoveDuplicate::test_remove_duplicates - Typ...
FAILED tests/test_array.py::TestSummaryRanges::test_summarize_ranges - Assert...
FAILED tests/test_unix.py::TestUnixPath::test_full_path - AssertionError: 'C:...
FAILED tests/test_unix.py::TestUnixPath::test_simplify_path - AssertionError:...
======================== 4 failed, 412 passed in 5.05s ========================
```

Run 3



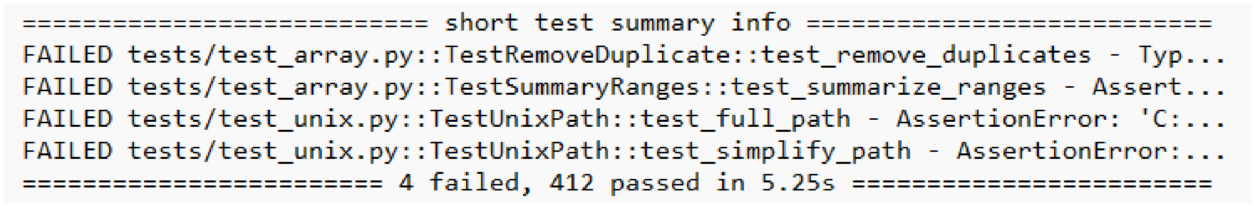
```
=========================== short test summary info ===========================
FAILED tests/test_array.py::TestRemoveDuplicate::test_remove_duplicates - Typ...
FAILED tests/test_array.py::TestSummaryRanges::test_summarize_ranges - Assert...
FAILED tests/test_unix.py::TestUnixPath::test_full_path - AssertionError: 'C:...
FAILED tests/test_unix.py::TestUnixPath::test_simplify_path - AssertionError:...
======================== 4 failed, 412 passed in 5.28s ========================
```

Run 4



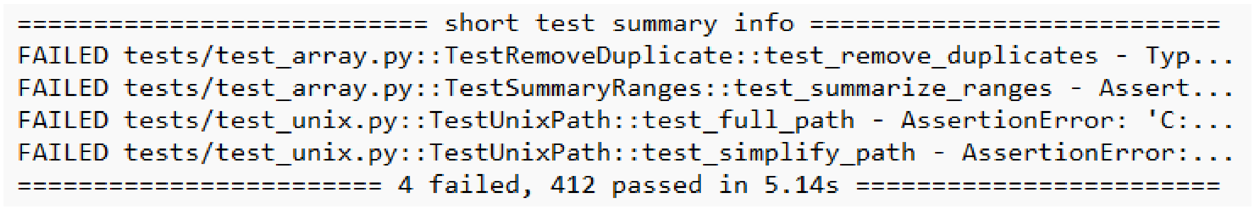
```
=========================== short test summary info ===========================
FAILED tests/test_array.py::TestRemoveDuplicate::test_remove_duplicates - Typ...
FAILED tests/test_array.py::TestSummaryRanges::test_summarize_ranges - Assert...
FAILED tests/test_unix.py::TestUnixPath::test_full_path - AssertionError: 'C:...
FAILED tests/test_unix.py::TestUnixPath::test_simplify_path - AssertionError:...
======================== 4 failed, 412 passed in 5.51s ========================
```

Run 5



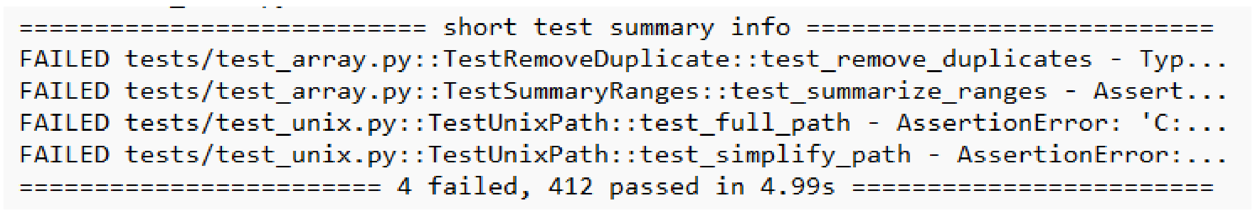
```
=========================== short test summary info ===========================
FAILED tests/test_array.py::TestRemoveDuplicate::test_remove_duplicates - Typ...
FAILED tests/test_array.py::TestSummaryRanges::test_summarize_ranges - Assert...
FAILED tests/test_unix.py::TestUnixPath::test_full_path - AssertionError: 'C:...
FAILED tests/test_unix.py::TestUnixPath::test_simplify_path - AssertionError:...
======================== 4 failed, 412 passed in 5.25s ========================
```

Run 6



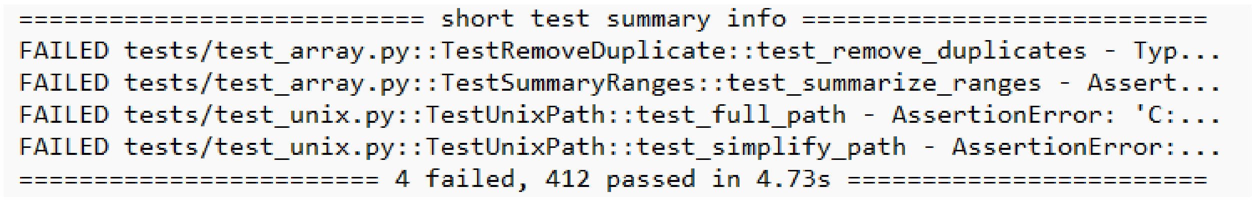
```
=========================== short test summary info ===========================
FAILED tests/test_array.py::TestRemoveDuplicate::test_remove_duplicates - Typ...
FAILED tests/test_array.py::TestSummaryRanges::test_summarize_ranges - Assert...
FAILED tests/test_unix.py::TestUnixPath::test_full_path - AssertionError: 'C:...
FAILED tests/test_unix.py::TestUnixPath::test_simplify_path - AssertionError:...
======================== 4 failed, 412 passed in 5.14s ========================
```

Run 7



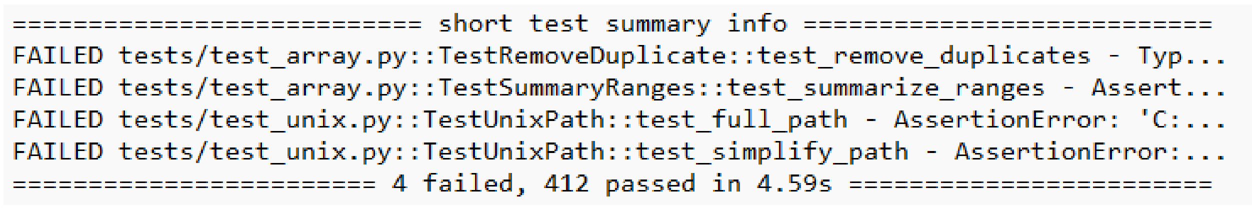
```
=========================== short test summary info ===========================
FAILED tests/test_array.py::TestRemoveDuplicate::test_remove_duplicates - Typ...
FAILED tests/test_array.py::TestSummaryRanges::test_summarize_ranges - Assert...
FAILED tests/test_unix.py::TestUnixPath::test_full_path - AssertionError: 'C:...
FAILED tests/test_unix.py::TestUnixPath::test_simplify_path - AssertionError:...
======================== 4 failed, 412 passed in 4.99s ========================
```

Run 8



```
=========================== short test summary info ===========================
FAILED tests/test_array.py::TestRemoveDuplicate::test_remove_duplicates - Typ...
FAILED tests/test_array.py::TestSummaryRanges::test_summarize_ranges - Assert...
FAILED tests/test_unix.py::TestUnixPath::test_full_path - AssertionError: 'C:...
FAILED tests/test_unix.py::TestUnixPath::test_simplify_path - AssertionError:...
======================== 4 failed, 412 passed in 4.73s ========================
```

Run 9



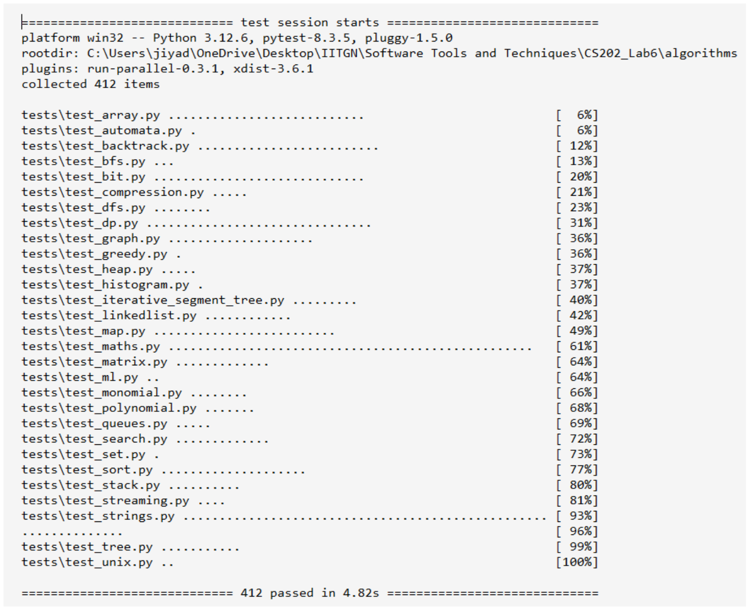
```
=========================== short test summary info ===========================
FAILED tests/test_array.py::TestRemoveDuplicate::test_remove_duplicates - Typ...
FAILED tests/test_array.py::TestSummaryRanges::test_summarize_ranges - Assert...
FAILED tests/test_unix.py::TestUnixPath::test_full_path - AssertionError: 'C:...
FAILED tests/test_unix.py::TestUnixPath::test_simplify_path - AssertionError:...
======================== 4 failed, 412 passed in 4.59s ========================
```

Run 10

As can be seen, only ***4 of the test cases are failing in each run - there are no non-deterministic failures in this case.*** Hence, I identified these 4 test cases as failing test cases and removed them (by commenting out the relevant portion in the test code) before proceeding.

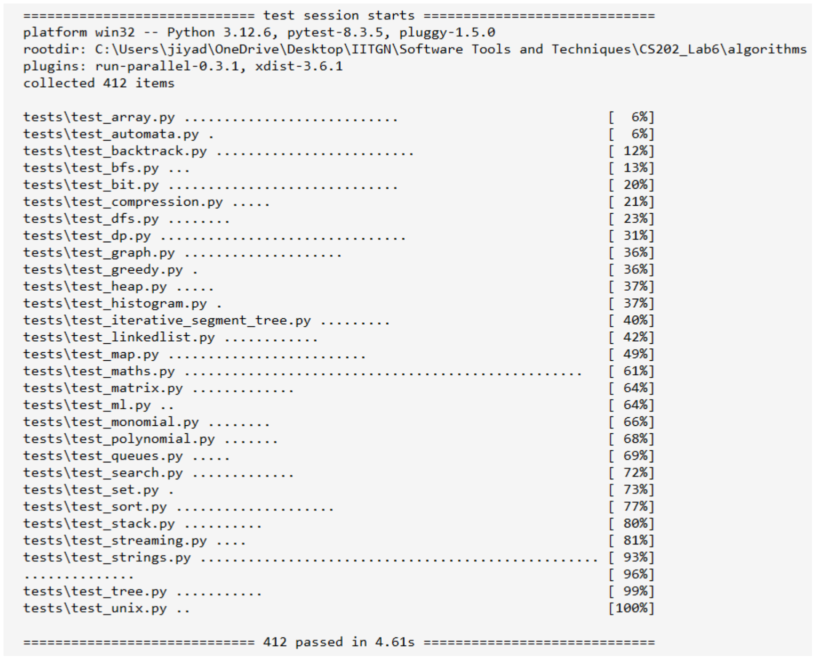
3.2 Average Sequential Execution Time After Removing Failing and Flaky Test Cases:

As expected, all test cases pass in all three runs after removing the failing test cases identified in the previous steps. I ran the cleaned test suite 3 times and the following output was obtained:



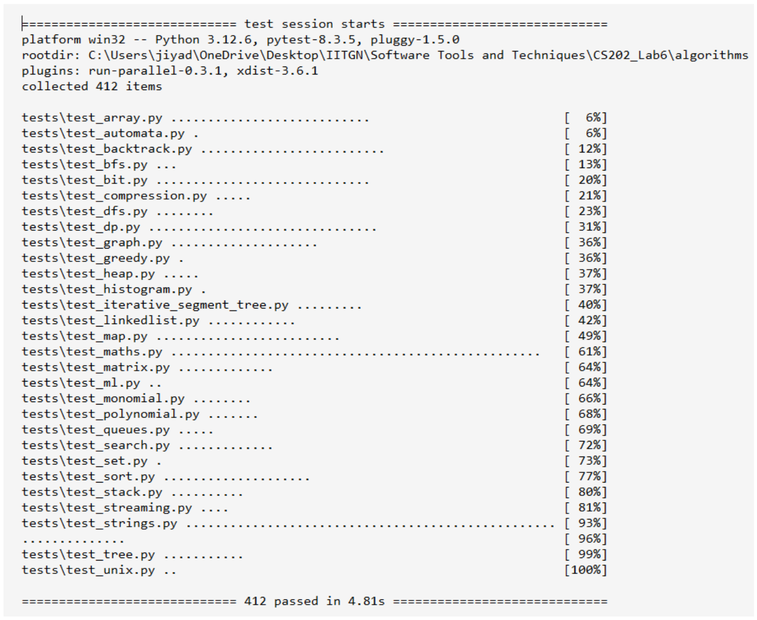
```
============================= test session starts =============================
platform win32 -- Python 3.12.6, pytest-8.3.5, pluggy-1.5.0
rootdir: C:\Users\jiyad\OneDrive\Desktop\IITGN\Software Tools and Techniques\CS202_Lab6\algorithms
plugins: run-parallel-0.3.1, xdist-3.6.1
collected 412 items

tests\test_array.py ...........................                          [  6%]
tests\test_automata.py .                                                 [  6%]
tests\test_backtrack.py .........................                        [ 12%]
tests\test_bfs.py ...                                                    [ 13%]
tests\test_bit.py .............................                          [ 20%]
tests\test_compression.py .....                                          [ 21%]
tests\test_dfs.py ........                                               [ 23%]
tests\test_dp.py ...............................                         [ 31%]
tests\test_graph.py ....................                                 [ 36%]
tests\test_greedy.py .                                                   [ 36%]
tests\test_heap.py .....                                                 [ 37%]
tests\test_histogram.py .                                                [ 37%]
tests\test_iterative_segment_tree.py .........                           [ 40%]
tests\test_linkedlist.py ............                                    [ 42%]
tests\test_map.py .........................                              [ 49%]
tests\test_maths.py ...................................................  [ 61%]
tests\test_matrix.py .............                                       [ 64%]
tests\test_ml.py ..                                                      [ 64%]
tests\test_monomial.py ........                                          [ 66%]
tests\test_polynomial.py .......                                         [ 68%]
tests\test_queues.py .....                                               [ 69%]
tests\test_search.py .............                                       [ 72%]
tests\test_set.py .                                                      [ 73%]
tests\test_sort.py ....................                                  [ 77%]
tests\test_stack.py ..........                                           [ 80%]
tests\test_streaming.py ....                                             [ 81%]
tests\test_strings.py ................................................. [ 93%]
..............                                                           [ 96%]
tests\test_tree.py ...........                                           [ 99%]
tests\test_unix.py ..                                                    [100%]

============================= 412 passed in 4.82s =============================
```

Run 1



```
============================= test session starts ==============================
platform win32 -- Python 3.12.6, pytest-8.3.5, pluggy-1.5.0
rootdir: C:\Users\jiyad\OneDrive\Desktop\IITGN\Software Tools and Techniques\CS202_Lab6\algorithms
plugins: run-parallel-0.3.1, xdist-3.6.1
collected 412 items

tests\test_array.py ...........................                           [  6%]
tests\test_automata.py .                                                  [  6%]
tests\test_backtrack.py .........................                         [ 12%]
tests\test_bfs.py ...                                                     [ 13%]
tests\test_bit.py .............................                           [ 20%]
tests\test_compression.py .....                                           [ 21%]
tests\test_dfs.py ........                                                [ 23%]
tests\test_dp.py ...............................                          [ 31%]
tests\test_graph.py ....................                                  [ 36%]
tests\test_greedy.py .                                                    [ 36%]
tests\test_heap.py .....                                                  [ 37%]
tests\test_histogram.py .                                                 [ 37%]
tests\test_iterative_segment_tree.py .........                            [ 40%]
tests\test_linkedlist.py ............                                     [ 42%]
tests\test_map.py .........................                               [ 49%]
tests\test_maths.py ..................................................    [ 61%]
tests\test_matrix.py .............                                        [ 64%]
tests\test_ml.py ..                                                       [ 64%]
tests\test_monomial.py ........                                           [ 66%]
tests\test_polynomial.py .......                                          [ 68%]
tests\test_queues.py .....                                                [ 69%]
tests\test_search.py .............                                        [ 72%]
tests\test_set.py .                                                       [ 73%]
tests\test_sort.py ....................                                   [ 77%]
tests\test_stack.py ..........                                            [ 80%]
tests\test_streaming.py ....                                              [ 81%]
tests\test_strings.py ................................................... [ 93%]
.............                                                             [ 96%]
tests\test_tree.py ...........                                            [ 99%]
tests\test_unix.py ..                                                     [100%]

============================= 412 passed in 4.61s ==============================
```

Run 2



```
============================= test session starts ==============================
platform win32 -- Python 3.12.6, pytest-8.3.5, pluggy-1.5.0
rootdir: C:\Users\jiyad\OneDrive\Desktop\IITGN\Software Tools and Techniques\CS202_Lab6\algorithms
plugins: run-parallel-0.3.1, xdist-3.6.1
collected 412 items

tests\test_array.py ...........................                           [  6%]
tests\test_automata.py .                                                  [  6%]
tests\test_backtrack.py .........................                         [ 12%]
tests\test_bfs.py ...                                                     [ 13%]
tests\test_bit.py .............................                           [ 20%]
tests\test_compression.py .....                                           [ 21%]
tests\test_dfs.py ........                                                [ 23%]
tests\test_dp.py ...............................                          [ 31%]
tests\test_graph.py ....................                                  [ 36%]
tests\test_greedy.py .                                                    [ 36%]
tests\test_heap.py .....                                                  [ 37%]
tests\test_histogram.py .                                                 [ 37%]
tests\test_iterative_segment_tree.py .........                            [ 40%]
tests\test_linkedlist.py ............                                     [ 42%]
tests\test_map.py .........................                               [ 49%]
tests\test_maths.py ..................................................    [ 61%]
tests\test_matrix.py .............                                        [ 64%]
tests\test_ml.py ..                                                       [ 64%]
tests\test_monomial.py ........                                           [ 66%]
tests\test_polynomial.py .......                                          [ 68%]
tests\test_queues.py .....                                                [ 69%]
tests\test_search.py .............                                        [ 72%]
tests\test_set.py .                                                       [ 73%]
tests\test_sort.py ....................                                   [ 77%]
tests\test_stack.py ..........                                            [ 80%]
tests\test_streaming.py ....                                              [ 81%]
tests\test_strings.py ................................................... [ 93%]
.............                                                             [ 96%]
tests\test_tree.py ...........                                            [ 99%]
tests\test_unix.py ..                                                     [100%]

============================= 412 passed in 4.81s ==============================
```

Run 3

Calculating the average execution time of the three runs, we get:

$$T_{seq} = \frac{(4.82 + 4.61 + 4.81)}{3} = 4.75\ seconds$$

3.3 Parallel Execution Results:

For the parallel execution, I did not add the failing test cases eliminated in the previous part, since we already know they are going to fail due to intrinsic syntax/logical errors present in them. Our goal here is to identify the test cases which are failing/flaky due to parallelization. Hence, I proceeded with the exact same tests established in the previous part. I automated the process of calculating the average running time of three runs per configuration and logging the summary of the three runs of each configuration using a Python script.

The following are the outputs for all 8 configurations discussed in the previous section:

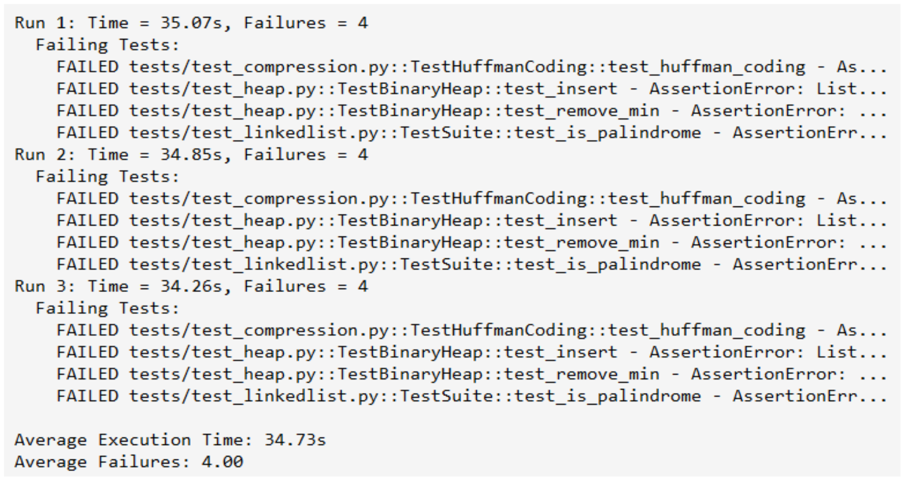
1. `--dist=load` + `--parallel-threads 1` + `-n 1`

```
Run 1: Time = 5.99s, Failures = 0
Run 2: Time = 5.51s, Failures = 0
Run 3: Time = 5.94s, Failures = 0

Average Execution Time: 5.81s
Average Failures: 0.00
```

2. `--dist=load` + `--parallel-threads 1` + `-n auto`

```
Run 1: Time = 6.72s, Failures = 0
Run 2: Time = 8.11s, Failures = 0
Run 3: Time = 10.08s, Failures = 0

Average Execution Time: 8.30s
Average Failures: 0.00
```

3. `--dist=load` + `--parallel-threads auto` + `-n 1`



```
Run 1: Time = 35.07s, Failures = 4
  Failing Tests:
    FAILED tests/test_compression.py::TestHuffmanCoding::test_huffman_coding - As...
    FAILED tests/test_heap.py::TestBinaryHeap::test_insert - AssertionError: List...
    FAILED tests/test_heap.py::TestBinaryHeap::test_remove_min - AssertionError: ...
    FAILED tests/test_linkedlist.py::TestSuite::test_is_palindrome - AssertionErr...
Run 2: Time = 34.85s, Failures = 4
  Failing Tests:
    FAILED tests/test_compression.py::TestHuffmanCoding::test_huffman_coding - As...
    FAILED tests/test_heap.py::TestBinaryHeap::test_insert - AssertionError: List...
    FAILED tests/test_heap.py::TestBinaryHeap::test_remove_min - AssertionError: ...
    FAILED tests/test_linkedlist.py::TestSuite::test_is_palindrome - AssertionErr...
Run 3: Time = 34.26s, Failures = 4
  Failing Tests:
    FAILED tests/test_compression.py::TestHuffmanCoding::test_huffman_coding - As...
    FAILED tests/test_heap.py::TestBinaryHeap::test_insert - AssertionError: List...
    FAILED tests/test_heap.py::TestBinaryHeap::test_remove_min - AssertionError: ...
    FAILED tests/test_linkedlist.py::TestSuite::test_is_palindrome - AssertionErr...

Average Execution Time: 34.73s
Average Failures: 4.00
```

4. `--dist=load` + `--parallel-threads auto` + `-n auto`

```
Run 1: Time = 48.04s, Failures = 3
  Failing Tests:
    FAILED tests/test_heap.py::TestBinaryHeap::test_insert - AssertionError: List...
    FAILED tests/test_linkedlist.py::TestSuite::test_is_palindrome - AssertionErr...
    FAILED tests/test_heap.py::TestBinaryHeap::test_remove_min - AssertionError: ...
Run 2: Time = 50.97s, Failures = 4
  Failing Tests:
    FAILED tests/test_heap.py::TestBinaryHeap::test_insert - AssertionError: List...
    FAILED tests/test_heap.py::TestBinaryHeap::test_remove_min - AssertionError: ...
    FAILED tests/test_linkedlist.py::TestSuite::test_is_palindrome - AssertionErr...
    FAILED tests/test_compression.py::TestHuffmanCoding::test_huffman_coding - As...
Run 3: Time = 43.66s, Failures = 4
  Failing Tests:
    FAILED tests/test_heap.py::TestBinaryHeap::test_insert - AssertionError: List...
    FAILED tests/test_heap.py::TestBinaryHeap::test_remove_min - AssertionError: ...
    FAILED tests/test_linkedlist.py::TestSuite::test_is_palindrome - AssertionErr...
    FAILED tests/test_compression.py::TestHuffmanCoding::test_huffman_coding - As...

Average Execution Time: 47.56s
Average Failures: 3.67
```

5. `--dist=no` + `--parallel-threads 1` + `-n 1`

```
Run 1: Time = 5.59s, Failures = 0
Run 2: Time = 6.43s, Failures = 0
Run 3: Time = 7.12s, Failures = 0

Average Execution Time: 6.38s
Average Failures: 0.00
```

6. `--dist=no` + `--parallel-threads 1` + `-n auto`

```
Run 1: Time = 16.91s, Failures = 0
Run 2: Time = 12.13s, Failures = 0
Run 3: Time = 14.96s, Failures = 0

Average Execution Time: 14.67s
Average Failures: 0.00
```

7. `--dist=no` + `--parallel-threads auto` + `-n 1`

```
Run 1: Time = 39.24s, Failures = 4
  Failing Tests:
    FAILED tests/test_compression.py::TestHuffmanCoding::test_huffman_coding - As...
    FAILED tests/test_heap.py::TestBinaryHeap::test_insert - AssertionError: List...
    FAILED tests/test_heap.py::TestBinaryHeap::test_remove_min - AssertionError: ...
    FAILED tests/test_linkedlist.py::TestSuite::test_is_palindrome - AssertionErr...
Run 2: Time = 54.70s, Failures = 4
  Failing Tests:
    FAILED tests/test_compression.py::TestHuffmanCoding::test_huffman_coding - As...
    FAILED tests/test_heap.py::TestBinaryHeap::test_insert - AssertionError: List...
    FAILED tests/test_heap.py::TestBinaryHeap::test_remove_min - AssertionError: ...
    FAILED tests/test_linkedlist.py::TestSuite::test_is_palindrome - AssertionErr...
Run 3: Time = 55.94s, Failures = 4
  Failing Tests:
    FAILED tests/test_compression.py::TestHuffmanCoding::test_huffman_coding - As...
    FAILED tests/test_heap.py::TestBinaryHeap::test_insert - AssertionError: List...
    FAILED tests/test_heap.py::TestBinaryHeap::test_remove_min - AssertionError: ...
    FAILED tests/test_linkedlist.py::TestSuite::test_is_palindrome - AssertionErr...

Average Execution Time: 49.96s
Average Failures: 4.00
```

8. `--dist=no` + `--parallel-threads auto` + `-n auto`

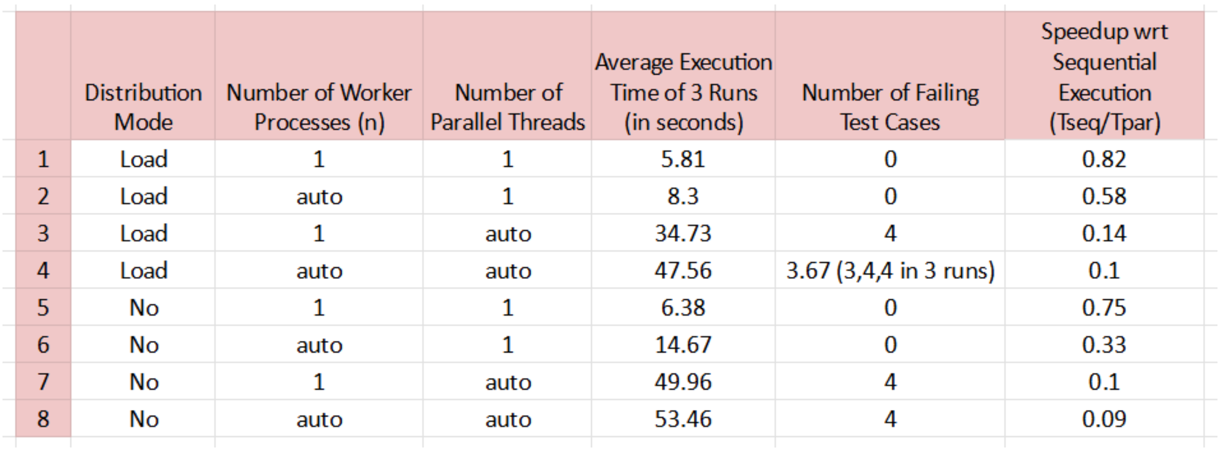
```
Run 1: Time = 52.28s, Failures = 4
  Failing Tests:
    FAILED tests/test_heap.py::TestBinaryHeap::test_insert - AssertionError: List...
    FAILED tests/test_heap.py::TestBinaryHeap::test_remove_min - AssertionError: ...
    FAILED tests/test_linkedlist.py::TestSuite::test_is_palindrome - AssertionErr...
    FAILED tests/test_compression.py::TestHuffmanCoding::test_huffman_coding - As...
Run 2: Time = 54.53s, Failures = 4
  Failing Tests:
    FAILED tests/test_heap.py::TestBinaryHeap::test_insert - AssertionError: List...
    FAILED tests/test_heap.py::TestBinaryHeap::test_remove_min - AssertionError: ...
    FAILED tests/test_linkedlist.py::TestSuite::test_is_palindrome - AssertionErr...
    FAILED tests/test_compression.py::TestHuffmanCoding::test_huffman_coding - As...
Run 3: Time = 53.58s, Failures = 4
  Failing Tests:
    FAILED tests/test_heap.py::TestBinaryHeap::test_insert - AssertionError: List...
    FAILED tests/test_heap.py::TestBinaryHeap::test_remove_min - AssertionError: ...
    FAILED tests/test_linkedlist.py::TestSuite::test_is_palindrome - AssertionErr...
    FAILED tests/test_compression.py::TestHuffmanCoding::test_huffman_coding - As...

Average Execution Time: 53.46s
Average Failures: 4.00
```

`---------------------------------------------------------------`

It was observed that ***even after removing the deterministically failing test cases (the ones identified during sequential execution, failing due to syntax/logical errors) before running the tests in parallel, some tests were failing non-deterministically during parallel execution. These are known as flaky test cases, and are a drawback of parallel testing.***

3.4 Average Parallel Execution Time in Each Parallel Testing Configuration and Failing (Flaky) Cases:.

The following table presents the results of the runs of each parallel configuration tested:



	Distribution Mode	Number of Worker Processes (n)	Number of Parallel Threads	Average Execution Time of 3 Runs (in seconds)	Number of Failing Test Cases	Speedup wrt Sequential Execution (Tseq/Tpar)
1	Load	1	1	5.81	0	0.82
2	Load	auto	1	8.3	0	0.58
3	Load	1	auto	34.73	4	0.14
4	Load	auto	auto	47.56	3.67 (3,4,4 in 3 runs)	0.1
5	No	1	1	6.38	0	0.75
6	No	auto	1	14.67	0	0.33
7	No	1	auto	49.96	4	0.1
8	No	auto	auto	53.46	4	0.09

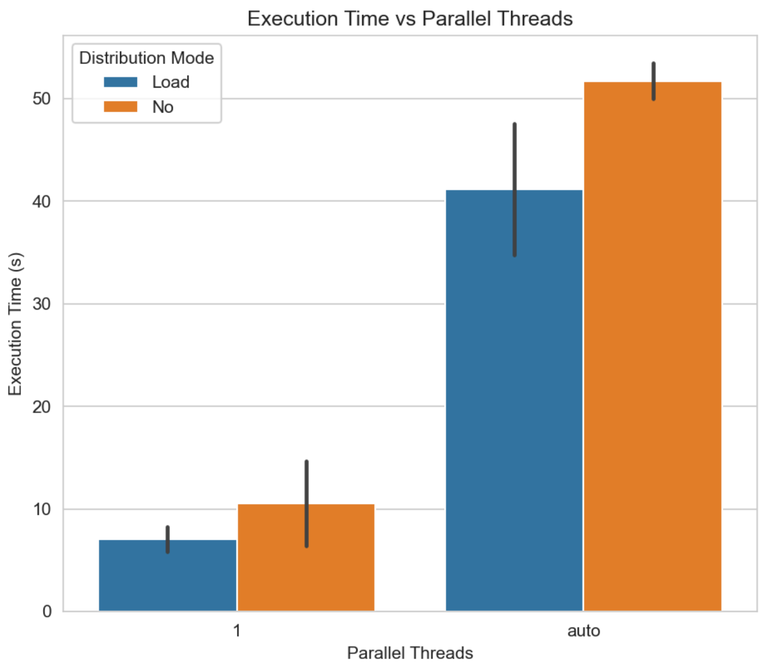
Execution Matrix for all 8 Parallelization Configurations Tested and their Speedup with respect to Sequential Execution Time Calculated Above

Possible Reasons for Parallel Execution Being Slower Than Sequential (Unexpected observation):

- Overhead: Managing multiple processes/threads adds synchronization and scheduling overhead.
- CPU Competition: Too many parallel tests can cause excessive context switching and reduce efficiency.
- Disk & I/O Bottlenecks: Parallel access to files can lead to contention, slowing tests.
- Parallel Strategy (--dist mode): Poor test distribution may lead to inefficiencies, as observed in the difference in times of both modes.
- Shared Resource Conflicts: Tests modifying the same resources may block each other, such as the one particular test case (Huffman coding) which encodes and decodes the same file using multiple threads/processes.
- System Load Issues: High CPU/memory usage from other tasks may have impacted parallel performance.

Analysis of the observations:

The following plots (and their inferences written below) represent the execution time trends and failure analysis for all the parallel testing modes:

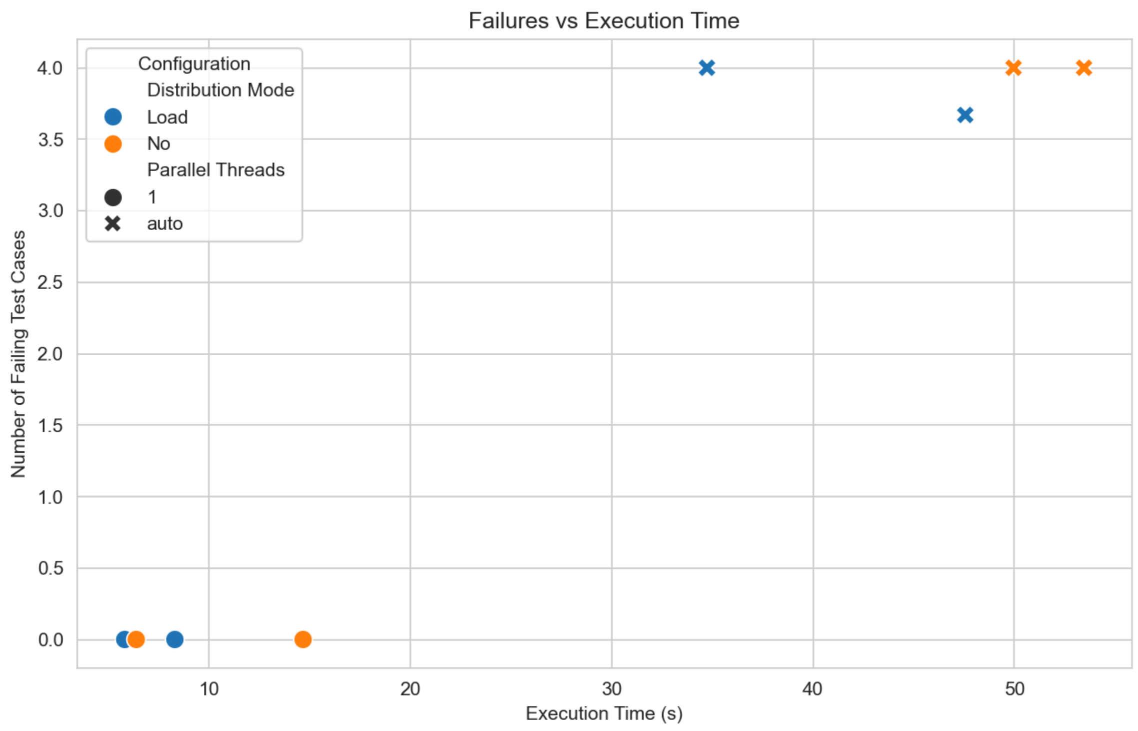


Execution Time Trends and Comparative Speedup Rates:

- **Single Worker, Single Thread (Load vs. No):** Load balancing improves performance, reducing execution time by 0.57s (~9%).
- **Multiple Workers, Single Thread (Load vs. No):** Load balancing significantly reduces execution time by 6.37s (~43%).
- **Auto Threads (Load vs. No):** Load balancing still improves execution time, but auto-threading causes a significant slowdown.
- **Fully Automatic Configuration (Load vs. No):** Load balancing offers better performance, but the gain is not as significant as in n=auto, parallel=1.

Impact of Parallelism on Execution Time:

- **Increasing parallel threads (parallel=auto) significantly increases execution time:**
    - For Load mode (n=1): 5.81s → 34.73s (***almost 6x increase***)
    - For No mode (n=1): 6.38s → 49.96s (***almost 8x increase***)
    - For Load mode (n=auto): 8.3s → 47.56s (***5.7x increase***)
    - For No mode (n=auto): 14.67s → 53.46s (***3.6x increase***)
- **Possible Causes of Slowdown:**
    - Thread competition: Too many threads competing for CPU resources.
    - Context switching overhead: More threads lead to unnecessary scheduling delays.
    - I/O-bound operations: If tests involve database access or network calls or file access (which is the case in one of the tests - huffman coding), excessive threads can increase wait time instead of reducing it.



Failure Rate Analysis:

- Single Worker, Single Thread (n=1, parallel=1):
  - No concurrency issues in this setup.
- Single Worker, Auto Threads (n=1, parallel=auto):
  - Both Load and No modes introduce 4 failures.
  - Running multiple threads in a single process introduces race conditions or non-thread-safe operations.
- Multiple Workers, Single Thread (n=auto, parallel=1):
  - Both Load and No modes have 0 failures.
  - Increasing worker processes alone does not cause failures.
- Fully Automatic (n=auto, parallel=auto):
  - Failures appear in Load mode (3, 4, 4 in different runs).
  - Consistent 4 failures in No mode. Auto-threading combined with multiple processes is the most failure-prone setup.

4. Conclusion

Based on the observations of both sequential and parallel test case execution with various configurations of distribution modes, worker processes, and parallel threads on the given dataset of programs and test suite, it can be concluded that ***this particular test suite is not suitable for test parallelization***. This conclusion is primarily based on the following key findings:

1. **Unexpected Increased Execution Time in Parallel Testing**

- Contrary to the expected performance improvement, all parallel execution configurations resulted in **higher execution times** compared to sequential execution.
- This suggests that overheads such as inter-process communication, resource contention, and scheduling inefficiencies outweigh the potential speedup gained from parallelization.
- Detailed reasoning for this behavior has been provided in the previous section.

2. **Expected Flaky Test Failures in Parallel Execution**

- Running tests in parallel introduced **non-deterministic failures** (flaky tests), which were not observed in sequential execution.
- These failures likely arise due to **shared resource conflicts**, race conditions, or improper test dependencies, leading to inconsistent test results.
- Such unpredictability reduces the reliability of parallel test execution and makes debugging more complex.

For pytest developers, the following improvements could help enhance thread safety:

- Introduce built-in concurrency checks: Warnings or errors should be raised when tests access shared resources unsafely.
- Test ordering control: Allow users to define dependency-aware execution sequences to prevent race conditions.

- Improve parallel execution strategies: Provide better default heuristics for test distribution, considering execution time and resource usage.
- Automated detection of flaky tests: Integrate mechanisms to detect non-deterministic test failures and suggest improvements.

By addressing these aspects, pytest can improve its support for robust and efficient parallel test execution.

5. References

Lecture Slides
https://pypi.org/project/pytest
https://docs.pytest.org/en/stable
https://pypi.org/project/pytest-xdist
https://pytest-xdist.readthedocs.io/en/stable
https://pypi.org/project/pytest-run-parallel
https://github.com/Quansight-Labs/pytest-run-parallel

CS202: Software Tools and Techniques

Lab 7 & Lab 8 Report: Vulnerability Analysis on Open-Source Software Repositories

Jiya Bhavin Desai | Roll No. 22110107 | 20.02.2025 & 27.02.2025

---

1. Introduction

Vulnerability analysis in open-source software is important for identifying security risks and ensuring the safety of software applications, especially for open-source projects. In this lab, we used Bandit, a static code analysis tool, to examine Python repositories for security vulnerabilities. By scanning three repositories on GitHub, I identified potential security threats and analyzed trends in vulnerability introduction, variation and resolution across the last 100 non-merge commits in each of the three repositories I selected for this lab. The objective was to enhance our understanding of vulnerability detection in open-source projects and improve our ability to analyze security reports.

1.1 Setup

- Creating a virtual environment with Python (no version compatibility was specified - implying no conflicts with the latest version. I used Python 3.12.6)
- A separate virtual environment for each repository was created to avoid dependency conflicts
- Installing the required dependencies: **bandit** (using `pip install bandit`)

1.2 Tools

- Bandit: A Python static analysis tool for detecting security vulnerabilities.
- GitHub: Where the open-source projects are hosted
- PowerShell: Automated script execution for running bandit

- SEART GitHub Search Engine: For selecting repositories based on our desired criteria

2. Methodology and Execution

2.1 Choosing three Git repositories

- We were expected to choose three large real-world projects on GitHub with a substantial number of commits that can be analysed for security vulnerability analysis using bandit.
- Along with satisfying the above criteria, I also wanted the repositories to align with my own fields of interest.
- I used the SEART GitHub search engine as recommended in this course for filtering the repositories based on my preferences and gave filters for the minimum number of commits (large-scale), minimum number of stars (to ensure popularity and significance in the open-source community) and the date of last commit to fall within recent years.
- Based on the above criteria, I chose to go ahead with the following three repositories:
    1. **MaxKB:** "Ready-to-use & flexible RAG Chatbot, supporting mainstream large language models (LLMs) such as DeepSeek-R1, Llama 3.3, Qwen2, OpenAI and more."[2]
    2. **Manim:** "A community-maintained Python framework for creating mathematical animations."[3]
    3. **Tianshou:** "An elegant PyTorch deep reinforcement learning library."[4]

2.2 Overall Workflow

- Selection of repositories based on the above-mentioned metrics.
- Setup of isolated environments for each repository.
- Installation and execution of Bandit on the last 100 non-merge commits.
- Collection and analysis of reported vulnerabilities.
- Answering the research questions asked in the lab based on the gathered data.

2.3 Cloning the repositories locally

I used the following command in my working directory for cloning my chosen repositories:

```
git clone https://github.com/repository-name
```

2.4 Detailed Execution and Analysis

A PowerShell script was used to automate the scanning process across all selected repositories. The script iterated through the repositories, ran Bandit on each commit, and saved the results for further analysis. The script collected data about the count of high, medium and low-level security vulnerabilities and their corresponding confidence levels, and the CWE (Common Weakness Enumeration) trends in the last 100 commits on the three chosen repositories.

Bandit categorizes security vulnerabilities based on their severity:

- **High Severity**: These vulnerabilities show a critical security risk. They can lead to serious consequences such as unauthorized access or complete system failure. Examples include hard-coded credentials, SQL injection, and remote code execution.
- **Medium Severity:** These vulnerabilities may not be immediately prone to exploitation but can still lead to security risks if combined with other flaws. Examples include weak encryption and insecure usage of external libraries.
- **Low Severity:** These vulnerabilities are less likely to be exploited or cause significant harm. They often simply represent bad coding practices, such as using weak hashing codes.

Bandit assigns confidence levels to vulnerabilities based on the likelihood of them being an actual security issue:

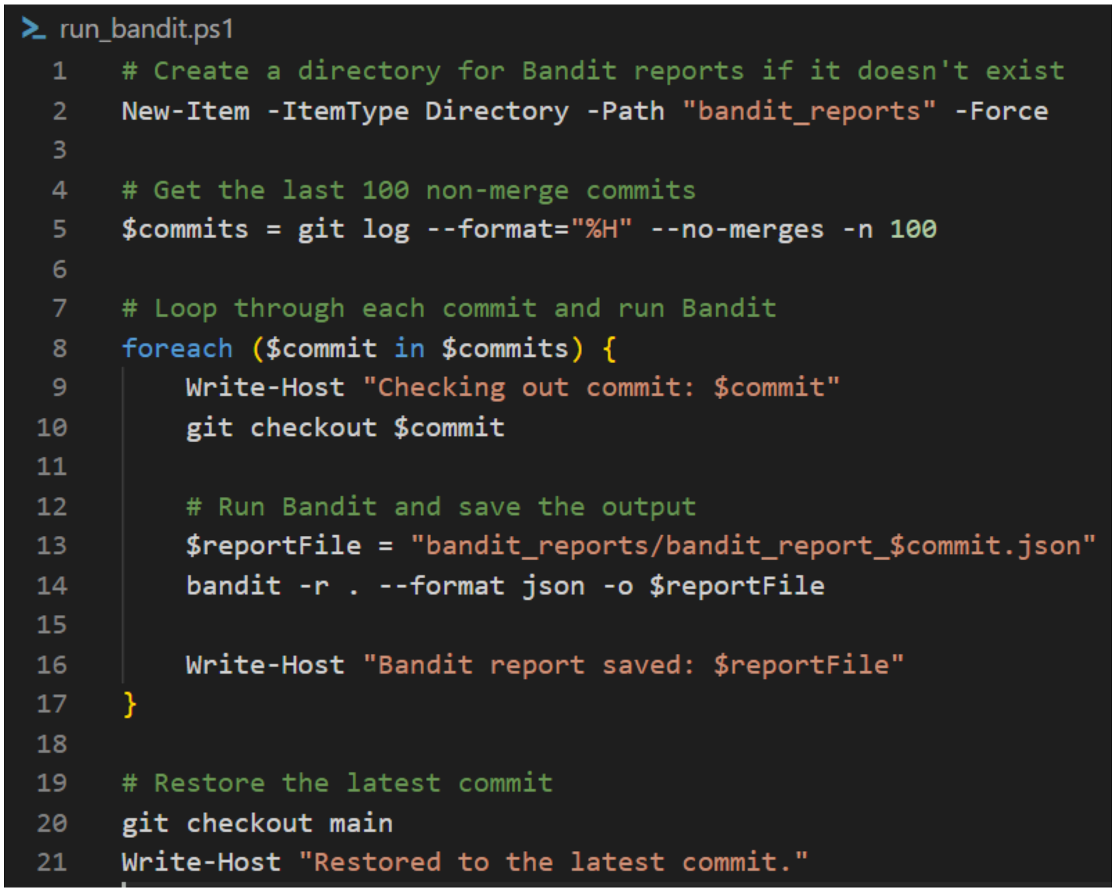
- **High Confidence:** The tool is almost certain that the detected issue is a real vulnerability.
- **Medium Confidence:** The issue is likely a security risk, but it may not be that sure about it, or about the significance of the vulnerabilities.
- **Low Confidence:** There is a higher chance of false positives when issues are identified with low confidence levels.

These confidence levels help developers prioritize fixes.

CWE (Common Weakness Enumeration) is a standardized list of software security weaknesses. Each CWE identifier represents a type of vulnerability, making it easier to categorize and track security flaws. A numeric code is associated with each type of vulnerability.

The following steps were followed for gathering the visualising the data:

1. **Collecting commit data by running run_bandit.ps**



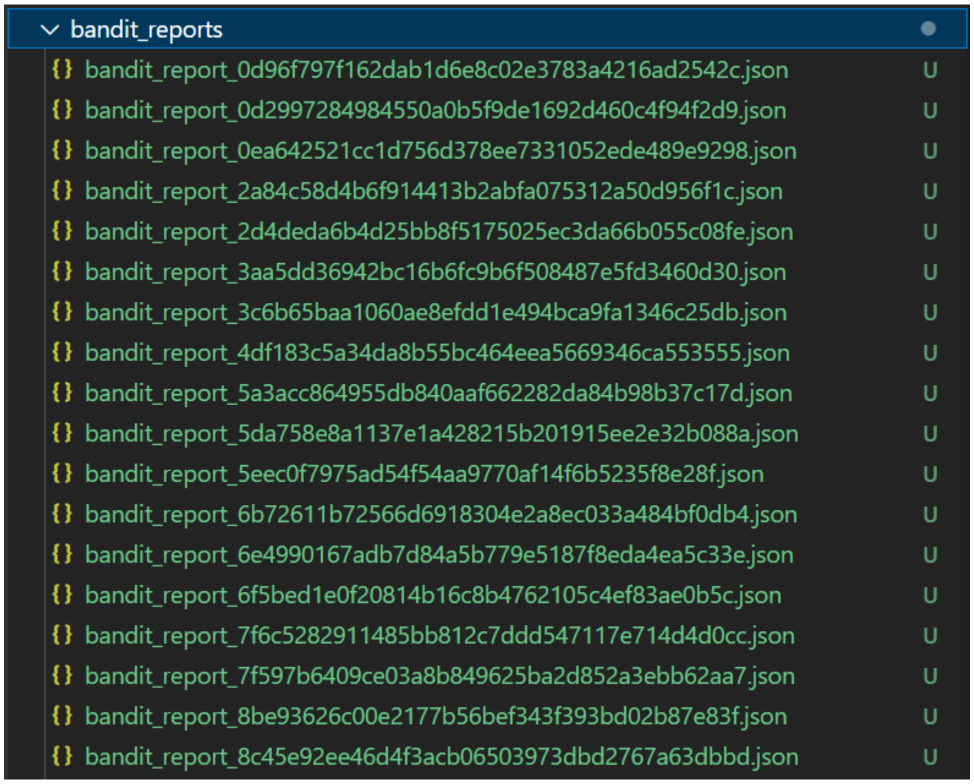
run_bandit.ps1

```powershell
# Create a directory for Bandit reports if it doesn't exist
New-Item -ItemType Directory -Path "bandit_reports" -Force

# Get the last 100 non-merge commits
$commits = git log --format="%H" --no-merges -n 100

# Loop through each commit and run Bandit
foreach ($commit in $commits) {
    Write-Host "Checking out commit: $commit"
    git checkout $commit

    # Run Bandit and save the output
    $reportFile = "bandit_reports/bandit_report_$commit.json"
    bandit -r . --format json -o $reportFile

    Write-Host "Bandit report saved: $reportFile"
}

# Restore the latest commit
git checkout main
Write-Host "Restored to the latest commit."
```

The powershell script I used for automating bandit runs

- The powershell script run_bandit.ps1 automates security scanning using Bandit on the last 100 non-merge commits in a Git repository.
- The script is to be run in the repository root directory.
- The results are stored in the bandit_reports directory.
- It creates a bandit_reports directory.

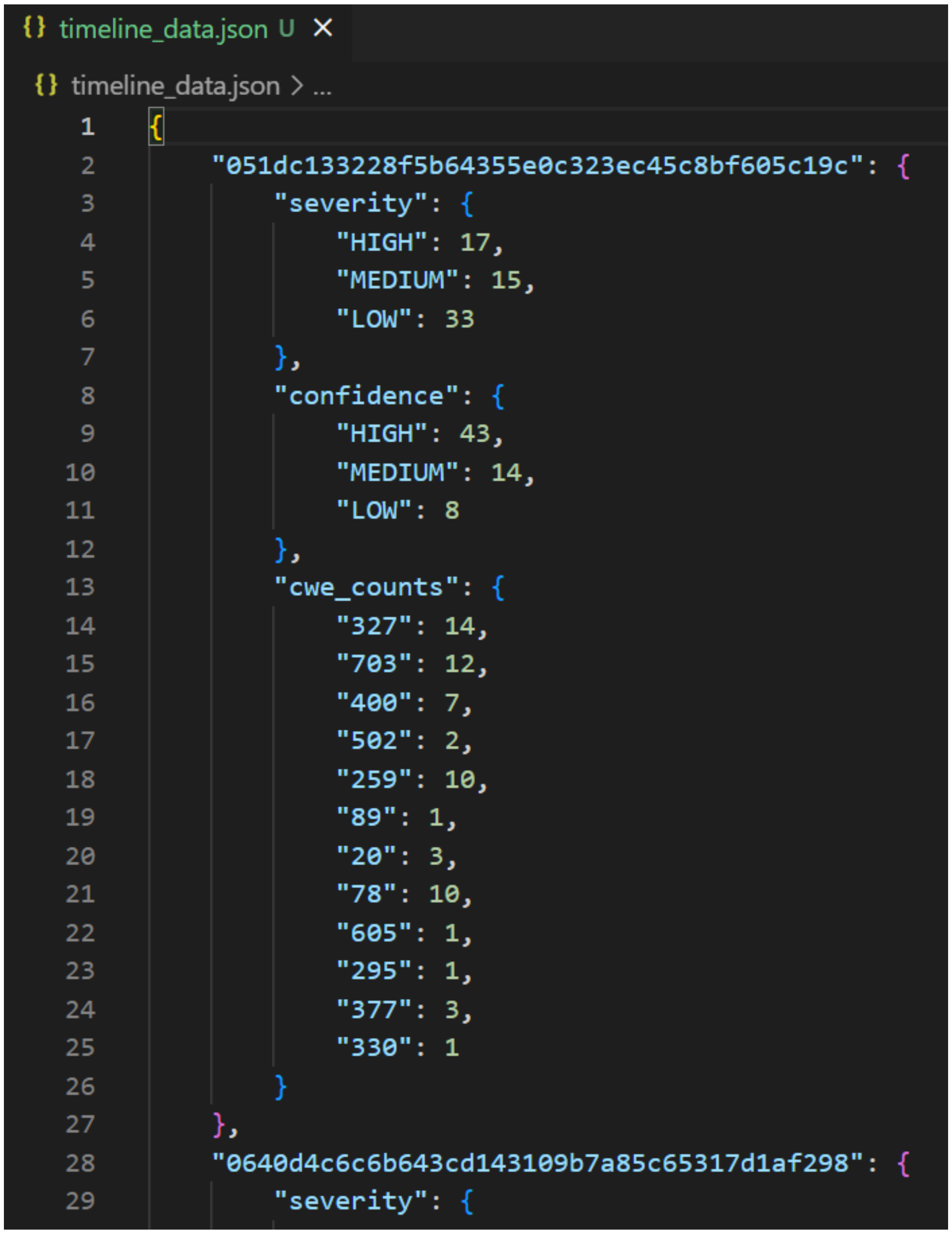
- It retrieves the last 100 commit hashes (excluding merge commits) and loops through each commit:
    - Checks out the commit.
    - Runs Bandit and saves the report in a json file.
    - Restores the latest commit (main branch).



```
bandit_reports
  bandit_report_0d96f797f162dab1d6e8c02e3783a4216ad2542c.json  U
  bandit_report_0d2997284984550a0b5f9de1692d460c4f94f2d9.json  U
  bandit_report_0ea642521cc1d756d378ee7331052ede489e9298.json  U
  bandit_report_2a84c58d4b6f914413b2abfa075312a50d956f1c.json  U
  bandit_report_2d4deda6b4d25bb8f5175025ec3da66b055c08fe.json  U
  bandit_report_3aa5dd36942bc16b6fc9b6f508487e5fd3460d30.json  U
  bandit_report_3c6b65baa1060ae8efdd1e494bca9fa1346c25db.json  U
  bandit_report_4df183c5a34da8b55bc464eea5669346ca553555.json  U
  bandit_report_5a3acc864955db840aaf662282da84b98b37c17d.json  U
  bandit_report_5da758e8a1137e1a428215b201915ee2e32b088a.json  U
  bandit_report_5eec0f7975ad54f54aa9770af14f6b5235f8e28f.json  U
  bandit_report_6b72611b72566d6918304e2a8ec033a484bf0db4.json  U
  bandit_report_6e4990167adb7d84a5b779e5187f8eda4ea5c33e.json  U
  bandit_report_6f5bed1e0f20814b16c8b4762105c4ef83ae0b5c.json  U
  bandit_report_7f6c5282911485bb812c7ddd547117e714d4d0cc.json  U
  bandit_report_7f597b6409ce03a8b849625ba2d852a3ebb62aa7.json  U
  bandit_report_8be93626c00e2177b56bef343f393bd02b87e83f.json  U
  bandit_report_8c45e92ee46d4f3acb06503973dbd2767a63dbbd.json  U
```

The generated json reports for each of the 100 commits

2. **Extracting and storing data by running analyze_bandit.py**
    - analyze_bandit.py iterates over each Bandit JSON report in the bandit_reports directory created in the above step and extracts.
        - Severity Levels: Counts high, medium, and low severity vulnerabilities.
        - Confidence Levels: Tracks how certain Bandit is about each issue.
        - CWE Categories: Identifies common vulnerability types.
    - It organizes this data into a structured dictionary and saves it as timeline_data.json.



{} timeline_data.json U ×

{} timeline_data.json > ...

```
{
    "051dc133228f5b64355e0c323ec45c8bf605c19c": {
        "severity": {
            "HIGH": 17,
            "MEDIUM": 15,
            "LOW": 33
        },
        "confidence": {
            "HIGH": 43,
            "MEDIUM": 14,
            "LOW": 8
        },
        "cwe_counts": {
            "327": 14,
            "703": 12,
            "400": 7,
            "502": 2,
            "259": 10,
            "89": 1,
            "20": 3,
            "78": 10,
            "605": 1,
            "295": 1,
            "377": 3,
            "330": 1
        }
    },
    "0640d4c6c6b643cd143109b7a85c65317d1af298": {
        "severity": {
```

A snippet from timeline_data.json for one of the repositories

3. **Visualizing the Data and Answering the questions by running generate_visualisations.py:**
    - Reads timeline_data.json
    - Extracts severity trends, confidence trends, and CWE frequency and plots graphs for these trends.

The gathered data about the severity levels, confidence levels and CWE counts was analyzed to answer the following research questions:

- **RQ1: When are vulnerabilities with high severity introduced and fixed along the development timeline in OSS repositories?**
    - Before answering this question, the definition of how a vulnerability is said to be fixed is described here (my interpretation):
    - A vulnerability is "fixed" when the code causing the issue is modified or removed so that bandit can no longer detect it.
    - **By tracking the count of reported vulnerabilities across commits, a decrease—or disappearance—of a particular issue indicates that it has been fixed, while persistent, increasing or fluctuating counts (which was the case in most observations) suggest that the issue is still present.**
    - Changes over several commits help confirm that a fix is stable and not just a temporary change or false negative.
- **RQ2: Do vulnerabilities of different severity levels follow similar introduction and elimination patterns?**
- **RQ3: Which CWEs are the most frequent across different OSS repositories?**

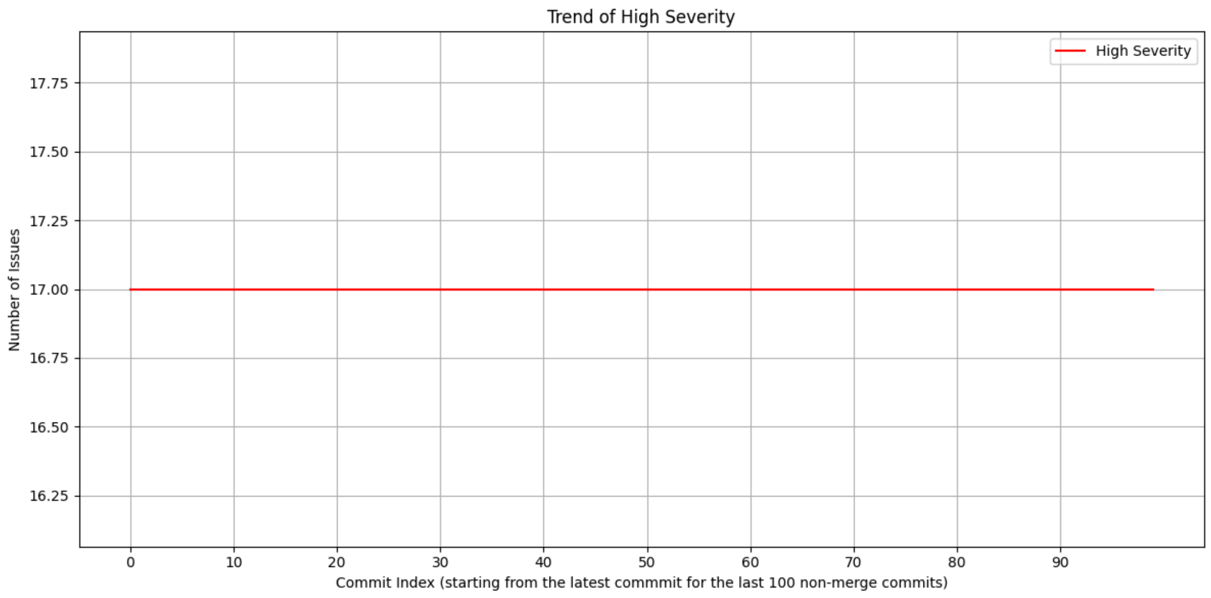
All obtained observations and results are presented in the next section.

3. Results and Analysis

3.1 Repository 1: MaxKB

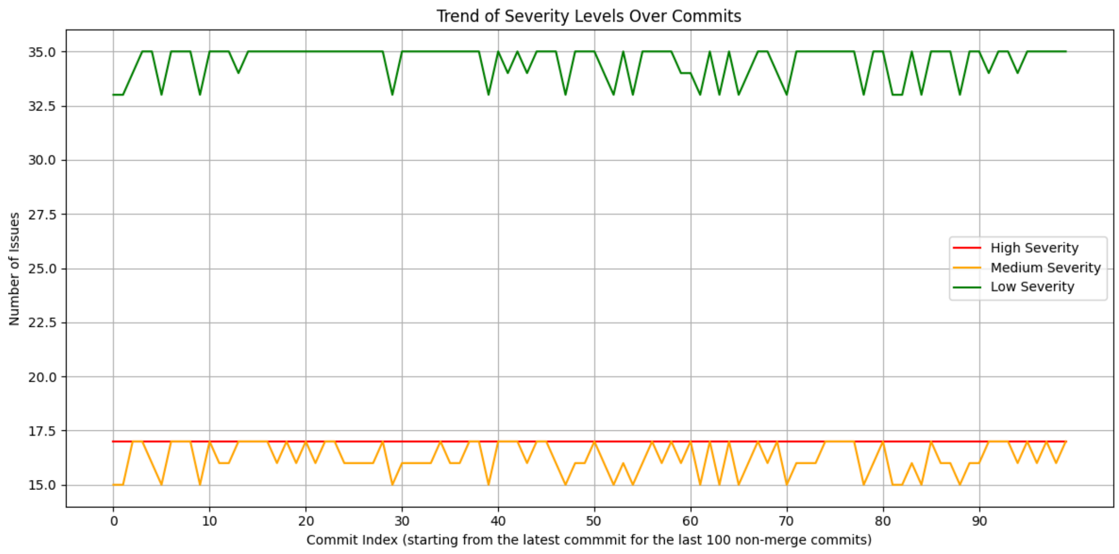
The following results were obtained for the first repository:

1. **RQ1: Trend in high severity vulnerabilities:**



The number of high severity vulnerabilities is not changing in the last 100 commits made in the repository. The developers are not addressing them - either the number was brought down significantly during the initial commits or the residual vulnerabilities, in spite of their high severity status assigned by bandit, are not affecting the working of the project. But the number 17 still seems to be high for high severity vulnerabilities.

2. **RQ2: Do vulnerabilities of different severity levels follow similar introduction and elimination patterns?**

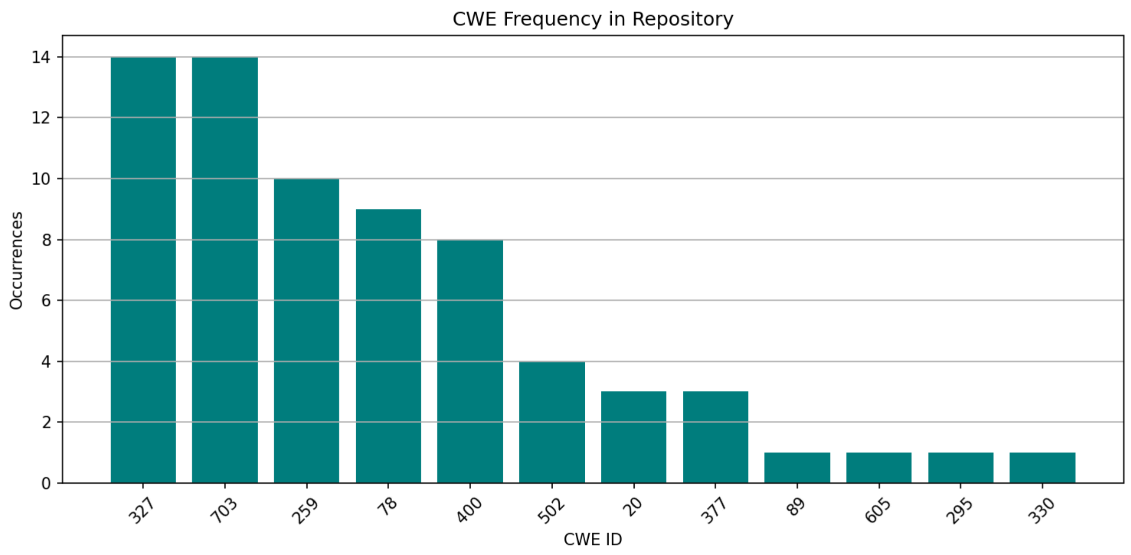


As can be seen, different severity level vulnerabilities have very different patterns of introduction and variation throughout the 100 commits we are considering.

- The number of high-severity vulnerabilities is staying constant at 17 - the developers weren't addressing them in the last 100 commits.
- The number of medium-severity vulnerabilities fluctuated between 15 and 17 - suggesting minor fixes.
- The number of low-severity vulnerabilities was the highest for every commit considered - fluctuating between 33 and 35.

3. **RQ3: Which CWEs are the most frequent across different OSS repositories?**

   The following plot shows the CWE counts for this repository:



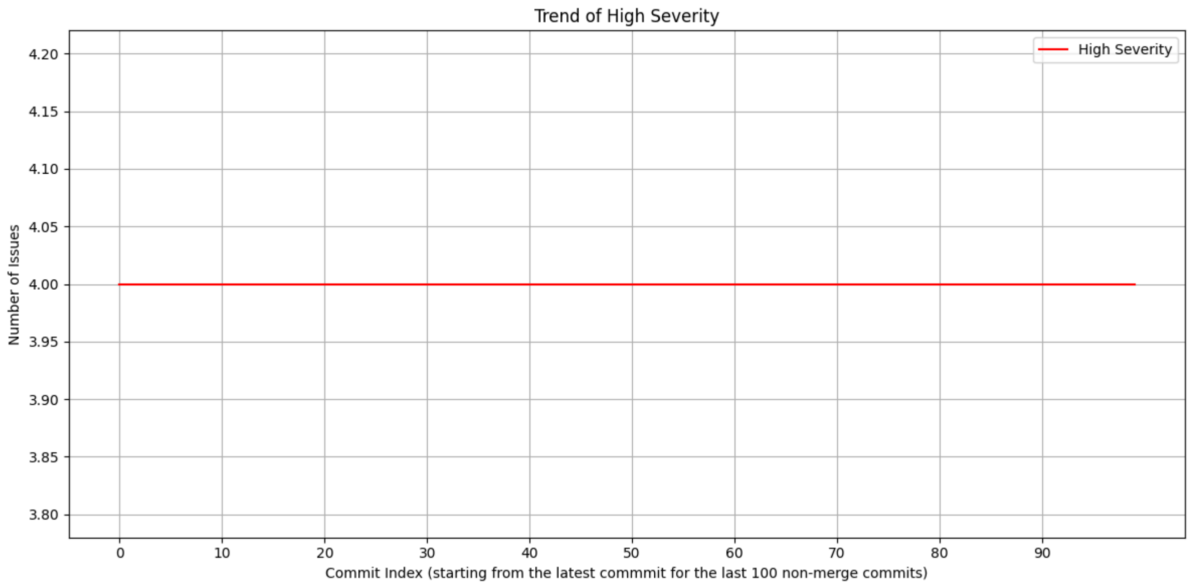
   The top 5 issues are:

   1. **ID 327**: This vulnerability occurs when an application uses cryptographic algorithms that are outdated, broken, or weak. Attackers can exploit these weaknesses to decrypt sensitive data, forge signatures, or bypass security measures.
   2. **ID 703**: This weakness occurs when an application does not correctly handle exceptions, errors, or unexpected conditions. This can lead to crashes, security bypasses, or information leaks.
   3. **ID 259:** When an application contains hard-coded credentials (for example, passwords, API keys, database credentials) in source code or configuration files.
   4. **ID 78:** This weakness occurs when the user input is not checked and cleaned before being used in system commands. Attackers can inject malicious commands and execute arbitrary code.
   5. **ID 400:** This vulnerability occurs when an application does not properly limit resource usage, allowing attackers to overload the system, leading to performance degradation or denial-of-serivice.

3.2 Repository 2: Manim

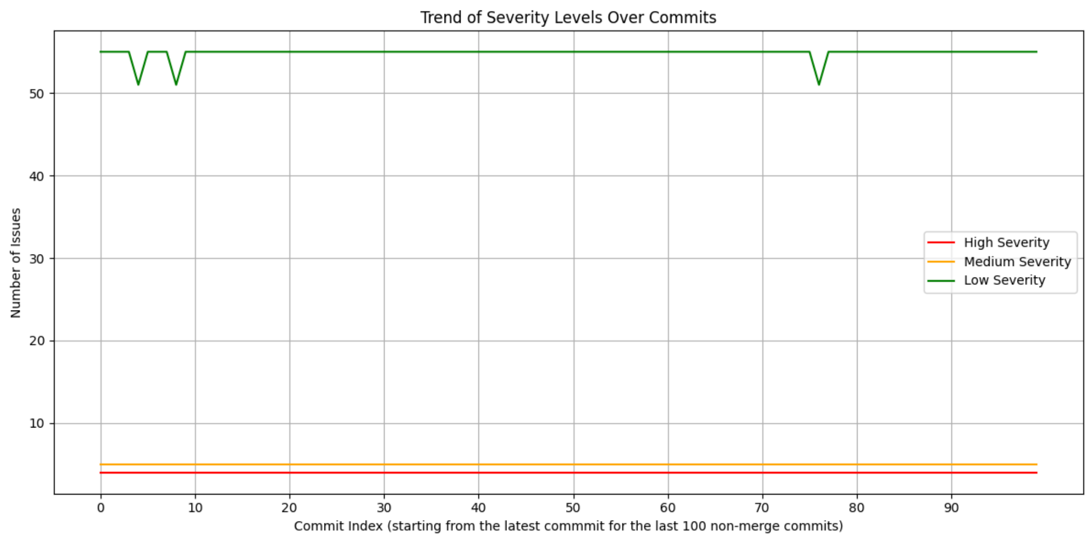
The following results were obtained for the second repository:

1. **RQ1: Trend in high severity vulnerabilities:**



Here also, the number of high severity vulnerabilities is staying the same at 4 over the last 100 commits, implying the same inference as in the case of the previous repository, although the number is much lesser here.

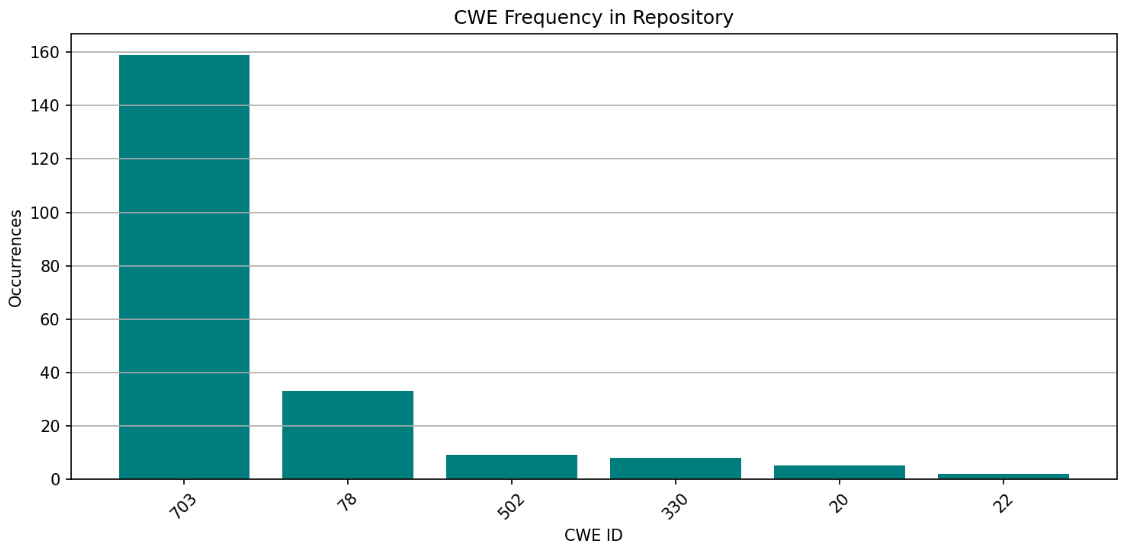
2. **RQ2: Do vulnerabilities of different severity levels follow similar introduction and elimination patterns?**



As can be seen, different severity level vulnerabilities have very different patterns of introduction and variation throughout the 100 commits we are considering.

- The number of high-severity vulnerabilities is staying constant at 4 - the developers weren't addressing them in the last 100 commits.
- The number of medium-severity vulnerabilities is also constant at 5.
- The number of low-severity vulnerabilities was the highest for every commit considered - fluctuating between 51 and 55, dipping to 55 at a few places in the timeline but then again going back to 55.

3. **RQ3: Which CWEs are the most frequent across different OSS repositories?**
   The following plot shows the CWE counts for this repository:

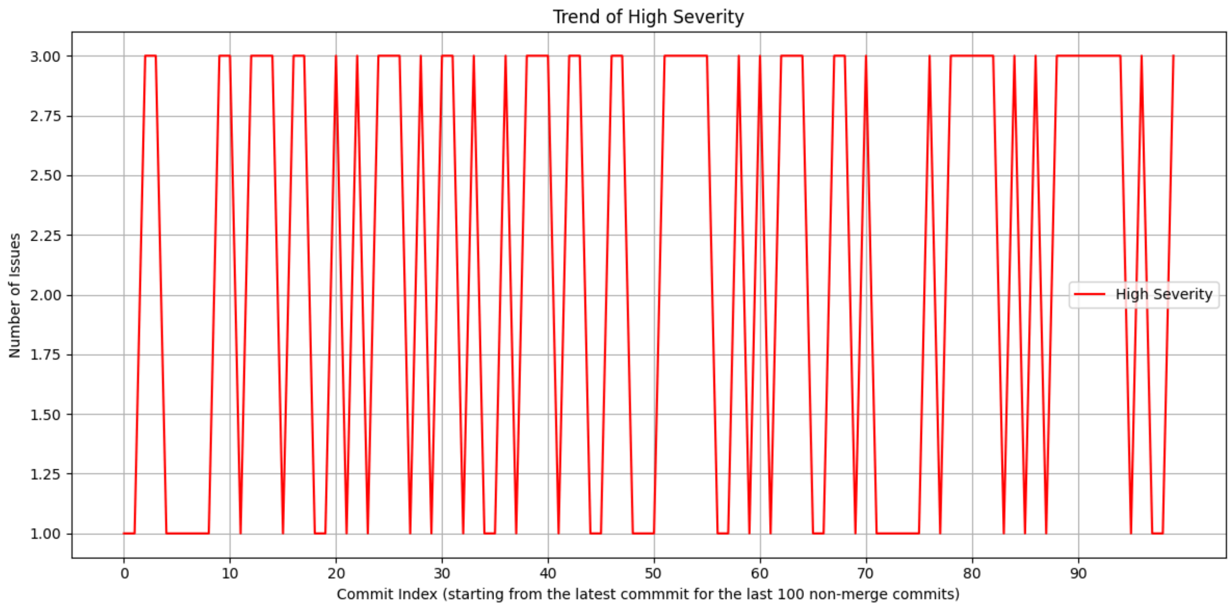


   The top 5 issues are:
   1. **ID 703:** This weakness occurs when an application does not correctly handle exceptions, errors, or unexpected conditions. This can lead to crashes, security bypasses, or information leaks.
   2. **ID 78:** This weakness occurs when the user-given input is not checked and cleaned before being used in system commands. Attackers can inject malicious commands and execute arbitrary code.
   3. **ID 330:** This weakness occurs when an application generates predictable or insufficiently random values for security-sensitive operations.
   4. **ID 502:** This vulnerability occurs when an application converts structured data back into an object without properly validating or formatting the input. Attackers can craft malicious serialized objects to execute arbitrary code or modify application behavior.
   5. **ID 20:** This vulnerability occurs when an application does not properly validate user input before processing it, leading to security issues such as SQL injection, XSS, command injection, or buffer overflows.

3.3 Repository 3:

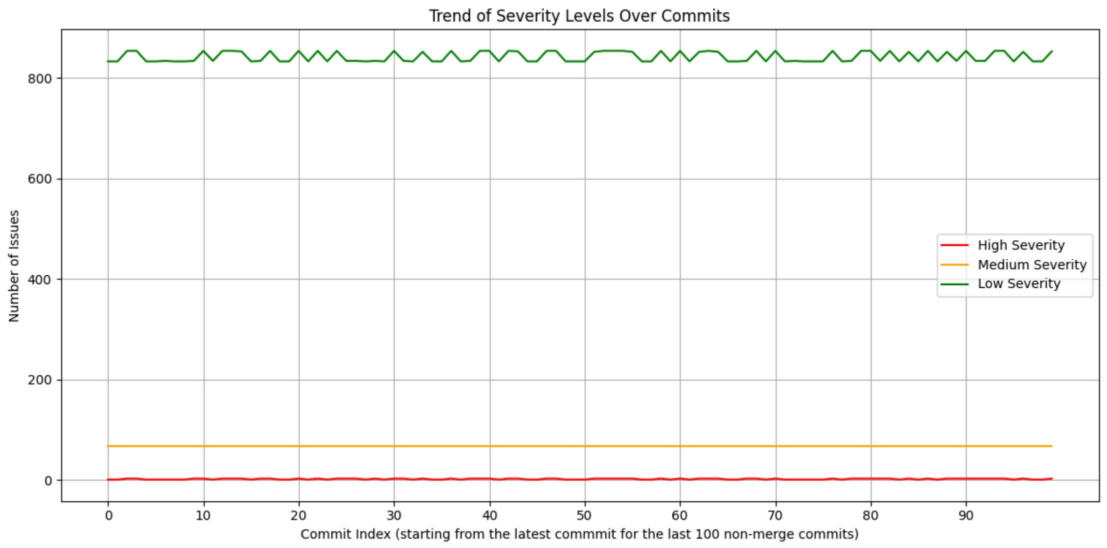
The following results were obtained for the third repository:

1. **RQ1: Trend in high severity vulnerabilities:**



Here also, the number of high severity vulnerabilities is varying between 1 and 3 over the last 100 commits - which suggests active efforts for fixing it and an overall low enough number for a high severity vulnerability.

2. **RQ2: Do vulnerabilities of different severity levels follow similar introduction and elimination patterns?**

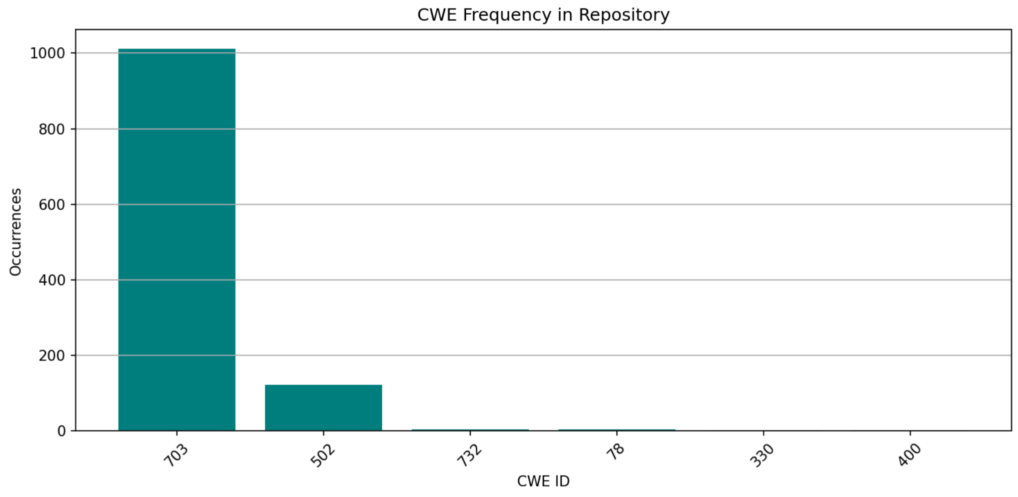


No, different severity level vulnerabilities have very different patterns of introduction and variation throughout the 100 commits we are considering.

- The number of high-severity vulnerabilities is varying between 1 and 3.
- The number of medium-severity vulnerabilities is also constant at 68.
- The number of low-severity vulnerabilities was the highest for every commit considered - fluctuating in the range of 800 across the 100 commits considered. Among the three repositories, this repository had the highest number of low-severity vulnerabilities.

3. **RQ3: Which CWEs are the most frequent across different OSS repositories?**

   The following plot shows the CWE counts for this repository:



   The top 5 issues are:

   1. **ID 703:** This weakness occurs when an application does not correctly handle exceptions, errors, or unexpected conditions. This can lead to crashes, security bypasses, or information leaks. The number of these issues seems to be extremely high in this particular repository, considering that I calculated the number of CWEs for the last 100 non-merge commits only.
   2. **ID 502:** This vulnerability occurs when an application converts structured data back into an objec without properly validating or sanitizing the input.
   3. **ID 732:** When a system assigns unrestricted access rights to files, directories, databases, or other critical resources.
   4. **ID 78:** This weakness occurs when user-supplied input is not checked and cleaned before being used in system commands. Attackers can inject malicious commands and execute arbitrary code.
   5. **ID 330:** This weakness occurs when an application generates predictable or insufficiently random values for security-sensitive operations, making it easier for attackers to guess or manipulate them.

4. Conclusion

This assignment provided a comprehensive analysis of security vulnerabilities in open-source repositories using Bandit, focusing on the detection and categorization of issues based on severity, confidence, and CWE classifications.

The data collection and analysis helped me answer the three research questions about the trends of introduction and resolution of high-severity commits in the three repositories, whether or not all levels of severities have the same trend of introduction and elimination and the most recurring CWEs across repositories.

By counting the number of CWE in 100 commits across three repositories, CWE-703 (Improper Error Handling) was identified as the most frequent one. My data collection and organisation approach ensured accurate vulnerability counts, confidence counts and CWE tracking. The visualizations highlighted trends in vulnerability introduction, persistence, variation and resolution, giving an idea about how different severity levels are handled in development.

5. References

[1] Lecture Slides

[2] https://github.com/1Panel-dev/MaxKB

[3] https://github.com/ManimCommunity/manim

[4] https://github.com/thu-ml/tianshou

[5] https://en.wikipedia.org/wiki/Common_Weakness_Enumeration

[6] https://cwe.mitre.org/about/index.html

[7] https://github.com/PyCQA/bandit