Lab Assignment 5

Lab topic: Code Coverage Analysis and Test Generation

Introduction

Objective

This lab entails calculation and analysis of different types of code coverage, as well as unit test cases generation using tools for automated tests. Students are to use an assigned dataset for Python programs and analyze some of the following tests metrics:

- Line (statement) coverage
- Branch coverage
- Function coverage

Additionally, students will also have hands-on exposure to automatic test generation tools to understand their strengths and limitations.

Learning Outcomes

By the end of this lab, students will be able to:

✓ Understand and differentiate various types of code coverage.
✓ Use automated tools to measure coverage on a given dataset of Python programs.
✓ Write/Generate effective unit tests to maximize coverage.
✓ Visualize coverage reports and analyse test effectiveness.

Tools and environmental setup

Operating System: Linux (SET-IITGN-VM)
Software Requirements:

- Python 3.10
- pytest (Testing framework) 8.3.5
- pytest-cov (Line/branch coverage analysis)
- pytest-func-cov (Function coverage analysis)
- coverage (Detailed coverage metrics) 7.7.0
- pynguin (Automated unit test generation) 0.40.0
- genhtml, lcov (Coverage visualization) 1.14

Methodology, Execution, Results and Analysis

Repository Setup & Configuring coverage tools

Firstly I have cloned the repository keon/algorithms using the following command:

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

To create and activate virtual environment I used this command:
python3.10 -m venv venv
source venv/bin/activate

```
Activities   Terminal                                   Mar 20  2:40 PM
set-iitgn-vm@set-iitgn-vm: ~/15_3/new_l5/algorithms
set-iitgn-vm@set-iitgn-vm:~/15_3/new_l5$ git clone https://github.com/keon/algorithms.git
Cloning into 'algorithms'...
remote: Enumerating objects: 5188, done.
remote: Counting objects: 100% (34/34), done.
remote: Compressing objects: 100% (20/20), done.
remote: Total 5188 (delta 23), reused 14 (delta 14), pack-reused 5154 (from 2)
Receiving objects: 100% (5188/5188), 1.44 MiB | 4.81 MiB/s, done.
Resolving deltas: 100% (3239/3239), done.
set-iitgn-vm@set-iitgn-vm:~/15_3/new_l5$ cd algorithms
set-iitgn-vm@set-iitgn-vm:~/15_3/new_l5/algorithms$ python3.10 -m venv venv
set-iitgn-vm@set-iitgn-vm:~/15_3/new_l5/algorithms$ source venv/bin/activate
```

To install requirements required to run the given repository, I ran these commands:
pip install -e .
python -c “import algorithms”

```
Activities   Terminal                                   Mar 20  2:44 PM
set-iitgn-vm@set-iitgn-vm: ~/15_3/new_l5/algorithms
(venv) set-iitgn-vm@set-iitgn-vm:~/15_3/new_l5/algorithms$ pip install -e .
Obtaining file:///home/set-iitgn-vm/15_3/new_l5/algorithms
  Preparing metadata (setup.py) ... done
Installing collected packages: algorithms
  Running setup.py develop for algorithms
Successfully installed algorithms-0.1.4

[notice] A new release of pip is available: 23.0.1 -> 25.0.1
[notice] To update, run: pip install --upgrade pip
(venv) set-iitgn-vm@set-iitgn-vm:~/15_3/new_l5/algorithms$ python -c "import algorithms"
(venv) set-iitgn-vm@set-iitgn-vm:~/15_3/new_l5/algorithms$
```

This is the command to install necessary dependencies:
pip install pytest pynguin

To get the current commit hash, I used the following command:
git log -1 --format=%H
Current commit hash: cad4754bc71742c2d6fcbd3b92ae74834d359844

```
(venv) set-iitgn-vm@set-iitgn-vm:~/15_3/new_l5/algorithms$ git log -1 --format=%H
cad4754bc71742c2d6fcbd3b92ae74834d359844
(venv) set-iitgn-vm@set-iitgn-vm:~/15_3/new_l5/algorithms$
```

Also I used this command:
pip install pytest-cov pytest-func-cov coverage

```
(venv) set-iitgn-vm@set-iitgn-vm:~/15_3/new_l5/algorithms$ pip install pytest pytest-cov pytest-func-cov coverage pynguin
Collecting pytest
  Using cached pytest-8.3.5-py3-none-any.whl (343 kB)
Collecting pytest-cov
  Using cached pytest_cov-6.0.0-py3-none-any.whl (22 kB)
Collecting pytest-func-cov
  Using cached pytest_func_cov-0.2.3-py3-none-any.whl (8.4 kB)
Collecting coverage
  Using cached coverage-7.7.0-cp310-cp310-manylinux_2_5_x86_64.manylinux1_x86_64.manylinux_2_17_x86_64.manylinux2014_x86_64.whl (239 kB)
Collecting pynguin
  Using cached pynguin-0.40.0-py3-none-any.whl (306 kB)
Collecting iniconfig
  Using cached iniconfig-2.1.0-py3-none-any.whl (6.0 kB)
Collecting pluggy<2,>=1.5
  Using cached pluggy-1.5.0-py3-none-any.whl (20 kB)
Collecting tomli>=1
  Using cached tomli-2.2.1-py3-none-any.whl (14 kB)
Collecting exceptiongroup>=1.0.0rc8
  Using cached exceptiongroup-1.2.2-py3-none-any.whl (16 kB)
Collecting packaging
  Using cached packaging-24.2-py3-none-any.whl (65 kB)
Collecting simple-parsing<0.2.0,>=0.1.6
  Using cached simple_parsing-0.1.7-py3-none-any.whl (112 kB)
Collecting asciitree<0.4.0,>=0.3.3
  Using cached asciitree-0.3.3-py3-none-any.whl
Collecting typing_inspect<0.10.0,>=0.9.0
  Using cached typing_inspect-0.9.0-py3-none-any.whl (8.8 kB)
Collecting rich<14.0.0,>=13.9.4
  Using cached rich-13.9.4-py3-none-any.whl (242 kB)
Collecting bytecode<0.17.0,>=0.16.0
  Using cached bytecode-0.16.1-py3-none-any.whl (41 kB)
Collecting astroid<4.0.0,>=3.3.5
  Using cached astroid-3.3.9-py3-none-any.whl (275 kB)
Collecting Pygments<3.0.0,>=2.18.0
  Using cached pygments-2.19.1-py3-none-any.whl (1.2 MB)
Collecting Jinja2<4.0.0,>=3.1.4
  Using cached jinja2-3.1.6-py3-none-any.whl (134 kB)
Collecting jellyfish<2.0.0,>=1.1.0
```

Executing Existing Test Cases (Test Suite A)

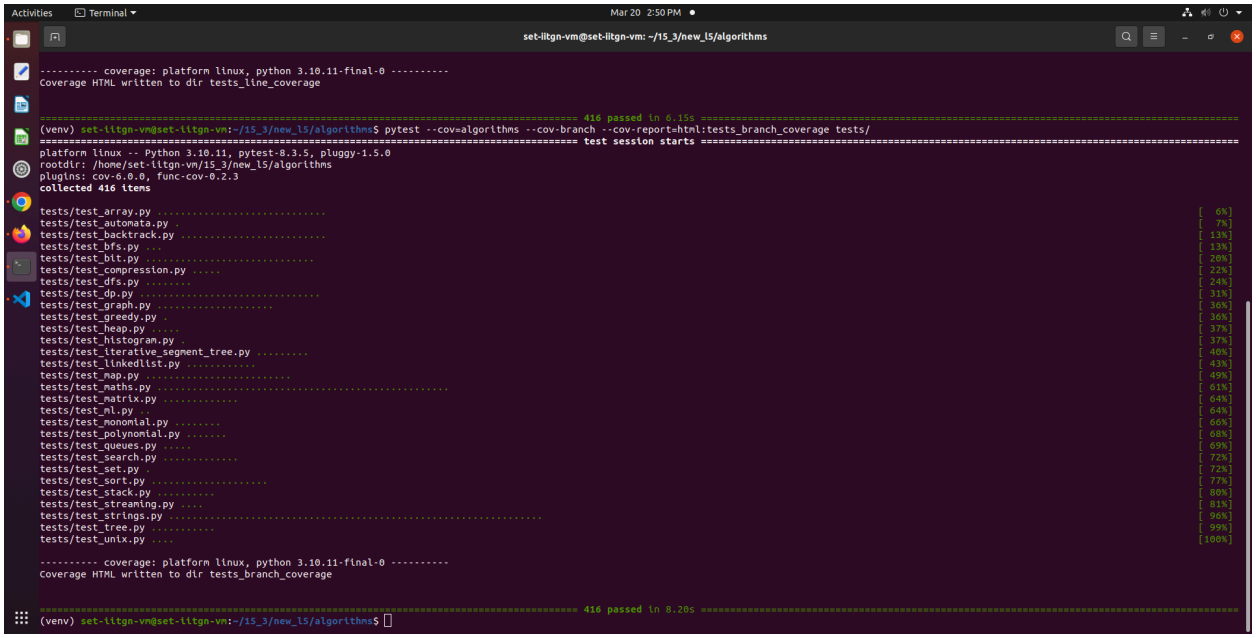
I executed and recorded the coverage metrics of Test Suite A which was provided in the repository using the following steps:

Line Coverage:

```
(venv) set-iitgn-vm@set-iitgn-vm:~/15_3/new_l5/algorithms$ pytest --cov=algorithms --cov-report=html:tests_line_coverage tests/
========================================================================== test session starts ==========================================================================
platform linux -- Python 3.10.11, pytest-8.3.5, pluggy-1.5.0
rootdir: /home/set-iitgn-vm/15_3/new_l5/algorithms
plugins: cov-6.0.0, func-cov-0.2.3
collected 416 items

tests/test_array.py ..............................                                                                                                                    [  6%]
tests/test_automata.py ..                                                                                                                                            [  7%]
tests/test_backtrack.py ...........................                                                                                                                  [ 13%]
tests/test_bfs.py ...                                                                                                                                                [ 13%]
tests/test_bit.py ................................                                                                                                                   [ 20%]
tests/test_compression.py .....                                                                                                                                      [ 22%]
tests/test_dfs.py ........                                                                                                                                           [ 24%]
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 .............                                                                                                                               [ 43%]
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 ..                                                                                                                                                 [ 72%]
tests/test_sort.py ......................                                                                                                                            [ 77%]
tests/test_stack.py .........                                                                                                                                        [ 80%]
tests/test_streaming.py ....                                                                                                                                         [ 81%]
tests/test_strings.py ......................................................................                                                                         [ 96%]
tests/test_tree.py ...........                                                                                                                                       [ 99%]
tests/test_unix.py ....                                                                                                                                              [100%]

---------- coverage: platform linux, python 3.10.11-final-0 ----------
Coverage HTML written to dir tests_line_coverage


========================================================================== 416 passed in 6.15s ==========================================================================
(venv) set-iitgn-vm@set-iitgn-vm:~/15_3/new_l5/algorithms$
```

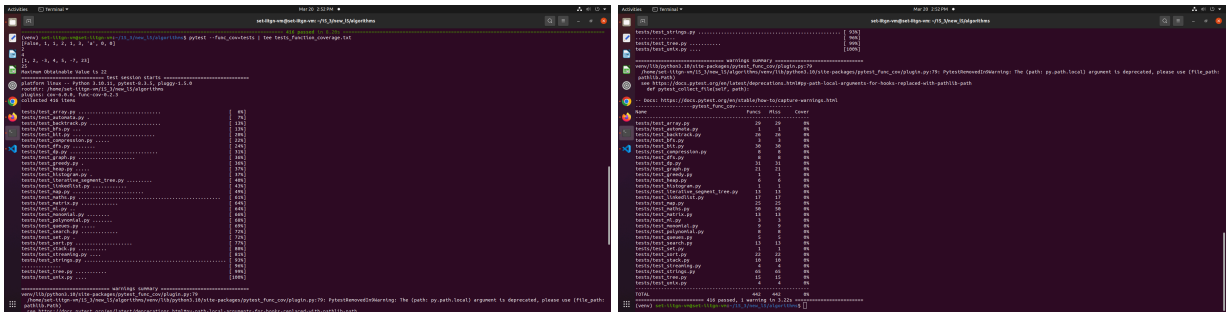
Line coverage report for Test Suite A can be found in tests_line_coverage folder.

Branch Coverage:



Branch coverage report for Test Suite A can be found in tests_branch_coverage folder.

Function Coverage:

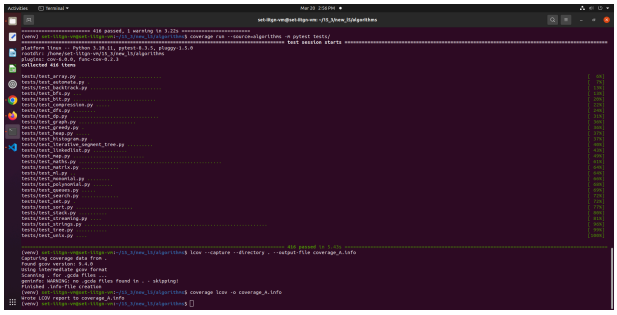


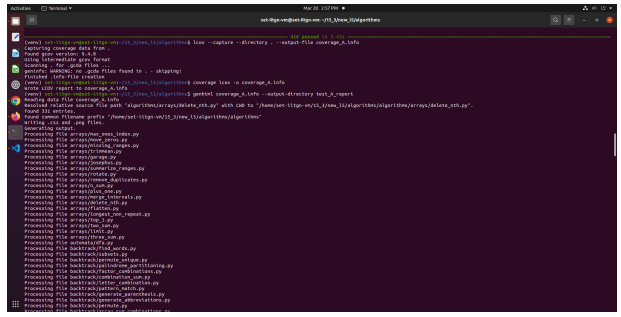
Function coverage report for Test Suite A can be found in tests_function_coverage.txt file.

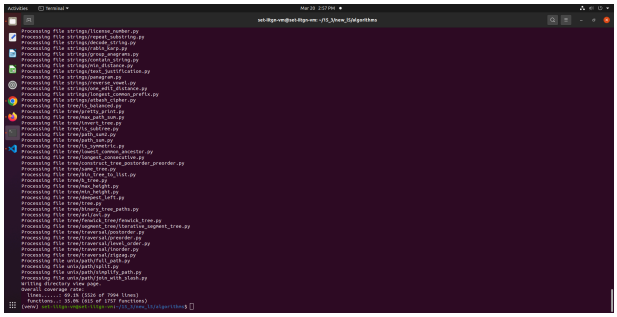
Analyzing Coverage of Test Suite A

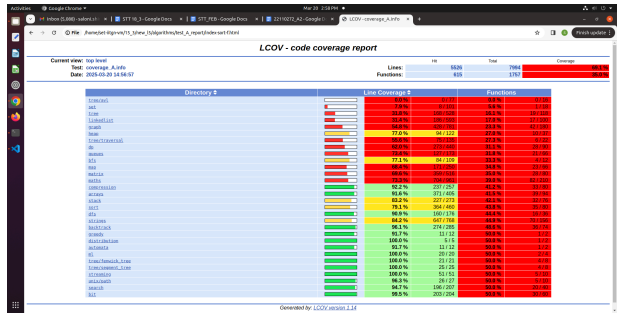
I used the following commands for visualization:

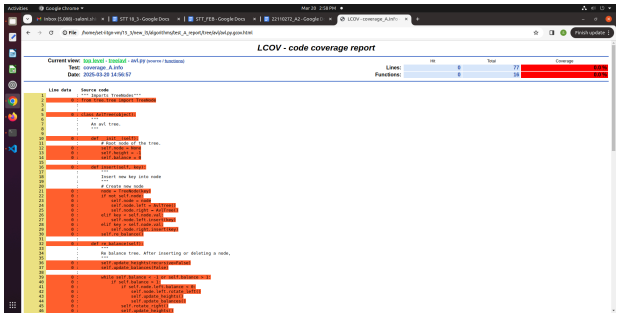
coverage run --source=algorithms -m pytest tests/
lcov --capture --directory . --output-file coverage_A.info
coverage lcov -o coverage_A.info
genhtml coverage_A.info --output-directory test_A_report
xdg-open test_A_report/index.html

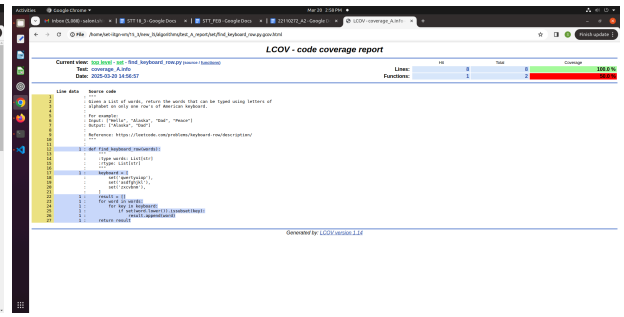


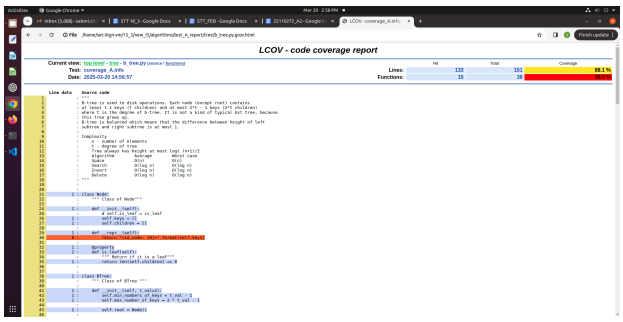












Orange color indicates those lines are not covered.

The line coverage is moderate but function coverage is very low which indicates that many functions are not being tested.

These are some worst performing directories:

- tree/avl: 0.0% line and function coverage (not tested at all)
- set: 7.9% line, 5.6% function coverage
- tree: 31.8% line, 16.1% function coverage
- linkedlist: 31.4% line, 17.0% function coverage
- graph: 54.8% line, 23.3% function coverage

These are some well tested directories:

- ml: 100.0% line and 50.0% function coverage
- tree/fenwick_tree: 100.0% line and 50.0% function coverage
- tree/segment_tree: 100.0% line and 50.0% function coverage
- search: 94.7% line, 50.0% function coverage
- unix/path: 96.3% line, 50.0% function coverage
- search: 94.7% line, 50.0% function coverage
- bit: 99.5% line, 50.0% function coverage

Tree and graph structures are poorly tested. Some components are well tested while others are almost ignored.

Generating Additional Test Cases (Test Suite B)

I used *get_low_coverage_files.py* file to get files with less than 50% line coverage or files with less than 50% branch coverage:

```python
from bs4 import BeautifulSoup

report_path = "tests_branch_coverage/index.html"
output_path = "low_coverage_0_10.txt"

with open(report_path, "r", encoding="utf-8") as file:
    parsed_html = BeautifulSoup(file, "html.parser")

coverage_rows = parsed_html.select("tr.region")
low_coverage = []

# Check coverage percentage
for row in coverage_rows:
    cells = row.find_all("td")
    if len(cells) < 5:
        continue
    file_name = cells[0].get_text(strip=True)
    coverage_text = cells[4].get_text(strip=True)
```

```python
    file_name = cells[0].get_text(strip=True)
    coverage_text = cells[4].get_text(strip=True)
    try:
        coverage_percent = float(coverage_text.rstrip("%").strip())
    except ValueError:
        continue
    if 0 < coverage_percent <= 10:
        low_coverage.append((file_name, coverage_percent))

# Save the results to an output file
with open(output_path, "w", encoding="utf-8") as output_file:
    for file, coverage in low_coverage:
        output_file.write(f"{file}: {coverage:.2f}%\n")

print(f"Results saved to '{output_path}'")
```

Then I used *generate_tests_B.py* file to generate tests using *pynguin* and saves them in tests_B folder.

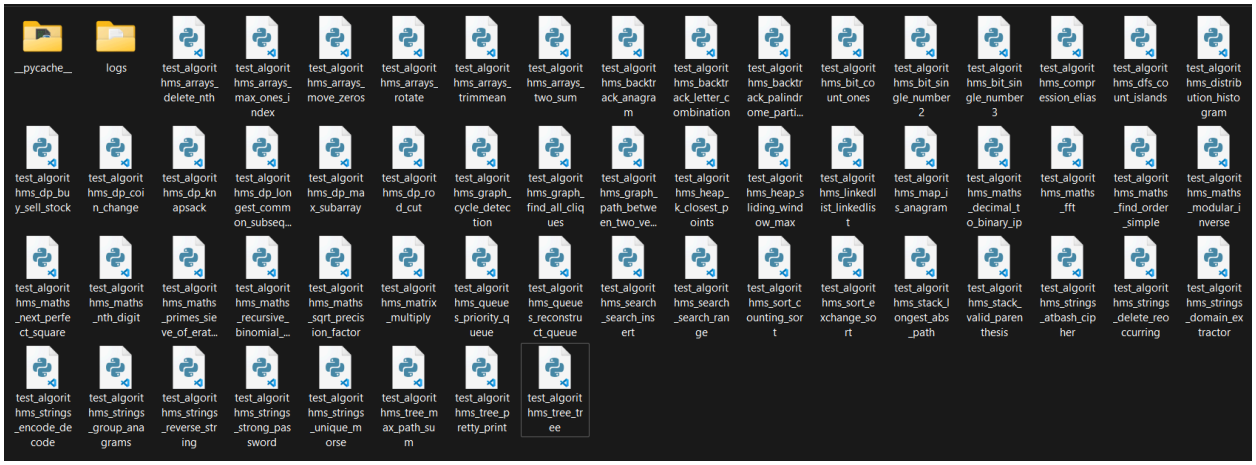
```python
import os
import subprocess

COVERAGE_REPORT = "low_coverage_0_10_branch.txt"
OUTPUT_FOLDER = "tests_B"
LOG_FOLDER = os.path.join(OUTPUT_FOLDER, "logs")
TIME_LIMIT = 30  # Timeout in seconds for each module

os.makedirs(OUTPUT_FOLDER, exist_ok=True)
os.makedirs(LOG_FOLDER, exist_ok=True)

with open(COVERAGE_REPORT, "r") as file:
    low_coverage_modules = [line.split(":")[0].strip() for line in file]

print(f"Found {len(low_coverage_modules)} Python files with low coverage.")

def execute_command_with_timeout(command, log_filepath, timeout_duration=TIME_LIMIT):
    """Executes a command and logs the output, with a timeout."""
    with open(log_filepath, "w") as log_file:
        process = subprocess.Popen(command, stdout=log_file, stderr=log_file)
        try:
            process.wait(timeout=timeout_duration)
            return process.returncode
        except subprocess.TimeoutExpired:
            process.kill()
            log_file.write(f"\n[ERROR] Timeout occurred while executing:\n{' '.join(command)}\n")
            return -1
```

```python
PROJECT_ROOT = "."

for module_file in low_coverage_modules:
    module_name = module_file.replace("/", ".").replace(".py", "")
    log_filepath = os.path.join(LOG_FOLDER, f"{module_name}.log")

    pynguin_command = [
        "pynguin",
        "--project-path", PROJECT_ROOT,
        "--module-name", module_name,
        "--output-path", OUTPUT_FOLDER
    ]

    print(f"Executing: {' '.join(pynguin_command)}")

    result_code = execute_command_with_timeout(pynguin_command, log_filepath)

    if result_code == 0:
        print(f"Successfully generated test cases for {module_name}")
    else:
        print(f"Test generation failed or timed out for {module_name}. See log at {log_filepath}")
```

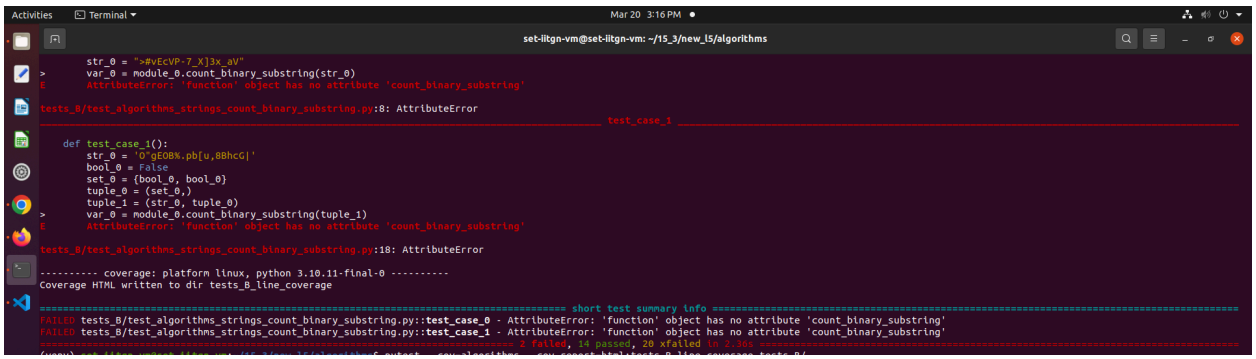
As many files were taking lot of time to run and still giving errors at the end, I kept timeout of 30 seconds for every file to generate test case using *pynguin*.

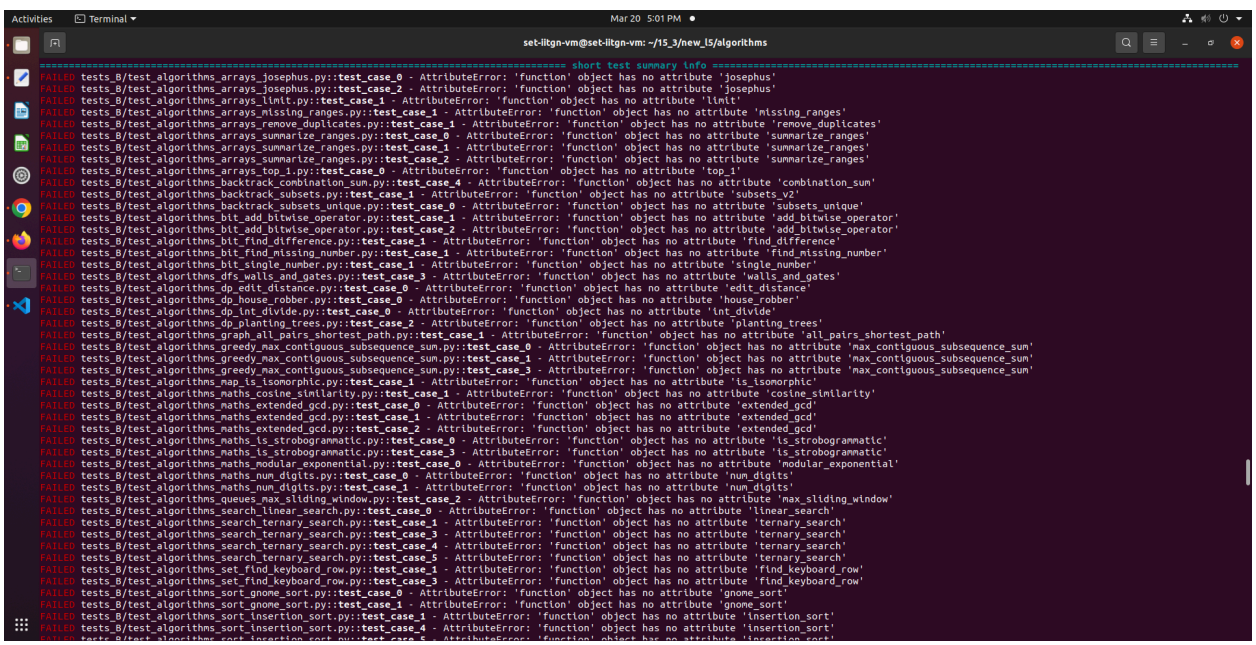
```
Results saved to 'low_coverage.txt'
(venv) set-titgn-vm@set-titgn-vm:~/15_3/new_l5/algorithms$ python get_low_coverage_files.py
Results saved to 'low_coverage_0.txt'
(venv) set-titgn-vm@set-titgn-vm:~/15_3/new_l5/algorithms$ python generate_tests_B.py
Found 56 Python files with low coverage.
Executing: pynguin --project-path . --module-name algorithms.dp.longest_common_subsequence --output-path tests_B
Successfully generated test cases for algorithms.dp.longest_common_subsequence
Executing: pynguin --project-path . --module-name algorithms.graph.clone_graph --output-path tests_B
```

Around 57 test files were generated in total by pynguin:



Even many generated test cases were also resulting in errors like these:



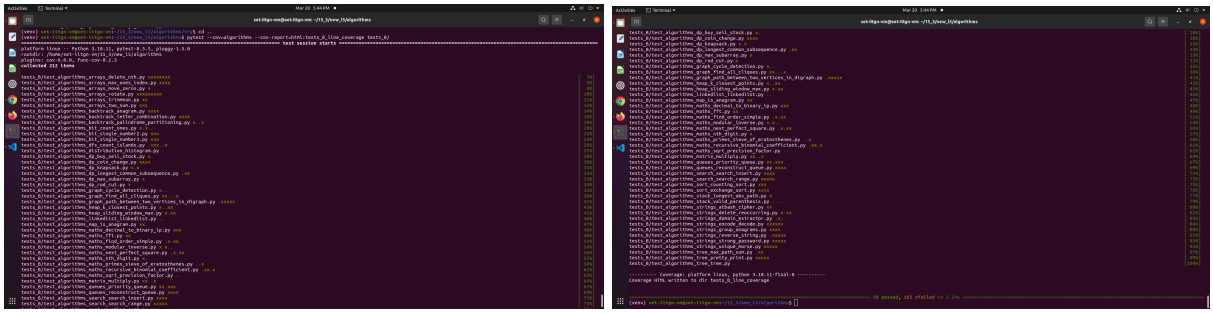


While generating coverage report and visualizing, I excluded those test cases.

Executing and Analyzing Coverage of Test Suite B

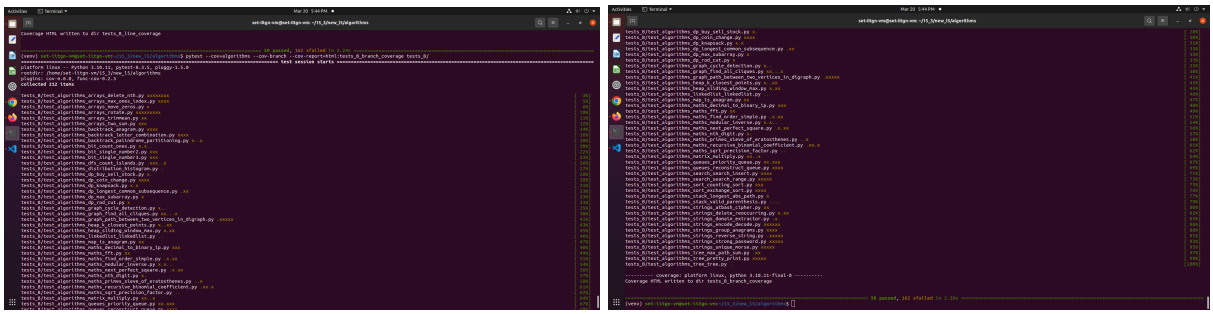
I executed and recorded the coverage metrics of Test Suite B using the following commands:

Line coverage for tests_B:



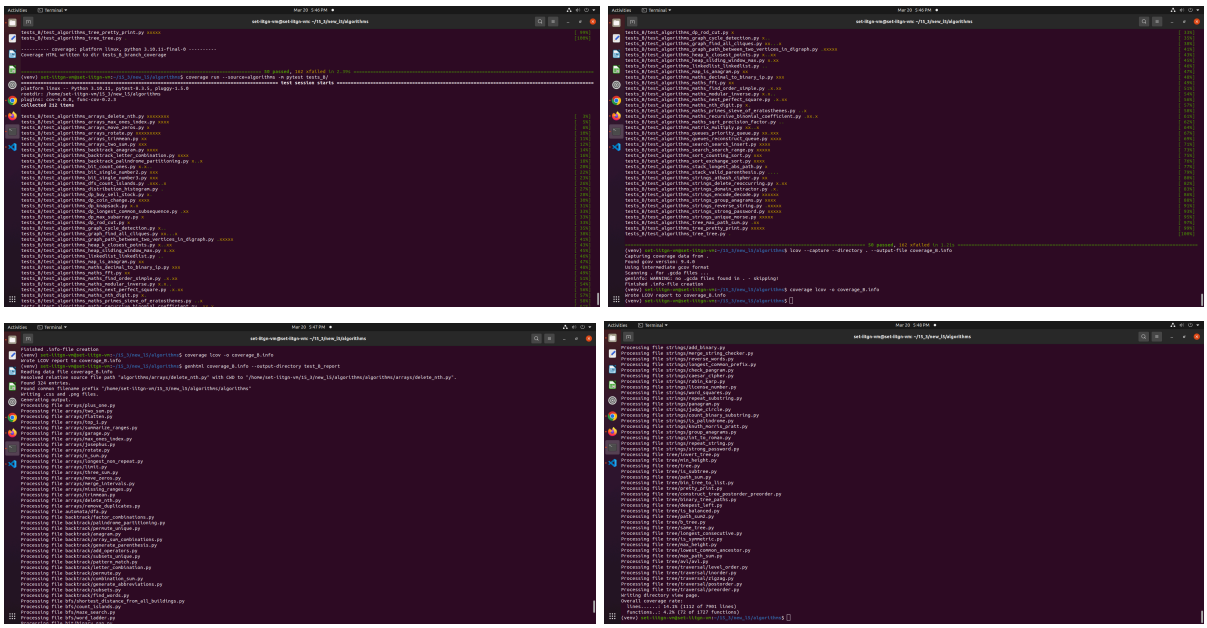
Line coverage report for Test Suite B can be found in tests_B_line_coverage folder.

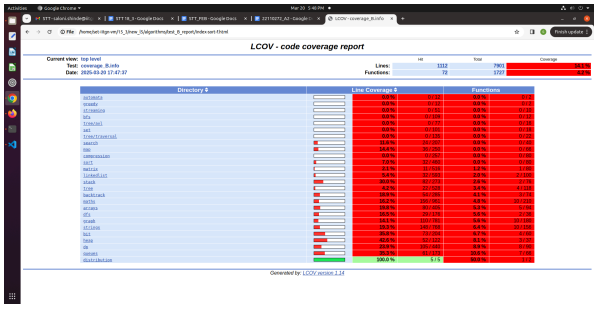
Branch coverage for tests_B:



Branch coverage report for Test Suite B can be found in tests_B_branch_coverage folder.

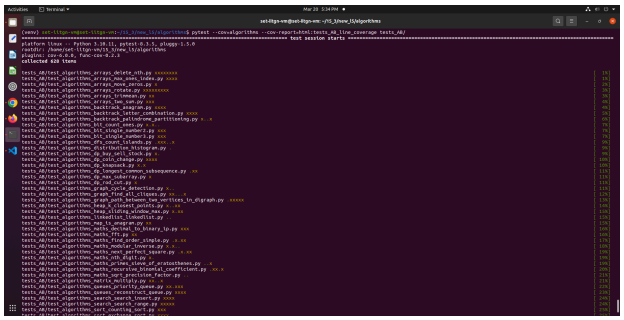
Visualization:

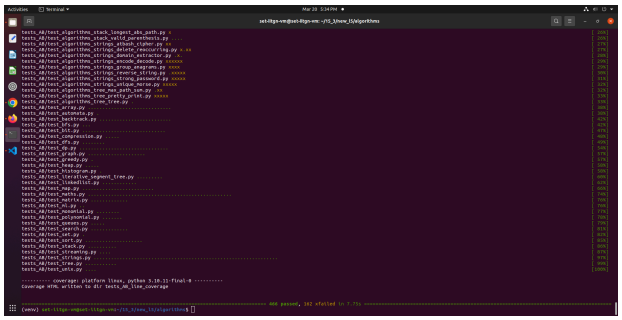




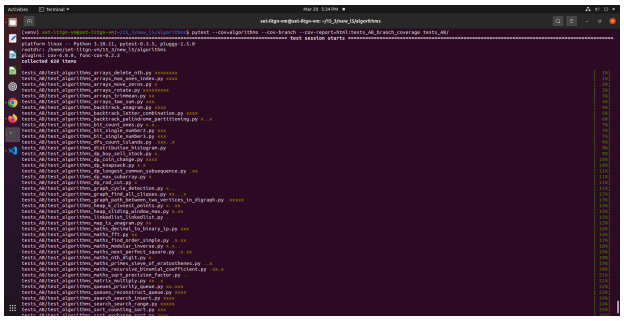
We combined Test Suite A and B and recorded their coverage report and visualizations:

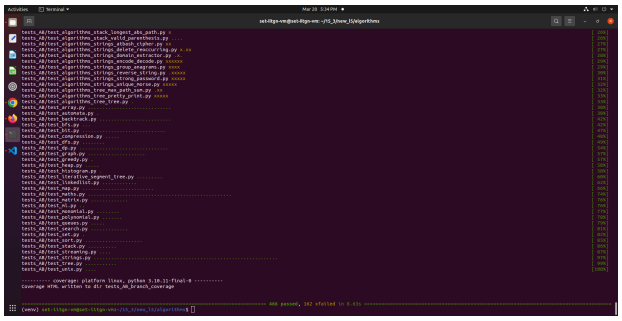
Line coverage for tests_AB:



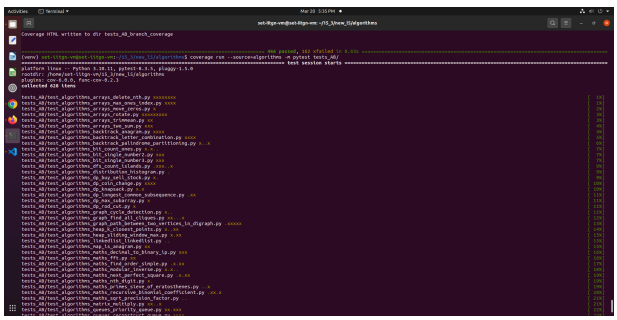


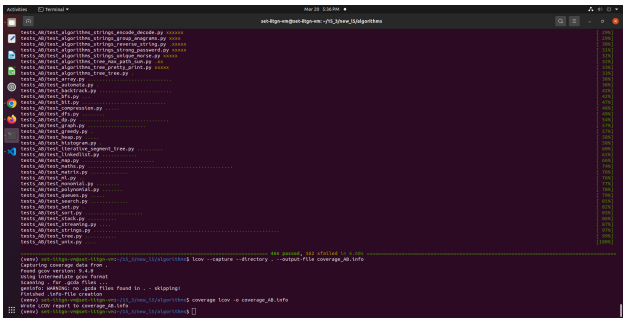
Branch coverage for tests_AB:

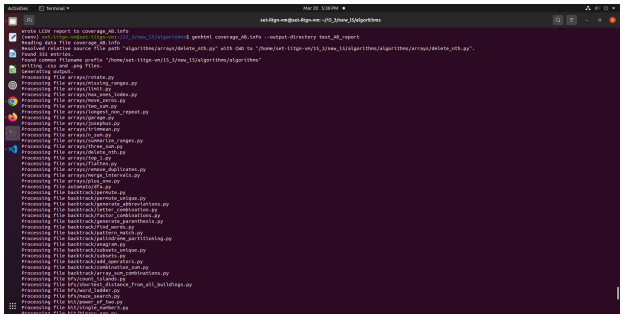


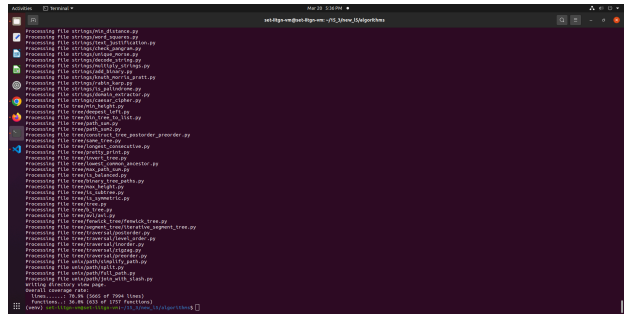


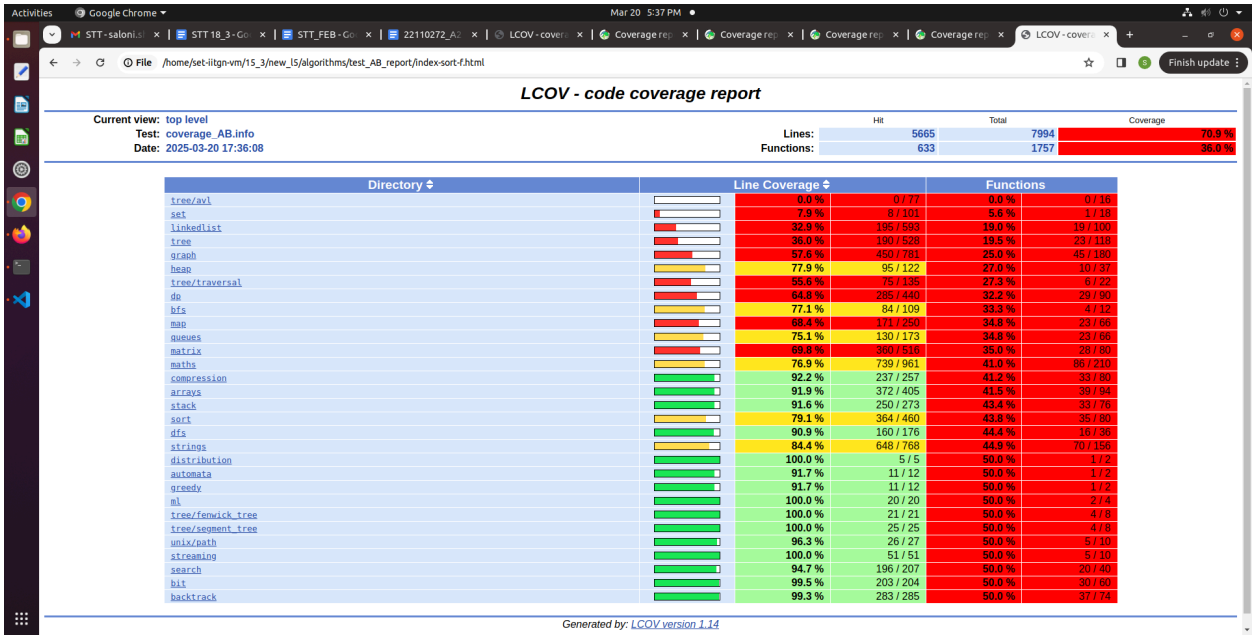
I did run following commands to get visualization of coverage report of tests_AB:











Coverage Analysis of test_suite_AB

- Overall Line Coverage: 71.6% (5665 / 7904)
- Overall Function Coverage: 36.0% (633 / 1757)

Worst Performing Directories (Poor Coverage)

- tree/avl: 0.0% (Not tested at all)
- set: Low coverage (exact numbers unclear but consistent with test_suite_A)
- linkedlist: Below 35% line and function coverage
- graph: Moderate coverage, but function coverage is still low

Well-Tested Directories

- ml: 100.0% line and 50.0% function coverage
- tree/fenwick_tree: 100.0% line and 50.0% function coverage
- tree/segment_tree: 100.0% line and 50.0% function coverage
- search: 94.7% line, 50.0% function coverage
- unix/path: 96.3% line, 50.0% function coverage
- bit: 99.5% line, 50.0% function coverage

Comparison with test_suite_A

- test_suite_AB has a higher overall line coverage (71.6%) compared to test_suite_A (~69%).
- Function coverage has improved slightly but remains low (36.0%).
- Some directories remain poorly tested, such as tree/avl, set, and linkedlist.

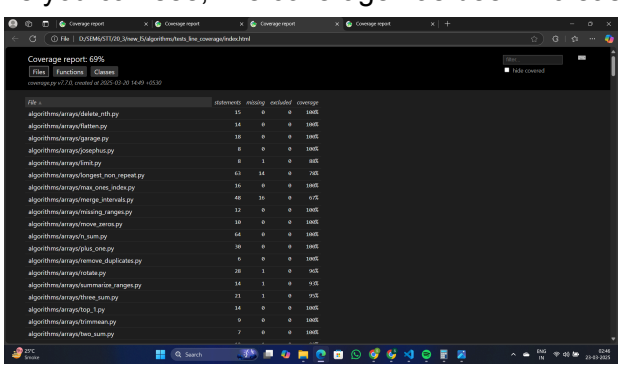
Conclusion

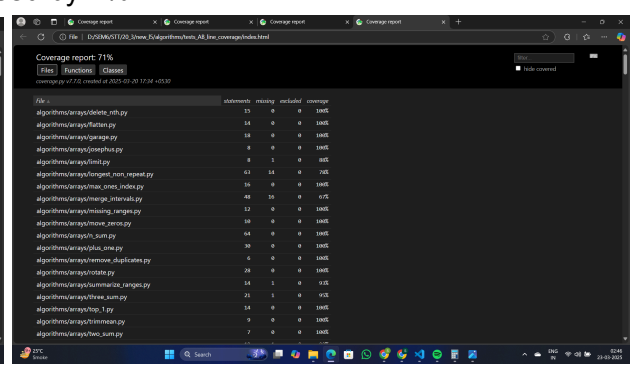
- Tree and graph structures still need better testing.
- Some components are well tested, but a significant portion of the codebase remains under-tested, especially in function coverage.
- Adding more test cases for untested functions and data structures (tree, graph, linked list) would improve overall test quality.

Comparing Test Suite A vs. Test Suite B

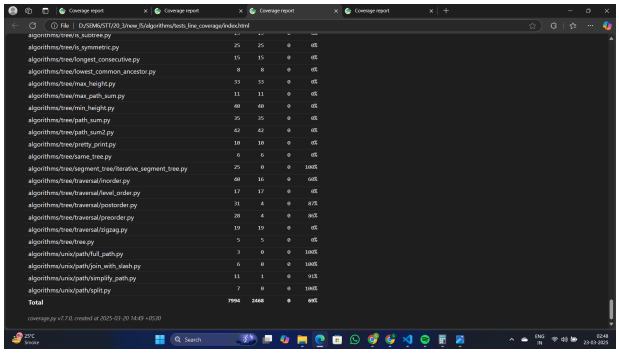
Metric	Test Suite A	Test Suite B	Test Suite AB
Line Coverage	69%	14%	71%
Branch Coverage	68%	12%	70%
LCOV Line Coverage	69.1%	14.1%	70.9%
LCOV Function Coverage	35%	4.2%	36%

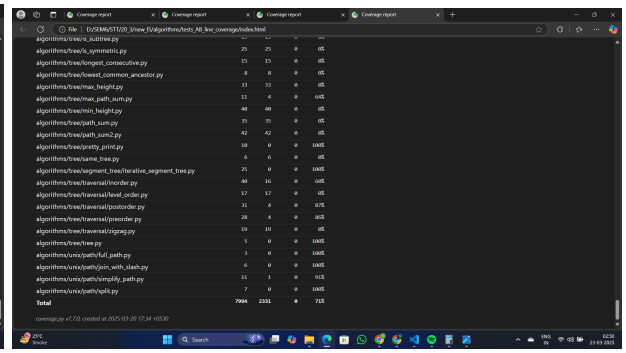
As you can see, line coverage has been increased by 2%:



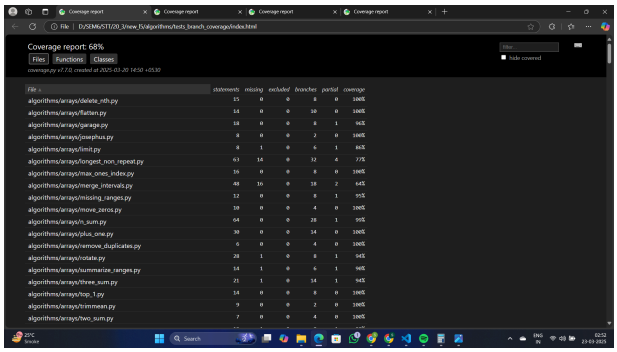


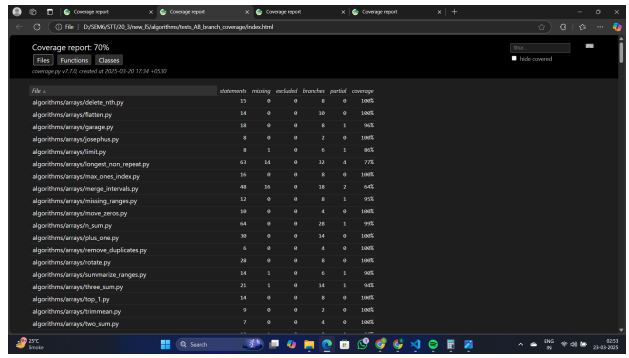
As you can see below, line coverage of few files has been increased from 0% to some value because of the new generated test cases:



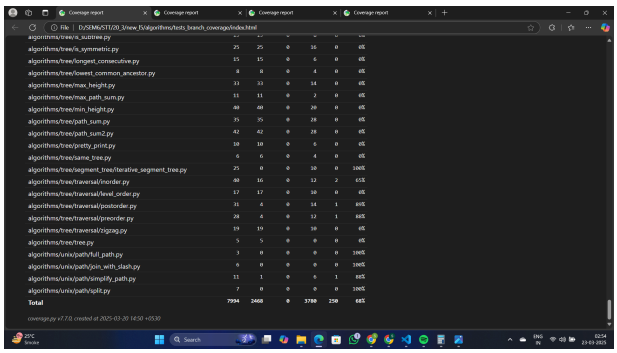


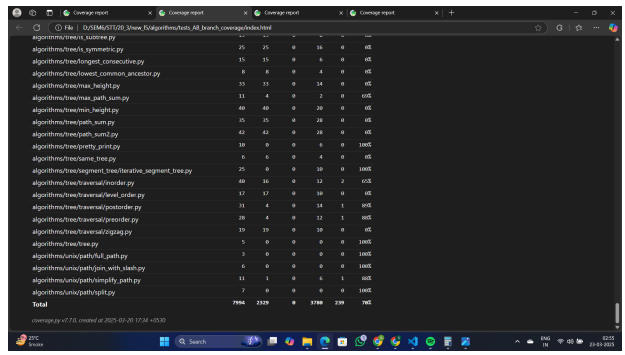
Branch coverage is also increased by 2%:





As you can see below, branch coverage of few files has been increased from 0% to some value because of the new generated test cases:





Discussion and Conclusion

Challenges Faced

According to test failures in **tests/test_array.py** and **algorithms/arrays/summarize_ranges.py** I corrected syntax issues and made other required changes to make it work properly.
During the lab, I faced challenges with test cases that caused program crashes, requiring manual filtering to ensure accurate coverage analysis. Additionally, generating test cases with

Pynguin was time-consuming, with some files taking too long to process. To address this, I implemented a 30-second timeout per file, which helped manage execution time while still obtaining useful test cases. These challenges highlighted the limitations of automated test generation tools and the need for careful tuning to balance efficiency and effectiveness in testing.

Lessons Learned

This experiment gave me hands-on experience with measuring and analyzing code coverage, emphasizing the challenges of automated test generation. By comparing manually written and generated test cases, I noticed small improvement in coverage, although some modules were still inadequately tested. The application of visualization tools facilitated detecting gaps, reaffirming the importance of a well-balanced testing strategy. In general, this exercise underscored the need for integrating automated and manual testing methodologies to gain more efficient software validation.

Lab Assignment 6

Lab topic: Python Test Parallelization

Introduction

Objective

This lab is intended to investigate the issue of parallelizing tests in Python. Students will analyze a few open-source Python projects to understand the efficacy and limitations of parallel running of tests. The objective is to explore various modes of parallelization, detect flaky tests, and understand the suitability of open-source projects for parallel running.

Learning Outcomes

By the end of this lab, students will be able to:
✓ Understand and apply different parallelization modes in pytest-xdist.
✓ Understand and apply different parallelization modes in pytest-run-parallel.
✓ Analyze test stability and flakiness in parallel execution environments.
✓ Evaluate the challenges and limitations of parallel test execution.
✓ Document and compare the parallel testing readiness of various open-source projects.

Tools and environmental setup

Operating System: Linux (SET-IITGN-VM)
Software Requirements:

- Python (*Python 3.8.10*)
- pytest (test execution)
- pytest-xdist (process level test parallelization)
- pytest-run-parallel (thread level test parallelization)

Methodology & Execution

Repository Setup & Setting up Virtual Environment and required dependencies

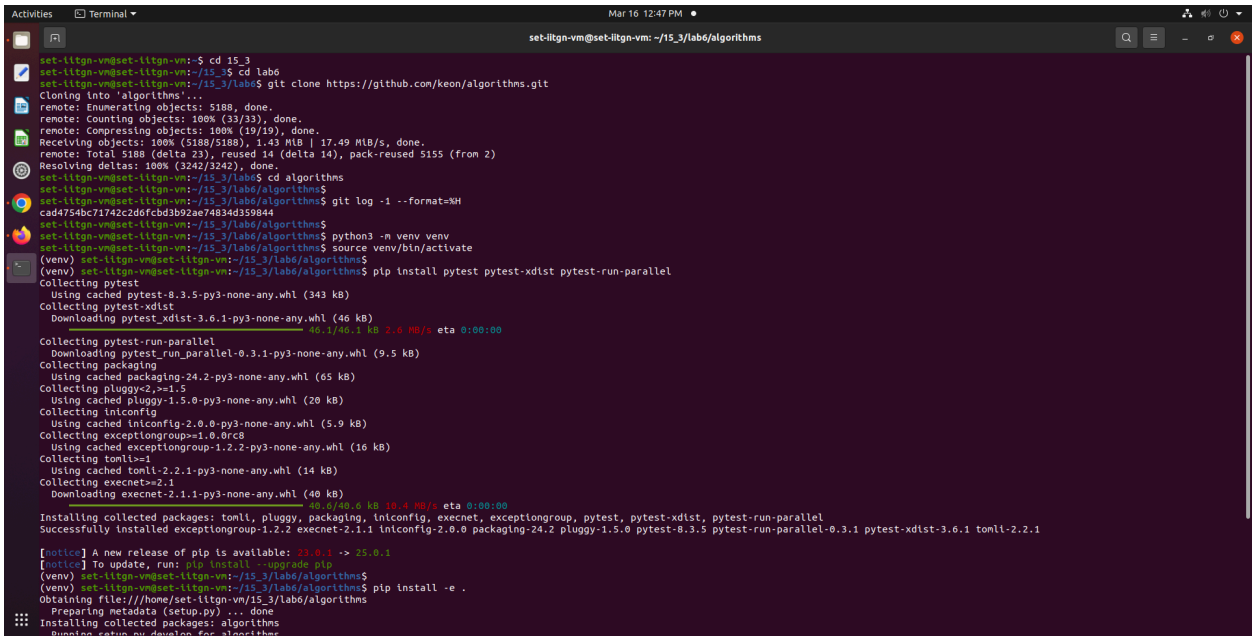
Firstly I have cloned the repository keon/algorithms using the following command:
git clone https://github.com/keon/algorithms.git

To get the current commit hash, I used the following command:
git log -1 --format=%H
Current commit hash: cad4754bc71742c2d6fcbd3b92ae74834d359844

To create and activate virtual environment I used this command:
python3.10 -m venv venv
source venv/bin/activate

To install requirements required to run the given repository, I ran these commands:
pip install -e .
python -c “import algorithms”

This is the command to install necessary dependencies:
pip install pytest pytest-xdist pytest-run-parallel

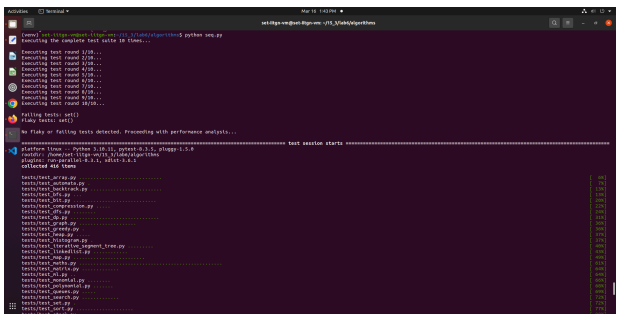


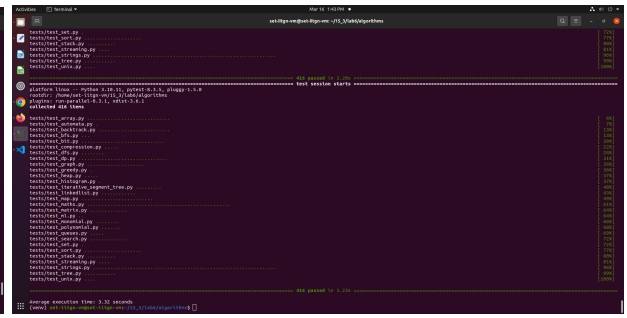
Sequential Test Execution

I used seq.py to run the test suite sequentially, it identifies failing and flaky tests, and measures execution time:

```python
import time
import subprocess
from collections import Counter

NUM_RUNS = 10
test_statistics = Counter()

def run_tests():
    """Run the full test suite multiple times."""
    print("Executing the complete test suite 10 times...\n")
    for i in range(NUM_RUNS):
        print(f"Executing test round {i + 1}/{NUM_RUNS}...")
        test_results = subprocess.run(["pytest", "tests", "--tb=short"], capture_output=True, text=True)

        analyze_test_results(test_results)

def analyze_test_results(results):
    """Analyze the output of pytest for failed tests."""
    for line in results.stdout.splitlines():
        if "FAILED" in line and "::" in line:
            test_name = line.split(" ")[1].strip()
            test_statistics[test_name] += 1

def get_test_status():
    """Get failing and flaky test names based on the statistics."""
    failing_tests = {test for test, count in test_statistics.items() if count == NUM_RUNS}
    flaky_tests = {test for test, count in test_statistics.items() if 0 < count < NUM_RUNS}
    return failing_tests, flaky_tests

def exclude_failing_tests(failing_tests):
    """Return arguments to exclude failing tests."""
    return [f"--deselect={test}" for test in failing_tests]
```

```python
def measure_execution_time(exclude_tests=None):
    """Measure the execution time of the test suite."""
    start = time.time()
    command = ["pytest", "tests"]
    if exclude_tests:
        command += exclude_tests
    subprocess.run(command, check=True)
    end = time.time()
    return end - start

def main():
    # Step 1: Run the tests multiple times and gather statistics
    run_tests()

    # Step 2: Analyze which tests are failing or flaky
    failing_tests, flaky_tests = get_test_status()

    print(f"\nFailing tests: {failing_tests}")
    print(f"Flaky tests: {flaky_tests}")

    # Step 3: Handle the results based on the failing/flaky tests
    if failing_tests or flaky_tests:
        print("\nExcluding failing and flaky tests to create a refined test suite...\n")

        exclude_tests = exclude_failing_tests(failing_tests)
        print(f"Excluding tests: {exclude_tests}")

        refined_times = [measure_execution_time(exclude_tests) for _ in range(5)]
        average_time = sum(refined_times) / len(refined_times)
        print(f"\nRefined average execution time (after removing flaky tests): {average_time:.2f} seconds")

    else:
        print("\nNo flaky or failing tests detected. Proceeding with performance analysis...\n")

        standard_times = [measure_execution_time() for _ in range(3)]
        average_time = sum(standard_times) / len(standard_times)
        print(f"\nAverage execution time: {average_time:.2f} seconds")

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





Average time of execution: 3.32 seconds

Parallel Test Execution

I used parallel.py to run the test suite in parallel with varying configurations and to measure performance:

```python
import time
import subprocess
import itertools
import json
import multiprocessing


def calculate_workers(worker_param):
    """Determine the number of workers based on pytest -n parameter."""
    if worker_param == "auto":
        return multiprocessing.cpu_count()  # Auto-select based on the number of CPU cores
    return int(worker_param)


def parse_failed_tests(stdout):
    """Extract failed test names from the pytest output."""
    failed_tests = []
    for line in stdout.splitlines():
        if "FAILED" in line and "::" in line:
            test_case = line.split("::")[1].split(" ")[0]
            failed_tests.append(test_case)
    return failed_tests


def calculate_average_duration(test_results):
    """Calculate the average duration of all test iterations."""
    total_duration = sum(res["iteration_duration"] for res in test_results)
    return total_duration / len(test_results)


def run_single_test_iteration(worker_param, thread_param, dist_mode, excluded_tests):
    """Run a single iteration of the test suite and return the results."""
    command = ["pytest", "tests", f"-n={worker_param}", f"--dist={dist_mode}", f"--parallel-threads={thread_param}"]
    start_time = time.time()
    result = subprocess.run(command + excluded_tests, capture_output=True, text=True)
    end_time = time.time()

    execution_time = end_time - start_time
    failed_tests = parse_failed_tests(result.stdout)

    return {
        "iteration_duration": execution_time,
        "failed_tests": failed_tests
    }
```

```python
def execute_test_suite(worker_param, thread_param, dist_mode, num_iterations=5):
    """Run the test suite multiple times and gather the results."""
    excluded_tests = []
    workers = calculate_workers(worker_param)  # Get the actual number of workers
    test_results = []
    failure_data = {}  # Track failed test names and how many times they fail

    for iteration in range(num_iterations):
        result = run_single_test_iteration(worker_param, thread_param, dist_mode, excluded_tests)

        # Track failed test occurrences
        for failed_test in result["failed_tests"]:
            if failed_test not in failure_data:
                failure_data[failed_test] = 0
            failure_data[failed_test] += 1

        test_results.append(result)

    avg_duration = calculate_average_duration(test_results)
    all_failed_tests = list(failure_data.keys())

    return {
        "requested_workers": worker_param,
        "actual_workers_used": workers,
        "configured_threads": thread_param,
        "distribution_mode": dist_mode,
        "average_duration": avg_duration,
        "failed_tests": all_failed_tests,
        "iteration_results": test_results,
        "failure_count": dict(failure_data)
    }


def save_results_to_file(results, filename="test_execution_results.json"):
    """Save the collected test results to a JSON file."""
    with open(filename, "w") as file:
        json.dump(results, file, indent=4)


if __name__ == "__main__":
    worker_options = ["1", "auto"]
    thread_options = ["1", "auto"]
    dist_options = ["load", "no"]

    all_test_results = []

    # Run tests for all combinations of worker, thread, and distribution configurations
    for worker, thread, dist in itertools.product(worker_options, thread_options, dist_options):
        print(f"Running tests with -n={worker}, --parallel-threads={thread}, --dist={dist}")
        result = execute_test_suite(worker, thread, dist)
        all_test_results.append(result)

    # Save all results to a JSON file
    save_results_to_file(all_test_results, filename="test_execution_results.json")

    print(f"Test execution completed. Results saved to test_execution_results.json")
```

```
(venv) set-iitgn-vm@set-iitgn-vm:~/15_3/lab6/algorithms$ python parallel.py
Running tests with -n=1, --parallel-threads=1, --dist=load
Running tests with -n=1, --parallel-threads=1, --dist=no
Running tests with -n=1, --parallel-threads=auto, --dist=load
Running tests with -n=1, --parallel-threads=auto, --dist=no
Running tests with -n=auto, --parallel-threads=1, --dist=load
Running tests with -n=auto, --parallel-threads=1, --dist=no
Running tests with -n=auto, --parallel-threads=auto, --dist=load
Running tests with -n=auto, --parallel-threads=auto, --dist=no
Test execution completed. Results saved to test_execution_results.json
(venv) set-iitgn-vm@set-iitgn-vm:~/15_3/lab6/algorithms$
```

Results can be found in test_execution_results.json present in submitted zip folder.

Results and Analysis

Sequential Execution Analysis

The total execution time for the sequential test run was 3.32 seconds.
No failing and flaky tests were identified.
So, the average execution time remained stable since start.

Parallel Execution Analysis

(a) Identify New Flaky Tests Failed in Parallel Test Executions

The test failures are observed in scenarios where configured_threads is set to auto.
When actual_workers_used = 4 (parallel execution), the following tests failed:

- TestBinaryHeap
- TestSuite
- TestHuffmanCoding

These tests did not fail in single-threaded runs (configured_threads = 1), indicating they are flaky in parallel execution.

This highlights that the test suite is not fully prepared for parallel execution.

(b) Inspect and Document the Causes of Test Failures in Parallel Runs:

- TestBinaryHeap
  Unexpected minimum element in test_remove_min: Likely due to race conditions when multiple threads try to remove elements from the heap simultaneously.
- TestSuite
  Expected True, got False in test_is_palindrome: May be caused by a shared linked list between test cases, leading to concurrency issues where one test modifies the list while another is checking for palindrome properties.
- TestHuffmanCoding
  Mismatch in encoded output in test_huffman_coding: The encoding dictionary generation might not be deterministic due to parallel execution, leading to inconsistent output across test runs.

(c) Calculate and compare speedup ratios for different parallelization modes and worker counts.

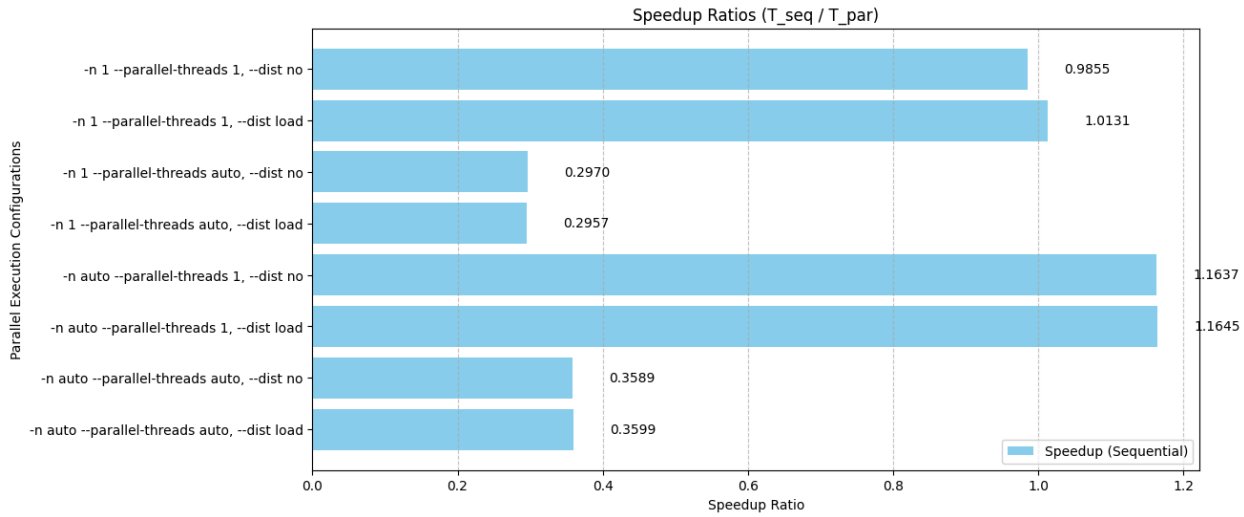
Configuration	Worker Count	Execution Time (seconds)	Speedup (T_seq/ T_par)	Speedup (T_par_max/ T_par)
-n auto --parallel-threads auto, --dist load	16	9.224	0.36	1.22
-n auto --parallel-threads auto, --dist no	16	9.251	0.36	1.21
-n auto --parallel-threads 1, --dist load	16	2.851	1.16	3.94
-n auto --parallel-threads 1, --dist no	16	2.853	1.16	3.94
-n 1 --parallel-threads auto, --dist load	16	11.227*	0.30	1.00
-n 1 --parallel-threads auto, --dist no	16	11.179	0.30	1.00
-n 1 --parallel-threads 1, --dist load	1	3.277	1.01	3.43
-n 1 --parallel-threads 1, --dist no	1	3.369	0.99	3.33

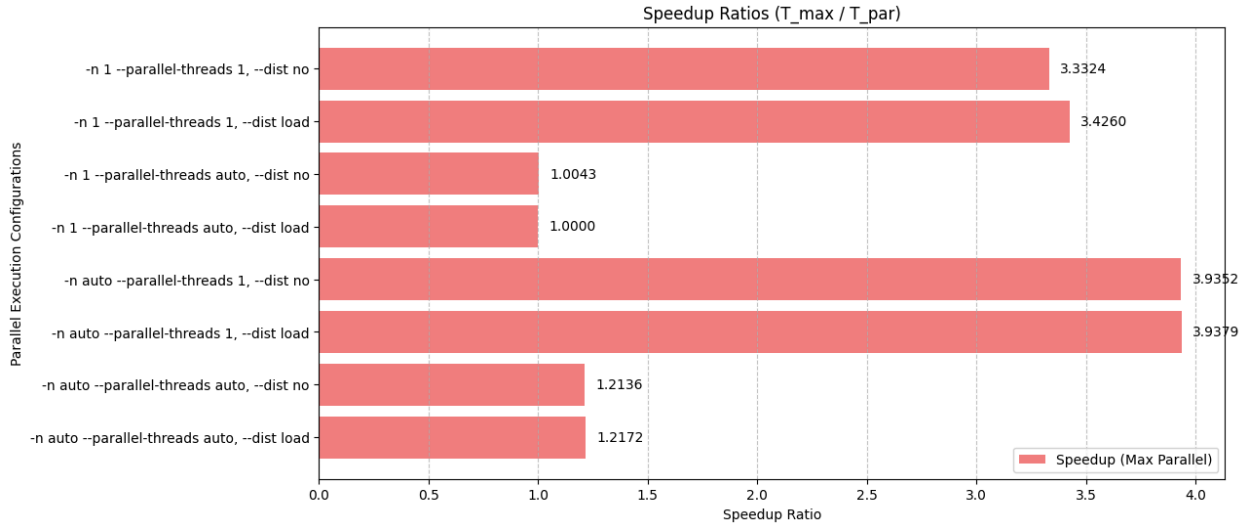
*T_par_max = maximum parallel execution time

Observations

- Configurations with --parallel-threads=1 exhibit the highest speedup when compared to the maximum execution time (T_par_max), achieving up to 3.94× improvement.
- Using --parallel-threads auto results in significantly higher execution times, leading to lower speedup values.
- Load distribution (--dist load) vs. no distribution (--dist no) has minimal impact on performance in most cases.
- The worst-performing configurations are those with -n 1 --parallel-threads auto, where execution time is close to T_par_max, leading to minimal speedup.

Graphical representation of speedup trends:





Discussion and Conclusion

<u>Challenges Observed</u>

The issues encountered are race conditions and contention for common resources, which can result in unpredictable behavior. Inconsistencies in timing can result in flaky test outcomes, lowering reliability. Also, performance degradation in multithreaded environments occurs because of synchronization overhead and resource contention.

Suggestions for pytest Developers

- Enhance thread safety mechanisms in pytest-run-parallel.
- Provide built-in tools for detecting and handling flaky tests.
- Improve resource isolation techniques to prevent parallel execution conflicts

Lessons Learned

Through this assignment, I learned significant information about running tests in parallel and the difficulties that come with it. I gained knowledge of how to leverage pytest-xdist and pytest-run-parallel for spreading tests between different threads and processes, speeding up execution. I also became aware of how flaky tests affect parallel execution and how race conditions, shared state problems, and non-deterministic execution can lead to failures. Moreover, I experimented with various parallelization modes and compared speedup ratios, further emphasizing the necessity for optimizing worker distribution.

Lab Assignment 7-8

Lab topic: Vulnerability Analysis on Open-Source Software Repositories

Introduction

Objective

This lab will expose students to Bandit, a code static analysis tool in an effort to identify security weaknesses in Python code. It will emphasize the way to install, configure, and run Bandit on Python projects on GitHub. Students will have hands-on skills in refining research communication skills.

Learning Outcomes

By the end of this lab, students will be able to:
✓ Understand the purpose and functionality of the bandit tool and setup bandit in local environment.
✓ Run bandit on python projects and interpret results.
✓ Perform a study on vulnerabilities in the open-source ecosystem.
✓ Learn to prepare a publication quality research report with in-depth analysis.

Tools and environmental setup

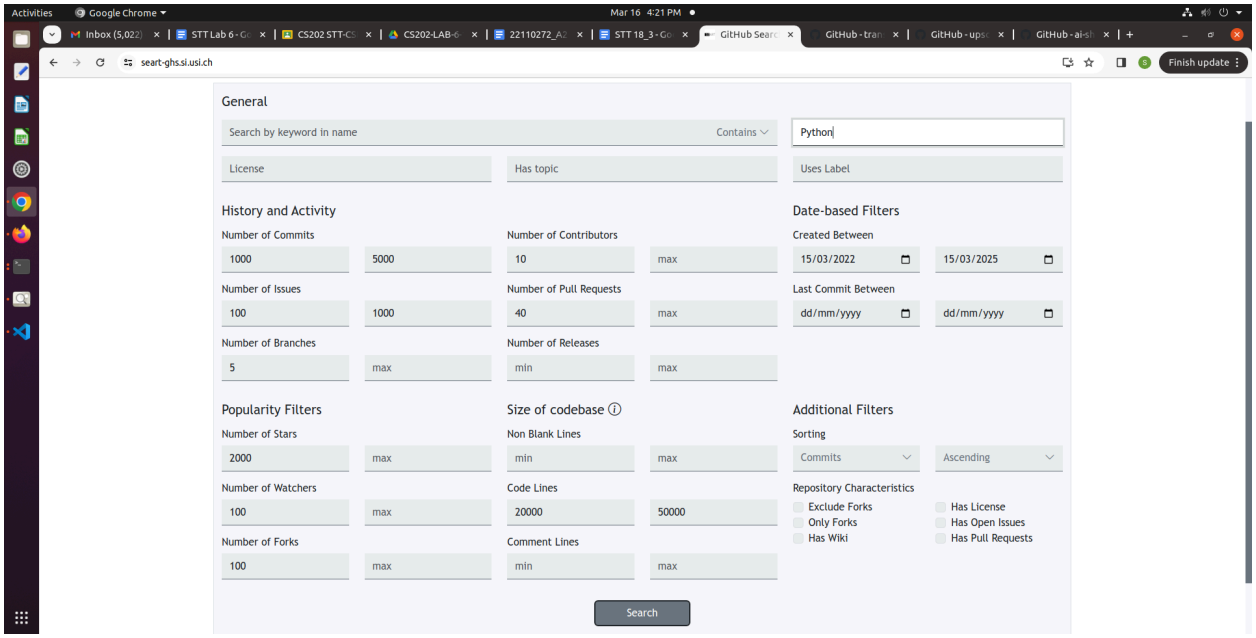
Operating System: Linux (SET-IITGN-VM)
Software Requirements:

- Python (*Python 3.8.10*)
- bandit (vulnerability scanning tool) 1.7.6

Methodology, Execution, Results and Analysis

Repository Selection Criteria

I chose top 3 repositories which came after searching on Github SEART Engine with following specifications:



MemGPT was redirected to letta.

<u>Selected Repositories:</u>

Repository 1: https://github.com/Lightning-AI/litgpt.git

Repository 2: https://github.com/mem0ai/mem0.git
Repository 3: https://github.com/letta-ai/letta.git

Virtual environment setup

Install Bandit: pip install bandit

To do virtual environmental setup for above selected repositories I followed following setup:

For litgpt and mem0, I used following commands:

```
cd litpgt
virtualenv venv
source venv/bin/activate
pip install -r requirements.txt || echo "No requirements file found"
deactivate
cd ..
```

For letta I had to install these dependencies:

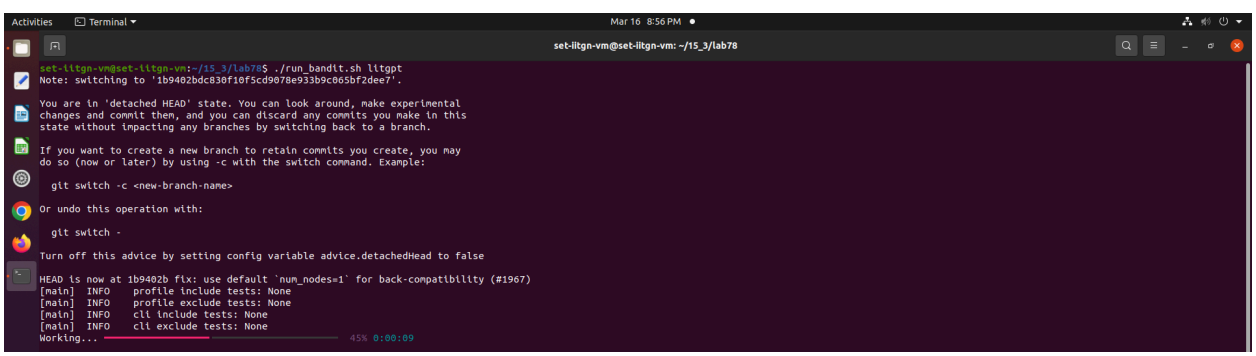
```
pip install poetry
poetry install
```

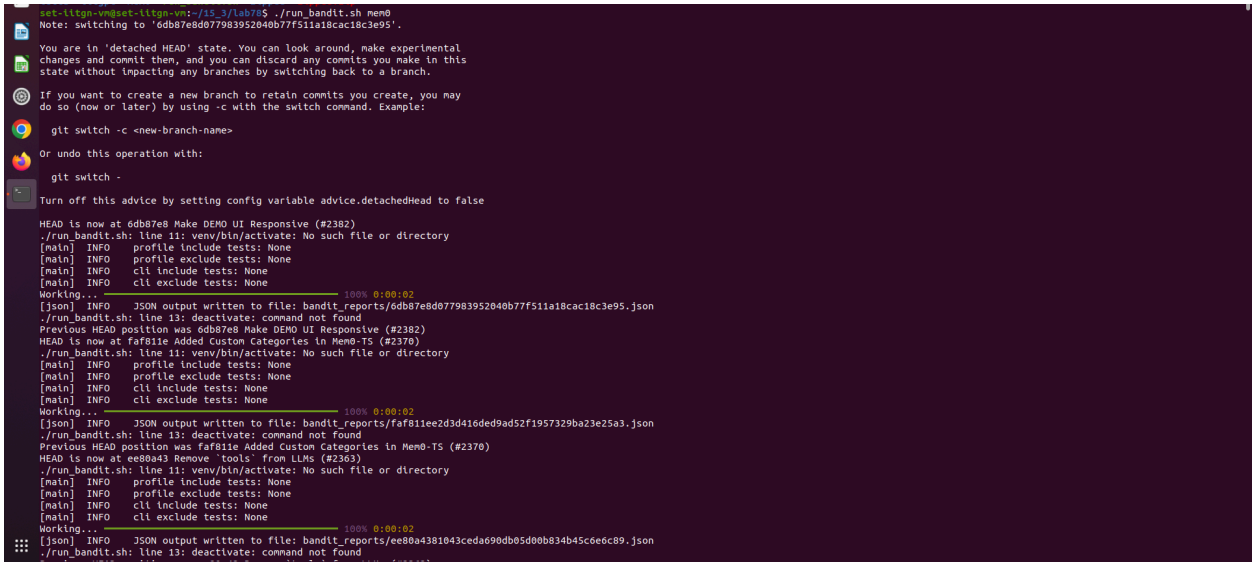
Bandit Execution on Selected Repositories

I used the following code saved in run_bandit.sh to execute bandit on the last 100 non-merge selected repositories:

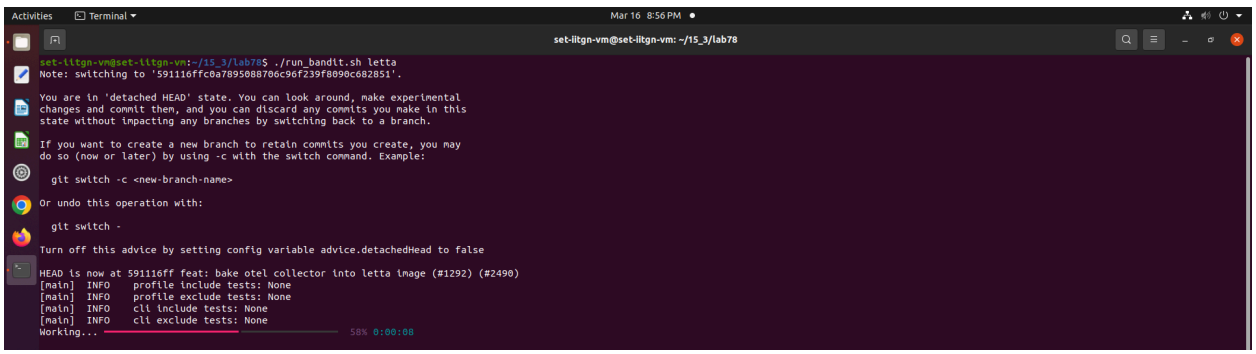
```
run_bandit.sh
run_bandit.sh
#!/bin/bash

REPO=$1
cd $REPO
git log --no-merges -n 100 --pretty=format:"%H" > commits.txt

mkdir -p bandit_reports

while read -r COMMIT; do
    git checkout $COMMIT
    source venv/bin/activate
    bandit -r . -f json -o "bandit_reports/${COMMIT}.json"
    deactivate
done < commits.txt

git checkout main  # Switch back to main branch
cd ..
```

To make the script executable and run:
chmod +x run_bandit.sh
./run_bandit.sh repo_name





```
set-iltgn-vm@set-iltgn-vm:~/15_3/lab78$ ./run_bandit.sh mem0
Note: switching to '6db87e8d07798395204 0b77f511a18cac18c3e95'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by switching back to a branch.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -c with the switch command. Example:

  git switch -c <new-branch-name>

Or undo this operation with:

  git switch -

Turn off this advice by setting config variable advice.detachedHead to false

HEAD is now at 6db87e8 Make DEMO UI Responsive (#2382)
./run_bandit.sh: line 11: venv/bin/activate: No such file or directory
[main]  INFO    profile include tests: None
[main]  INFO    profile exclude tests: None
[main]  INFO    cli include tests: None
[main]  INFO    cli exclude tests: None
Working... ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 100% 0:00:02
[json]  INFO    JSON output written to file: bandit_reports/6db87e8d07798395204 0b77f511a18cac18c3e95.json
./run_bandit.sh: line 13: deactivate: command not found
Previous HEAD position was 6db87e8 Make DEMO UI Responsive (#2382)
HEAD is now at faf811e Added Custom Categories in Mem0-TS (#2370)
./run_bandit.sh: line 11: venv/bin/activate: No such file or directory
[main]  INFO    profile include tests: None
[main]  INFO    profile exclude tests: None
[main]  INFO    cli include tests: None
[main]  INFO    cli exclude tests: None
Working... ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 100% 0:00:02
[]son]  INFO    JSON output written to file: bandit_reports/faf811ee2d3d416ded9ad52f1957329ba23e25a3.json
./run_bandit.sh: line 13: deactivate: command not found
Previous HEAD position was faf811e Added Custom Categories in Mem0-TS (#2370)
HEAD is now at ee80a43 Remove `tools` from LLMs (#2363)
./run_bandit.sh: line 11: venv/bin/activate: No such file or directory
[main]  INFO    profile include tests: None
[main]  INFO    profile exclude tests: None
[main]  INFO    cli include tests: None
[main]  INFO    cli exclude tests: None
Working... ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 100% 0:00:02
[json]  INFO    JSON output written to file: bandit_reports/ee80a4381043ceda690db05d00b834b45c6e6c89.json
./run_bandit.sh: line 13: deactivate: command not found
```



```
Activities   Terminal   Mar 16  8:56 PM
set-iltgn-vm@set-iltgn-vm: ~/15_3/lab78
set-iltgn-vm@set-iltgn-vm:~/15_3/lab78$ ./run_bandit.sh letta
Note: switching to '591116ffc0a7895088706c96f239f8090c682851'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by switching back to a branch.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -c with the switch command. Example:

  git switch -c <new-branch-name>

Or undo this operation with:

  git switch -

Turn off this advice by setting config variable advice.detachedHead to false

HEAD is now at 591116ff feat: bake otel collector into letta image (#1292) (#2490)
[main]  INFO    profile include tests: None
[main]  INFO    profile exclude tests: None
[main]  INFO    cli include tests: None
[main]  INFO    cli exclude tests: None
Working... ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 58% 0:00:08
```

Individual Repository-Level Analysis

To assess security vulnerabilities across all selected repositories I used Bandit to scan Python files across last 100 non-merge commits. The analysis focuses on three key aspects:

- Confidence Levels – Categorizing issues based on HIGH, MEDIUM, and LOW confidence
- Severity Levels – Classifying issues into HIGH, MEDIUM, and LOW severity
- Unique CWEs Identified – Extracting Common Weakness Enumeration (CWE) identifiers per commit

To automate this process I used indi_analysis.py:

```python
import os
import json
import pandas as pd

# Constants
REPOS = ['litgpt', 'mem0', 'letta']
OUTPUT_FOLDER = 'individual_analysis'
os.makedirs(OUTPUT_FOLDER, exist_ok=True)

# Utility function to parse each Bandit report file
def parse_report_file(report_file):
    """
    Parse a single Bandit report file and extract the relevant data.
    """
    with open(report_file, 'r') as f:
        report = json.load(f)

    high_conf, med_conf, low_conf = 0, 0, 0
    high_sev, med_sev, low_sev = 0, 0, 0
    unique_cwes = set()

    for result in report.get('results', []):
        # Confidence counting
        confidence = result['issue_confidence']
        if confidence == 'HIGH':
            high_conf += 1
        elif confidence == 'MEDIUM':
            med_conf += 1
        elif confidence == 'LOW':
            low_conf += 1

        # Severity counting
        severity = result['issue_severity']
        if severity == 'HIGH':
            high_sev += 1
        elif severity == 'MEDIUM':
            med_sev += 1
        elif severity == 'LOW':
            low_sev += 1

        # CWE ID extraction
        cwe_id = result.get('issue_cwe', {}).get('id')
        if cwe_id:
            unique_cwes.add(cwe_id)

    return high_conf, med_conf, low_conf, high_sev, med_sev, low_sev, list(unique_cwes)
```

```python
# Function to process all reports within a repository
def process_repo_reports(repo):
    """
    Process all Bandit reports for a given repository.
    """
    reports_path = os.path.join(repo, 'bandit_reports')
    data = []

    # Iterate through all report files in the repository
    for report_file in os.listdir(reports_path):
        commit = report_file.replace('.json', '')
        report_path = os.path.join(reports_path, report_file)

        # Parse the current report file
        high_conf, med_conf, low_conf, high_sev, med_sev, low_sev, unique_cwes = parse_report_file(report_path)

        # Store the parsed data for the current report
        data.append({
            'repo': repo,
            'commit': commit,
            'high_conf': high_conf,
            'med_conf': med_conf,
            'low_conf': low_conf,
            'high_sev': high_sev,
            'med_sev': med_sev,
            'low_sev': low_sev,
            'unique_cwes': unique_cwes,
            'total_no_unique_cwes': len(unique_cwes)
        })

    return pd.DataFrame(data)

# Function to analyze a single repository
def analyze_repo(repo):
    """
    Analyze a single repository and generate its summary CSV.
    """
    print(f"Analyzing {repo}...")
    df = process_repo_reports(repo)
    df = df.sort_values('commit')
    output_path = os.path.join(OUTPUT_FOLDER, f'{repo}_bandit_summary.csv')
    df.to_csv(output_path, index=False)
    print(f"{repo} analysis complete. Summary saved to {output_path}")

# Main execution to analyze all repositories
def analyze_all_repos():
    """
    Analyze all repositories in the list and save their individual summaries.
    """
    for repo in REPOS:
        analyze_repo(repo)
    print("Analysis complete. CSV summaries saved in 'individual_analysis' folder.")

# Run the analysis for all repositories
if __name__ == "__main__":
    analyze_all_repos()
```

Results are stored in ‘individual_analysis’ folder.

The stored csv’s:

litgpt:

repo	commit	high_conf	med_conf	low_conf	high_sev	med_sev	low_sev	unique_cwes	total_no_unique_cwes
litgpt	0031d554be2edb20c775ba61a05f60ac9b044793	2973	14	7	22	37	2935	[259, 327, 330, 78, 400, 20, 502, 22, 377, 732, 703]	11
litgpt	00af8b0d9b43eaa3f8c812e3e8bd53ef38cbb49d	2943	13	7	22	37	2904	[259, 327, 330, 78, 400, 20, 502, 22, 377, 732, 703]	11
litgpt	0fb584787eb51f69347b82c1c9488cc175bd868d	2943	13	7	22	37	2904	[259, 327, 330, 78, 400, 20, 502, 22, 377, 732, 703]	11
litgpt	11ac040f076bd8a2880364fec11093b54d99b71b	2943	13	7	22	37	2904	[259, 327, 330, 78, 400, 20, 502, 22, 377, 732, 703]	11
litgpt	151d0be30b538ae08eb6737e8850118ec3955926	2964	13	7	22	37	2925	[259, 327, 330, 78, 400, 20, 502, 22, 377, 732, 703]	11
litgpt	17a58dfa65c6c5c21455acf56d0f9c7533dc0f49	2973	14	7	22	37	2935	[259, 327, 330, 78, 400, 20, 502, 22, 377, 732, 703]	11
litgpt	1809a39299c837dfa823b6f191d441106a901940	2951	13	7	22	37	2912	[259, 327, 330, 78, 400, 20, 502, 22, 377, 732, 703]	11
litgpt	1811ecc5e7b872b4fe277b74bc899f558b01ca4c	2964	14	7	22	37	2926	[259, 327, 330, 78, 400, 20, 502, 22, 377, 732, 703]	11
litgpt	1b568fc86e324c538e06c7e9ee29d835d4339a15	2964	13	7	22	37	2925	[259, 327, 330, 78, 400, 20, 502, 22, 377, 732, 703]	11
litgpt	1b9402bdc830f10f5cd9078e933b9c065bf2dee7	2977	16	7	22	37	2941	[259, 327, 330, 78, 400, 20, 502, 22, 377, 732, 703]	11
litgpt	1d93671269de04fc96a63a795e137de5b13dc99b	2973	14	7	22	37	2935	[259, 327, 330, 78, 400, 20, 502, 22, 377, 732, 703]	11
litgpt	1f00c068abc75afd5debf88be590c444c810f813	2977	16	7	22	37	2941	[259, 327, 330, 78, 400, 20, 502, 22, 377, 732, 703]	11
litgpt	22528bf29c116fb70b210a0c574507d2c62f6619	2952	13	7	22	37	2913	[259, 327, 330, 78, 400, 20, 502, 22, 377, 732, 703]	11
litgpt	250599b8ad30273ab31bac17166f845e7cecb4ec	2973	14	7	22	37	2935	[259, 327, 330, 78, 400, 20, 502, 22, 377, 732, 703]	11
litgpt	29a33cbd25271fc07be86a51049e81aba126e88f	2964	13	7	22	37	2925	[259, 327, 330, 78, 400, 20, 502, 22, 377, 732, 703]	11
litgpt	33eab00da13684a75b02eb9297fa4065d3f85e1e	2947	13	7	22	37	2908	[259, 327, 330, 78, 400, 20, 502, 22, 377, 732, 703]	11
litgpt	34930ba9fe491516b16369be01da1a9ae9fac36f	2958	14	7	22	37	2920	[259, 327, 330, 78, 400, 20, 502, 22, 377, 732, 703]	11
litgpt	36bfe87d345c2ad0d3a21b46b212f2687fb59c2d	2964	13	7	22	37	2925	[259, 327, 330, 78, 400, 20, 502, 22, 377, 732, 703]	11
litgpt	3b7f5bb624e59b2eb7b57d5334a272a4506f03e2	2975	16	7	22	37	2939	[259, 327, 330, 78, 400, 20, 502, 22, 377, 732, 703]	11
litgpt	3ca2311797b526181904b83a5697a76ea65b5082	2964	13	7	22	37	2925	[259, 327, 330, 78, 400, 20, 502, 22, 377, 732, 703]	11
litgpt	3ea3d93593fbe00da48f0730444f0730e087a945	2953	13	7	22	37	2914	[259, 327, 330, 78, 400, 20, 502, 22, 377, 732, 703]	11
litgpt	3eb93c8d6eff67f9c26d11778bfaa08801b78526	2977	16	7	22	37	2941	[259, 327, 330, 78, 400, 20, 502, 22, 377, 732, 703]	11
litgpt	3f28cd67ba4fd7ad9502e86dec35ae6aef0b5283	2956	13	7	22	37	2917	[259, 327, 330, 78, 400, 20, 502, 22, 377, 732, 703]	11
litgpt	3f4992aa89cc999ecd031894cf2d1f82efdb6855	2951	13	7	22	37	2912	[259, 327, 330, 78, 400, 20, 502, 22, 377, 732, 703]	11
litgpt	40c08dcfffbf315ff9de0ae5fc26c8e1e0c0a9bc	2973	14	7	22	37	2935	[259, 327, 330, 78, 400, 20, 502, 22, 377, 732, 703]	11
litgpt	467fb871c32a4c30b9ee1f2314e4f71f3c89f133	2951	13	7	22	37	2912	[259, 327, 330, 78, 400, 20, 502, 22, 377, 732, 703]	11

mem0:

repo	commit	high_conf	med_conf	low_conf	high_sev	med_sev	low_sev	unique_cwes	total_no_unique_cwes
mem0	0234c85be5062a81455a1f69a8c34ee5bdd8a47f	710	25	27	5	46	711	[259, 327, 377, 330, 78, 400, 20, 89, 605, 94, 703]	11
mem0	07d2f1108102885e7dce5cb7a90488acbbdff5ac	710	25	27	5	46	711	[259, 327, 377, 330, 78, 400, 20, 89, 605, 94, 703]	11
mem0	17887b59597a29ed6623b2331d03876d72289f45	684	25	27	5	46	685	[259, 327, 377, 330, 78, 400, 20, 89, 605, 94, 703]	11
mem0	192c33f19029432bc096c4df2557e36476127f6b	710	25	27	5	46	711	[259, 327, 377, 330, 78, 400, 20, 89, 605, 94, 703]	11
mem0	1aef468ebe130e6e7aedf753552f644eb76739cd	710	25	27	5	46	711	[259, 327, 377, 330, 78, 400, 20, 89, 605, 94, 703]	11
mem0	1be0d70d02a61ae17d03898f9ad2cf7b2f721926	684	25	27	5	46	685	[259, 327, 377, 330, 78, 400, 20, 89, 605, 94, 703]	11
mem0	20c03eaa92803cbb84a1a9bd9ec504903c97584d	701	25	27	5	46	702	[259, 327, 377, 330, 78, 400, 20, 89, 605, 94, 703]	11
mem0	23dbce4f59585c0ed78689948481408ab7aa181d	710	25	27	5	46	711	[259, 327, 377, 330, 78, 400, 20, 89, 605, 94, 703]	11
mem0	2556c5fe882ffb342c4631c58472af340438152c	686	25	27	5	46	687	[259, 327, 377, 330, 78, 400, 20, 89, 605, 94, 703]	11
mem0	2611343cbe435a384328cb622965dd0a3f5da295	710	25	27	5	46	711	[259, 327, 377, 330, 78, 400, 20, 89, 605, 94, 703]	11
mem0	29d63306a4b75f6c7115fbbbe5f62ed7e360d5ef	684	25	27	5	46	685	[259, 327, 377, 330, 78, 400, 20, 89, 605, 94, 703]	11
mem0	2c31a930a375aa6f5984044b6f20aab0d1d3a379	710	25	27	5	46	711	[259, 327, 377, 330, 78, 400, 20, 89, 605, 94, 703]	11
mem0	2d0c8fe94e793209b995007e2e4f8aff650bf41e	684	25	27	5	46	685	[259, 327, 377, 330, 78, 400, 20, 89, 605, 94, 703]	11
mem0	2d5062bd40e789175d457e195a55eef069ab1bec	713	25	27	5	46	714	[259, 327, 377, 330, 78, 400, 20, 89, 605, 94, 703]	11
mem0	308e79bb68922262a66aa03e1eba4e46d2a57de5	684	25	27	5	46	685	[259, 327, 377, 330, 78, 400, 20, 89, 605, 94, 703]	11
mem0	329d0cc945fd718cc185fc422911464a40e45cf8	710	25	27	5	46	711	[259, 327, 377, 330, 78, 400, 20, 89, 605, 94, 703]	11
mem0	32ebdaef2f67840a72039bad16bbb4b99f007b39	686	25	27	5	46	687	[259, 327, 377, 330, 78, 400, 20, 89, 605, 94, 703]	11
mem0	33fcc53e4b7ce652351b8762339b686ccc502961	710	25	27	5	46	711	[259, 327, 377, 330, 78, 400, 20, 89, 605, 94, 703]	11
mem0	369d5325f94d39ca52477db089cae00b4eb97490	684	25	27	5	46	685	[259, 327, 377, 330, 78, 400, 20, 89, 605, 94, 703]	11
mem0	371848cfbc12c3b1e34cf7e174cd2bfa656c1036	684	25	27	5	46	685	[259, 327, 377, 330, 78, 400, 20, 89, 605, 94, 703]	11
mem0	41a42da774a12b3e92b7914c3f2f74af6e5c0ea5	710	25	27	5	46	711	[259, 327, 377, 330, 78, 400, 20, 89, 605, 94, 703]	11
mem0	4237b9220bfb717fc45e2bb5d1d0e88fe9e1cc88	710	25	27	5	46	711	[259, 327, 377, 330, 78, 400, 20, 89, 605, 94, 703]	11
mem0	43186636974947 9bcc100ba4491a040fd2986e07	686	25	27	5	46	687	[259, 327, 377, 330, 78, 400, 20, 89, 605, 94, 703]	11
mem0	434b555a2996ea632391479e8f1335e5908c07e9	684	25	27	5	46	685	[259, 327, 377, 330, 78, 400, 20, 89, 605, 94, 703]	11
mem0	48176bd19436556fb864bb51496eae19d7074c37	684	25	27	5	46	685	[259, 327, 377, 330, 78, 400, 20, 89, 605, 94, 703]	11
mem0	4be426f7624e3f486b88bd9106ec0d71edd33f5f	756	25	27	5	46	757	[259, 327, 377, 330, 78, 400, 20, 89, 605, 94, 703]	11

letta:

repo	commit	high_conf	med_conf	low_conf	high_sev	med_sev	low_sev	unique_cwes	total_no_unique_cwes
letta	03010d262a1882944fc3d4eab454692311874e7c	3598	35	113	25	158	3563	[259, 327, 377, 330, 78, 400, 20, 502, 22, 89, 732, 605, 94, 703]	14
letta	052c296ea17fefda1d1db434533aeba0010e15e3	3746	38	113	25	161	3711	[259, 327, 377, 330, 78, 400, 20, 502, 22, 89, 732, 605, 94, 703]	14
letta	0607f78f8c6a4caa0b7a1309fcbf7d894c13a124	3753	38	113	25	161	3718	[259, 327, 377, 330, 78, 400, 20, 502, 22, 89, 732, 605, 94, 703]	14
letta	0ef6de6c0cd2ff6cfec9076481b519922d37ab4a	3596	35	113	25	158	3561	[259, 327, 377, 330, 78, 400, 20, 502, 22, 89, 732, 605, 94, 703]	14
letta	0fbe22dd4368cc7e2b8d352e5b1dd366b8615d25	3694	37	113	25	160	3659	[259, 327, 377, 330, 78, 400, 20, 502, 22, 89, 732, 605, 94, 703]	14
letta	0feb23167007b31b664300a84845efd110bb7390	3689	37	113	25	160	3654	[259, 327, 377, 330, 78, 400, 20, 502, 22, 89, 732, 605, 94, 703]	14
letta	18e4062d050964854e5c36de535f1fc91aae5cc1	3599	35	113	25	158	3564	[259, 327, 377, 330, 78, 400, 20, 502, 22, 89, 732, 605, 94, 703]	14
letta	22c7c71c66cece835ff29c1a6480a52633f42cab	3706	35	113	25	158	3671	[259, 327, 377, 330, 78, 400, 20, 502, 22, 89, 732, 605, 94, 703]	14
letta	2392d551ca853e0e65b7c57c8f5c0fabf8864653	3656	37	112	25	159	3621	[259, 327, 377, 330, 78, 400, 20, 502, 22, 89, 732, 605, 94, 703]	14
letta	274382953a182320cb397914c084537603ddaad8	3682	35	113	25	158	3647	[259, 327, 377, 330, 78, 400, 20, 502, 22, 89, 732, 605, 94, 703]	14
letta	28122d64e97abd77e41a8395fc8ca67044656b17	3599	35	113	25	158	3564	[259, 327, 377, 330, 78, 400, 20, 502, 22, 89, 732, 605, 94, 703]	14
letta	2a3549fe11b023207719a5731675e91810b4ce6a	3656	37	112	25	159	3621	[259, 327, 377, 330, 78, 400, 20, 502, 22, 89, 732, 605, 94, 703]	14
letta	2b00b2b14b3978c0b351464629b25d2372b74708	3621	35	114	25	159	3586	[259, 327, 377, 330, 78, 400, 20, 502, 22, 89, 732, 605, 94, 703]	14
letta	2e3415bc61b044614b89b2b3d76fda6538a45bef	3599	35	113	25	158	3564	[259, 327, 377, 330, 78, 400, 20, 502, 22, 89, 732, 605, 94, 703]	14
letta	2f3b5a869d096f8ac719f7357693f62d301de1d5	3661	37	112	25	159	3626	[259, 327, 377, 330, 78, 400, 20, 502, 22, 89, 732, 605, 94, 703]	14
letta	3385d303c440236f7b74d3dd8d33c61f8d28e570	3598	35	112	25	157	3563	[259, 327, 377, 330, 78, 400, 20, 502, 22, 89, 732, 605, 94, 703]	14
letta	347d5e4ba82eda68d60debeb48095c992597ae13	3597	35	113	25	158	3562	[259, 327, 377, 330, 78, 400, 20, 502, 22, 89, 732, 605, 94, 703]	14
letta	35e6f098e04ff63b5929884914851ff69f6dfcf0	3626	35	114	25	159	3591	[259, 327, 377, 330, 78, 400, 20, 502, 22, 89, 732, 605, 94, 703]	14
letta	3b742b8f8044784d5c50914ef966248f62de469b	3594	35	112	25	157	3559	[259, 327, 377, 330, 78, 400, 20, 502, 22, 89, 732, 605, 94, 703]	14
letta	3e96bf41d2f1f88211da270365379397deba95a4	3594	35	112	25	157	3559	[259, 327, 377, 330, 78, 400, 20, 502, 22, 89, 732, 605, 94, 703]	14
letta	3fa3126c311368102052c63d967483da60e40883	3729	36	113	25	159	3694	[259, 327, 377, 330, 78, 400, 20, 502, 22, 89, 732, 605, 94, 703]	14
letta	3ffc9512f115b92bbf23f02bee82e83185bc466f	3621	35	114	25	159	3586	[259, 327, 377, 330, 78, 400, 20, 502, 22, 89, 732, 605, 94, 703]	14
letta	459fef1c5b089a5593294acb8e668bc1d259a2c5	3599	35	113	25	158	3564	[259, 327, 377, 330, 78, 400, 20, 502, 22, 89, 732, 605, 94, 703]	14
letta	472ad1ab721bea5c40d568ab33cd916ca4be6bd6	3599	35	113	25	158	3564	[259, 327, 377, 330, 78, 400, 20, 502, 22, 89, 732, 605, 94, 703]	14
letta	4c5cbbedb2d3eb6f31727322c9644d42bc831542	3629	35	114	25	159	3594	[259, 327, 377, 330, 78, 400, 20, 502, 22, 89, 732, 605, 94, 703]	14
letta	5086e133e72e3e5712c652b2d5e39c18bc72ab6d	3599	35	113	25	158	3564	[259, 327, 377, 330, 78, 400, 20, 502, 22, 89, 732, 605, 94, 703]	14

Overall Dataset-Level Analysis

This is the code saved in over_analysis.py to get results of RQ2 and RQ3 part:

```python
import os
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import ast

# Ensure overall dataset analysis folder exists
os.makedirs('overall_analysis', exist_ok=True)

# Combine data from all repositories
repos = ['litgpt', 'mem0', 'letta']
dfs = []
for repo in repos:
    repo_path = os.path.join('individual_analysis', f'{repo}_bandit_summary.csv')
    df = pd.read_csv(repo_path)
    # Convert unique_cwes from string to list
    df['unique_cwes'] = df['unique_cwes'].apply(ast.literal_eval)
    dfs.append(df)

combined_df = pd.concat(dfs, ignore_index=True)
combined_df.to_csv('overall_analysis/combined_bandit_summary.csv', index=False)

# Set seaborn style and palette for more distinct color choices
sns.set_theme(style="whitegrid")
color_palette = sns.color_palette("Set2", n_colors=3)  # Set2 palette for distinct colors
```

```python
# RQ2: Different severity patterns (repo-wise)
for repo in repos:
    plt.figure(figsize=(12, 6))
    repo_df = combined_df[combined_df['repo'] == repo].copy()
    repo_df['short_commit'] = repo_df['commit'].str[:5]  # Shorten commit to 5 characters

    for i, severity in enumerate(['high_sev', 'med_sev', 'low_sev']):
        plt.plot(repo_df['short_commit'], repo_df[severity],
                 marker='o', linestyle='-', color=color_palette[i],
                 label=severity, markersize=8, linewidth=2)

    plt.xlabel('Commit')
    plt.ylabel('Severity Count')
    plt.title(f'Severity Patterns Over Time ({repo})')
    plt.xticks(rotation=90)
    plt.legend()
    plt.tight_layout()
    plt.savefig(f'overall_analysis/RQ2_severity_patterns_{repo}.png')
    plt.close()

# RQ3: CWE coverage analysis
cwe_counts = combined_df['unique_cwes'].explode().value_counts()
plt.figure(figsize=(12, 6))
cwe_counts.plot(kind='bar', color=sns.color_palette("husl", n_colors=10))  # Different palette for bar chart
plt.xlabel('CWE ID')
plt.ylabel('Frequency')
plt.title('CWE Coverage Across Repositories')
plt.xticks(rotation=45, ha='right')  # Rotate and align right to avoid cutoff
plt.tight_layout()
plt.savefig('overall_analysis/RQ3_cwe_coverage.png')
plt.close()

print("Overall dataset analysis complete. Results saved in 'overall_analysis' folder.")
```

RQ1: When are high-severity vulnerabilities introduced and fixed?

A vulnerability is considered fixed when a previously detected CWE ID is no longer found in subsequent commits, the number of high-severity issues gradually decreases, and no new instances of the same CWE ID reappear over a sustained period.

<u>Purpose:</u>
This RQ investigates the timeline of high-severity vulnerabilities—when they appear and when they are removed—to understand security risk trends during OSS development.

<u>Approach:</u>
Monitored high-severity vulnerabilities in each commit across the selected repositories.
Analyzed the introduction and resolution of vulnerabilities over time.

<u>Results:</u>
litgpt: High-severity vulnerabilities remained stable at **22** with no significant reduction.
mem0: High-severity vulnerabilities remained stable at **5** with no significant reduction.
letta: High-severity vulnerabilities remained stable at **25** with no significant reduction.
Unique CWE IDs remain constant across all commits for each repository.

RQ2: Do vulnerabilities of different severity follow the same pattern?

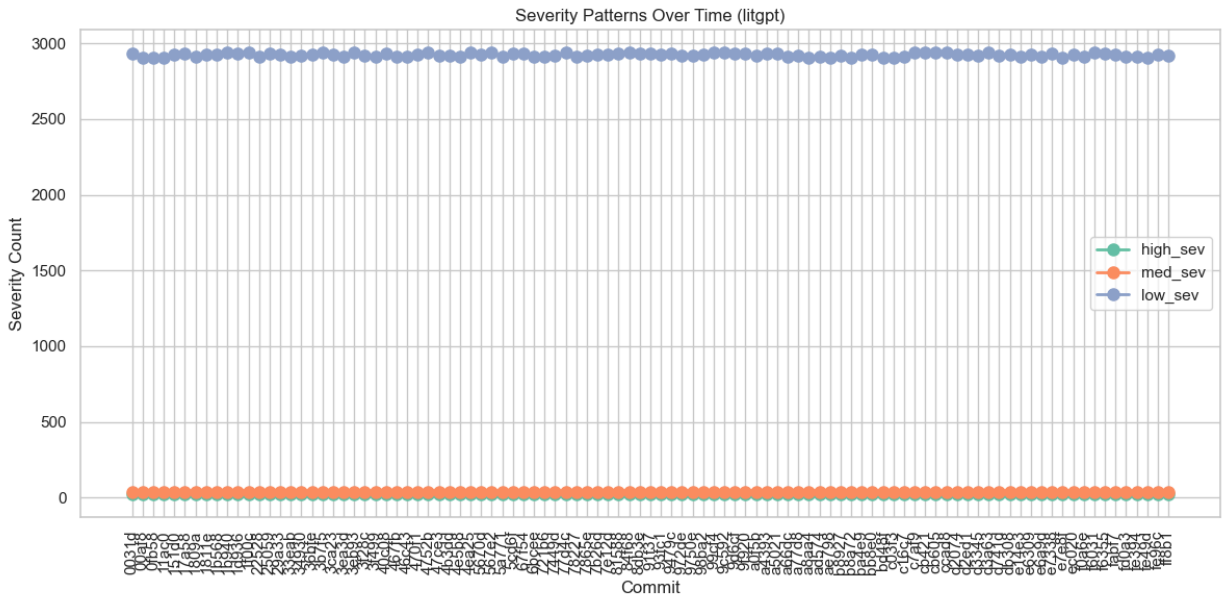
<u>Purpose:</u>
This RQ examines whether vulnerabilities of varying severity levels follow similar introduction and resolution patterns across different repositories.

<u>Approach:</u>
Monitored vulnerabilities of all severity levels in each commit.
Analyzed trends in the introduction and resolution of vulnerabilities over time for different severity levels.

Compared findings across three repositories.

<u>Results:</u>

Below is the severity trend graph for litgpt repository:



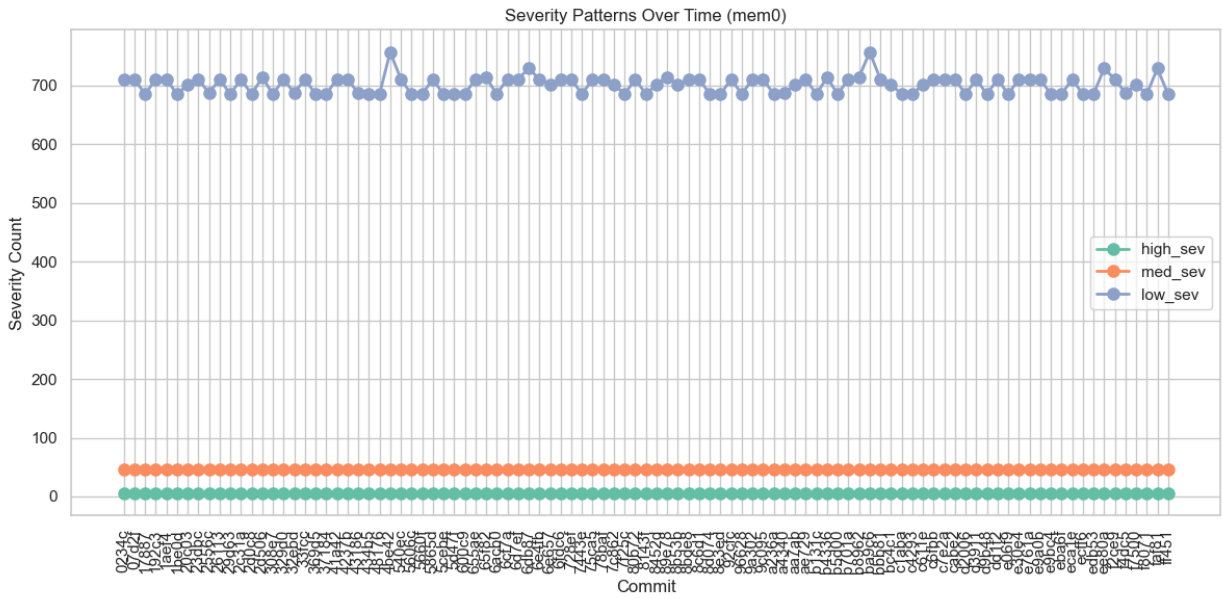
Observations:

- Low-severity issues dominate (~3000 per commit), staying consistent over time.
- Medium-severity issues are stable but much lower in count.
- High-severity issues are rare, indicating minimal critical failures.
- No major spikes or drops, showing a stable bug pattern.

Takeaways:

- Stable codebase, mostly minor issues.
- Critical failures are minimal, suggesting reliability.
- Potential for quality improvements in medium-severity areas.

Below is the severity trend graph for mem0 repository:



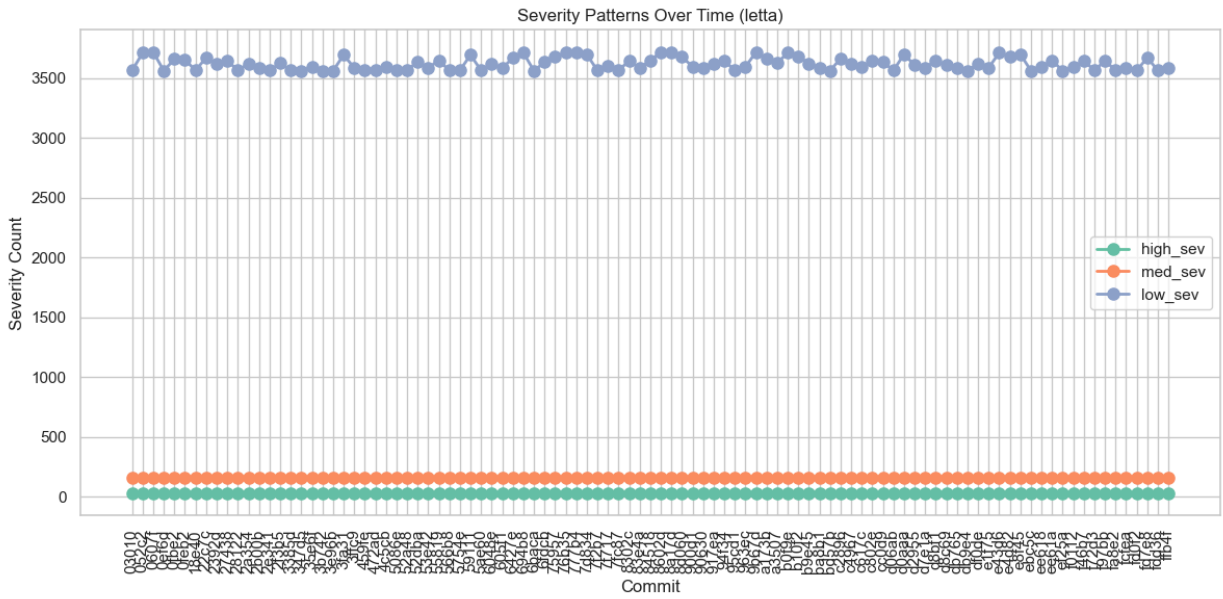
Observations:

- Low-severity issues (~700 per commit) are the most frequent and remain stable.
- Medium-severity issues (~50-100 per commit) show consistency with no major fluctuations.
- High-severity issues (~0-30 per commit) are the least frequent and remain stable.
- No significant spikes or drops in severity counts.

Takeaways:

- Code quality is stable, dominated by low-severity issues.
- High-severity issues are minimal, indicating reliable code.
- Medium-severity issues could be an area for improvement.

Below is the severity trend graph for letta repository:



Observations:

- Low-severity issues (~3500 per commit) are the most frequent and remain relatively stable.
- Medium-severity issues (~200 per commit) are consistently present with minimal fluctuations.
- High-severity issues (~50 per commit) are the least frequent but appear in every commit.
- No major spikes or dips in severity counts across commits, suggesting a consistent pattern.

Takeaways:

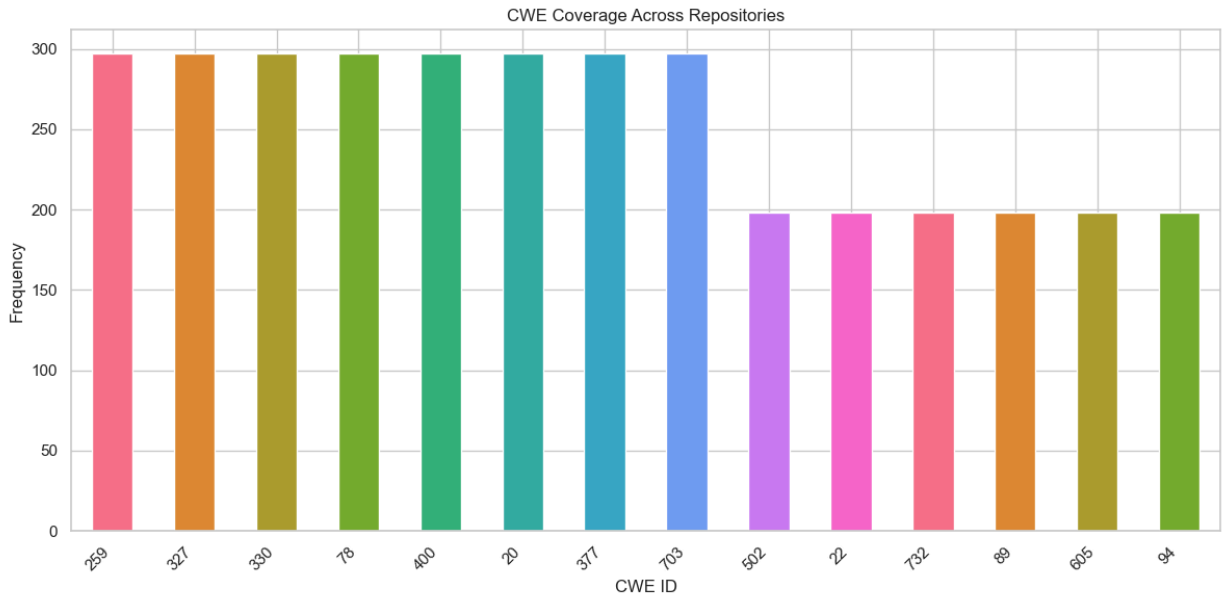
- Code quality is stable, dominated by low-severity issues.
- High-severity issues are present but limited, which is a positive sign.
- Medium-severity issues should be monitored to prevent future increases.

RQ3: Which CWEs are most frequent across repositories?

<u>Purpose:</u>
This RQ identifies the most commonly occurring CWE (Common Weakness Enumeration) vulnerabilities in OSS repositories.

<u>Approach:</u>
Retrieved CWE data from multiple repositories.
Analyzed the frequency distribution of various CWE categories.
Determined the most prevalent CWEs by comparing their occurrences across repositories.

<u>Results:</u>



Observations:

- Most frequently occurring CWEs (~300 occurrences each):
  - CWE-259: Use of Hardcoded Credentials
  - CWE-327: Usage of Cryptographically Weak Algorithms
  - CWE-330: Insufficiently Random Value Generation
  - CWE-78: Injection of OS Commands
  - CWE-400: Excessive Resource Consumption
  - CWE-20: Improper Validation of User Input
  - CWE-377: Unsafe Temporary File Management
  - CWE-703: Incorrect Handling of Exceptional Cases
- Less frequently occurring CWEs (~200 occurrences each):
  - CWE-502: Deserialization of Untrusted Input
  - CWE-22: Directory Traversal Attack
  - CWE-732: Weak File or Resource Permissions
  - CWE-89: SQL Query Injection
  - CWE-605: Repeated Binding of the Same Port
  - CWE-94: Execution of Injected Code
- The top CWEs are all security-related, indicating prevalent security weaknesses across repositories.
- CWE-259 and CWE-327 are among the most frequent, suggesting authentication and cryptographic issues are common.
- CWE-502 and CWE-22 appear less frequently but are still significant.

Takeaways:

- Authentication and cryptographic weaknesses dominate, requiring better security practices.
- Input validation issues (CWE-20, CWE-78) are also frequent, indicating potential command injection risks.

- Deserialization and path traversal vulnerabilities need attention, as they can lead to remote code execution or file access attacks.
- Security teams should prioritize fixing top CWE categories to enhance repository security.

Discussion and Conclusion

Challenges Faced

Bandit's detection algorithms had limitations, often failing to identify complex or context-specific vulnerabilities accurately. Processing large commit histories took significant time.

Lessons Learned

This experiment emphasized the need to choose repositories with ongoing development for effective vulnerability analysis. Dependency management was essential to guarantee seamless running of security tools such as Bandit. Post-processing Bandit output was required to eliminate false positives and prioritize remediation by severity. Visualization of security trends gave better insights into how vulnerabilities change over time. Issues such as hardcoded passwords and poor encryption were common problems that highlighted the requirements for stronger authentication and crypto practices. Automating the analysis heightened efficiency, and learning from patterns of vulnerability fixing confirmed the importance of ongoing security scanning in open-source software projects.

Resources:

- pytest Documentation
- pytest-xdist Documentation
- pytest-run-parallel Documentation
- keon/algorithms Repository
- Common Weakness Enumeration (CWE)
- CWE Top 25 - 2024
- Bandit Documentation
- SEART GitHub Search Engine
- Lecture 5 slides
- pynguin
- coverage
- pytest
- pytest-cov
- pytest-func-cov

Results Zip File: This file includes all the analyzed results obtained during the lab assignment. It can be accessed here.

Complete Zip File: This file contains execution logs, screenshots, and all result files. It is available for download here.