CS 202 Software Tools and Techniques for CSE

Lab 5 - Code Coverage Analysis and Test Generation

Introduction :

In this lab, we will analyze and measure different types of code coverage and generate unit test cases using automated testing tools. Our goal is to understand how well a given test suite exercises a Python program by checking line, branch, and function coverage. We will use tools like pytest, pytest-cov, and coverage to measure these metrics and pynguin to generate additional test cases. By comparing the effectiveness of existing and generated test cases, we aim to improve overall code coverage. This lab will help us learn how to evaluate test quality, visualize coverage reports, and identify gaps in testing.

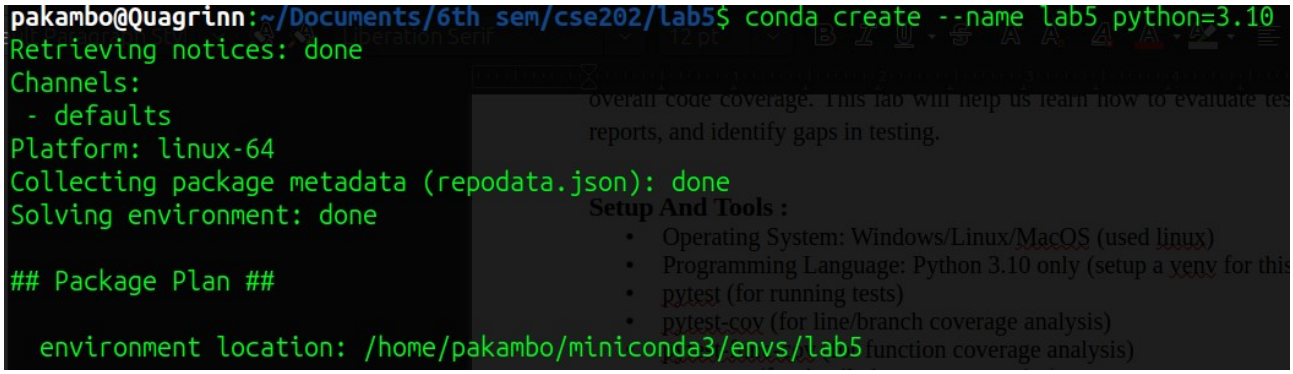
Setup And Tools :

- Operating System: Windows/Linux/MacOS (used linux)
- Programming Language: Python 3.10 only (setup a venv for this)
- pytest (for running tests)
- pytest-cov (for line/branch coverage analysis)
- pytest-func-cov (for function coverage analysis)
- coverage (for detailed coverage metrics)
- pynguin (for automated unit test generation)
- Some other tools (genhtml, lcov) introduced in Lecture 5

Methodology and Execution :

- ***Environment and tool setup :***

→ Created a venv in conda using the command “conda create –name stt python=3.10” in my lab 5 directory.



→ Activate the env with the ‘conda activate stt’ command.

→ Installed the required tools using the command ‘pip install pytest pytest-cov pytest-func-cov coverage pynguin’

```
(lab5) pakambo@Quagrinn:~/Documents/6th sem/cse202/lab5$ pip install pytest pytest-cov py
test-func-cov coverage pynguin
Collecting pytest
  Using cached pytest-8.3.5-py3-none-any.whl.metadata (7.6 kB)
Collecting pytest-cov
  Using cached pytest_cov-6.0.0-py3-none-any.whl.metadata (27 kB)
Collecting pytest-func-cov
  Downloading pytest_func_cov-0.2.3-py3-none-any.whl.metadata (3.1 kB)
Collecting coverage
  Downloading coverage-7.7.1-cp310-cp310-manylinux_2_5_x86_64.manylinux1_x86_64.manylinux
_2_17_x86_64.manylinux2014_x86_64.whl.metadata (8.5 kB)
```

- ***Lab activities :***

(1) *Cloning repository :*

→ Cloned the repository keon/algorithms in my lab 5 directory. The current commit hash is cad4754bc71742c2d6fcbd3b92ae74834d359844.

```
(lab5) pakambo@Quagrinn:~/Documents/6th sem/cse202/lab5$ git clone https://github.com/keo
n/algorithms.git
Cloning into 'algorithms'...
remote: Enumerating objects: 5188, done.
remote: Counting objects: 100% (33/33), done.
remote: Compressing objects: 100% (19/19), done.
remote: Total 5188 (delta 23), reused 14 (delta 14), pack-reused 5155 (from 2)
Receiving objects: 100% (5188/5188), 1.43 MiB | 817.00 KiB/s, done.
Resolving deltas: 100% (3242/3242), done.
```

→ Set up the required dependencies using the command ‘pip install -r requirements.txt’ and ‘pip install -r test_requirements.txt’

```
(lab5) pakambo@Quagrinn:~/Documents/6th sem/cse202/lab5/algorithms$ pip install -r requir
ements.txt
(lab5) pakambo@Quagrinn:~/Documents/6th sem/cse202/lab5/algorithms$ pip install -r test_r
equirements.txt
Collecting flake8 (from -r test_requirements.txt (line 1))
  Downloading flake8-7.1.2-py2.py3-none-any.whl.metadata (3.8 kB)
Collecting python-coveralls (from -r test_requirements.txt (line 2))
```

(2) Configure tools to inspect the repository :

→ In the root of the repository (algorithms/), create a pytest.ini file to specify which directories to include in testing:

```
pakambo@Quagrinn: ~/Documents/6th sem/cse202/lab5/algorithms
 GNU nano 7.2                                pytest.ini *
[pytest]
testpaths = tests
addopts = --cov=algorithms --cov-branch --cov-report=term-missing
```

→ Ran the command ‘pytest --collect-only’ to check if pytest is correctly set up. But got some errors.

```
(lab5) pakambo@Quagrinn:~/Documents/6th sem/cse202/lab5/algorithms$ pytest --collect-only
============================= test session starts ==============================
platform linux -- Python 3.10.16, pytest-8.3.5, pluggy-1.5.0
rootdir: /home/pakambo/Documents/6th sem/cse202/lab5/algorithms
configfile: pytest.ini
testpaths: tests
plugins: func-cov-0.2.3, cov-6.0.0
collected 0 items / 29 errors
/home/pakambo/miniconda3/envs/lab5/lib/python3.10/site-packages/coverage/control.py:907:
CoverageWarning: No data was collected. (no-data-collected)
  self._warn("No data was collected.", slug="no-data-collected")
```

→ There were 29 errors and majority of them are module not found errors.

```
========================== short test summary info ===========================
ERROR tests/test_array.py
ERROR tests/test_automata.py
ERROR tests/test_backtrack.py
ERROR tests/test_bfs.py
ERROR tests/test_bit.py
ERROR tests/test_compression.py
ERROR tests/test_dfs.py
ERROR tests/test_dp.py
ERROR tests/test_graph.py
ERROR tests/test_greedy.py
ERROR tests/test_heap.py
ERROR tests/test_histogram.py
ERROR tests/test_iterative_segment_tree.py
ERROR tests/test_linkedlist.py
ERROR tests/test_map.py
ERROR tests/test_maths.py
ERROR tests/test_matrix.py
ERROR tests/test_ml.py
ERROR tests/test_monomial.py
ERROR tests/test_polynomial.py
ERROR tests/test_queues.py
ERROR tests/test_search.py
ERROR tests/test_set.py
ERROR tests/test_sort.py
ERROR tests/test_stack.py
ERROR tests/test_streaming.py
ERROR tests/test_strings.py
ERROR tests/test_tree.py
ERROR tests/test_unix.py
!!!!!!!!!!!!!!!!!!!! Interrupted: 29 errors during collection !!!!!!!!!!!!!!!!!!!!
```

→ Fixed the module not found errors using ‘pip install -e .’

```
(algos) pakambo@Quagrinn:~/Documents/6th sem/cse202/lab6/algorithms$ pip install -e .
Obtaining file:///home/pakambo/Documents/6th%20sem/cse202/lab6/algorithms
  Installing build dependencies ... done
  Checking if build backend supports build_editable ... done
  Getting requirements to build editable ... done
  Preparing editable metadata (pyproject.toml) ... done
Building wheels for collected packages: algorithms
  Building editable for algorithms (pyproject.toml) ... done
  Created wheel for algorithms: filename=algorithms-0.1.4-0.editable-py3-none-any.whl size=8644 sha256=a7d23f5af
a91ba1760d0d3532fd8351b26d7ed58f4b9d2c3d92b8d39bbcfedc3
  Stored in directory: /tmp/pip-ephem-wheel-cache-0rje853b/wheels/a5/a2/ba/dae627835974516fe865c522fc43b1113f855
e9d1f97e64b32
Successfully built algorithms
Installing collected packages: algorithms
Successfully installed algorithms-0.1.4
```

→ There was also an syntax error which was fixed by adding a comma at the end of line 12 in test_array.py file.

```
E     File "/home/pakambo/Documents/6th sem/cse202/lab6/algorithms/tests/test_array.py", line 13
E       rotate_v1, rotate_v2, rotate_v3,
E       ^^^^^^^^^
E   SyntaxError: invalid syntax
```

algorithms > tests > test_array.py > ...

You, 3 weeks ago | 10 authors (Rahul Goswami and others)

```python
from algorithms.arrays import (
    delete_nth, delete_nth_naive,
    flatten_iter, flatten,
    garage,
    josephus,
    longest_non_repeat_v1, longest_non_repeat_v2,
    get_longest_non_repeat_v1, get_longest_non_repeat_v2,
    Interval, merge_intervals,
    missing_ranges,
    move_zeros,
    plus_one_v1, plus_one_v2, plus_one_v3,
    remove_duplicates,
    rotate_v1, rotate_v2, rotate_v3,
    summarize_ranges,
    three_sum,
    two_sum,
    max_ones_index
```

→ Then ran the command ‘pytest –collect-only’ again. This listed all discovered test files under tests/, confirming that only this repository is being analyzed.

```
(lab5) pakambo@Quagrinn:~/Documents/6th sem/cse202/lab5/algorithms$ pytest --collect-only
============================= test session starts ==============================
platform linux -- Python 3.10.16, pytest-8.3.5, pluggy-1.5.0
rootdir: /home/pakambo/Documents/6th sem/cse202/lab5/algorithms
configfile: pytest.ini
testpaths: tests
plugins: func-cov-0.2.3, cov-6.0.0
collected 416 items

<Dir algorithms>
  <Dir tests>
    <Module test_array.py>
```

→ Ran ‘pytest –cov=algorithms’ to check if pytest-cov is properly limiting analysis to algorithms/,

```
(lab5) pakambo@Quagrinn:~/Documents/6th sem/cse202/lab5/algorithms$ pytest --cov=algorithms
============================= test session starts ==============================
platform linux -- Python 3.10.16, pytest-8.3.5, pluggy-1.5.0
rootdir: /home/pakambo/Documents/6th sem/cse202/lab5/algorithms
configfile: pytest.ini
testpaths: tests
plugins: func-cov-0.2.3, cov-6.0.0
collected 416 items

tests/test_array.py ..................F...F.......                          [  6%]
tests/test_automata.py .                                                     [  7%]
```

(3) <u>Execute test suite A and record coverage metrics :</u>

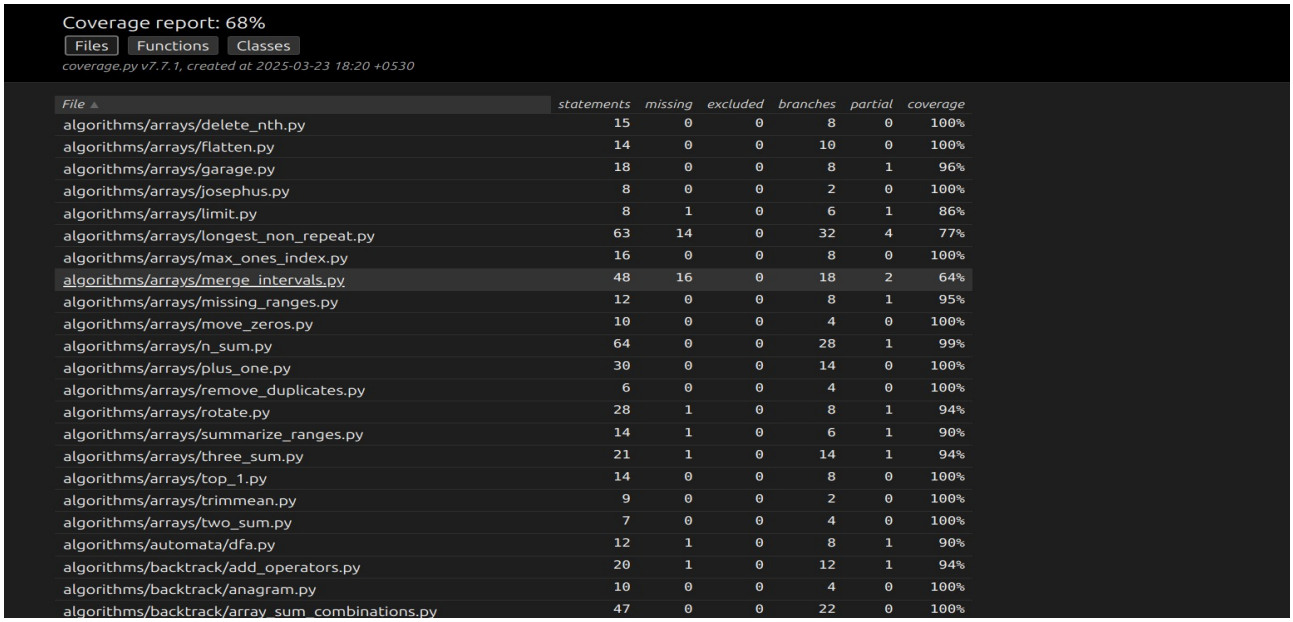
→ Ran the test suite A with coverage measurement.

```
(lab5) pakambo@Quagrinn:~/Documents/6th sem/cse202/lab5/algorithms$ pytest --cov=algorithms --cov-branch --cov-report=term-missing
============================= test session starts ==============================
platform linux -- Python 3.10.16, pytest-8.3.5, pluggy-1.5.0
rootdir: /home/pakambo/Documents/6th sem/cse202/lab5/algorithms
configfile: pytest.ini
testpaths: tests
plugins: func-cov-0.2.3, cov-6.0.0
collected 416 items

tests/test_array.py ..................F...F.......                          [  6%]
tests/test_automata.py .                                                     [  7%]
```

→ This command will run all test cases in the tests/ folder, measure coverage for the algorithms/ directory and also shows which lines are missing coverage directly in the terminal.
→ Now generated a covergae report using the command 'coverage html'. This creates a htmlcov/ directory containing an interactive report which can be opened in a browser using the followign command.

```
(lab5) pakambo@Quagrinn:~/Documents/6th sem/cse202/lab5/algorithms$ coverage html
Wrote HTML report to htmlcov/index.html
(lab5) pakambo@Quagrinn:~/Documents/6th sem/cse202/lab5/algorithms$ xdg-open htmlcov/inde
x.html
(lab5) pakambo@Quagrinn:~/Documents/6th sem/cse202/lab5/algorithms$ Opening in existing b
rowser session.
```



Coverage report: 68%

Files | Functions | Classes

coverage.py v7.7.1, created at 2025-03-23 18:20 +0530

File	statements	missing	excluded	branches	partial	coverage
algorithms/arrays/delete_nth.py	15	0	0	8	0	100%
algorithms/arrays/flatten.py	14	0	0	10	0	100%
algorithms/arrays/garage.py	18	0	0	8	1	96%
algorithms/arrays/josephus.py	8	0	0	2	0	100%
algorithms/arrays/limit.py	8	1	0	6	1	86%
algorithms/arrays/longest_non_repeat.py	63	14	0	32	4	77%
algorithms/arrays/max_ones_index.py	16	0	0	8	0	100%
algorithms/arrays/merge_intervals.py	48	16	0	18	2	64%
algorithms/arrays/missing_ranges.py	12	0	0	8	1	95%
algorithms/arrays/move_zeros.py	10	0	0	4	0	100%
algorithms/arrays/n_sum.py	64	0	0	28	1	99%
algorithms/arrays/plus_one.py	30	0	0	14	0	100%
algorithms/arrays/remove_duplicates.py	6	0	0	4	0	100%
algorithms/arrays/rotate.py	28	1	0	8	1	94%
algorithms/arrays/summarize_ranges.py	14	1	0	6	1	90%
algorithms/arrays/three_sum.py	21	1	0	14	1	94%
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/dfa.py	12	1	0	8	1	90%
algorithms/backtrack/add_operators.py	20	1	0	12	1	94%
algorithms/backtrack/anagram.py	10	0	0	4	0	100%
algorithms/backtrack/array_sum_combinations.py	47	0	0	22	0	100%

(4) Analyze test suite A coverage :

File	Statements	Missing	Excluded	Branches	Partial	Coverage
algorithms/arrays/delete_nth.py	15	0	0	8	0	100%
algorithms/arrays/flatten.py	14	0	0	10	0	100%
...	...	...	...	...	...	...
Total	7994	2468	0	3780	250	68%

The generated HTML report contains the following columns:

- Statements : Total number of executable lines of code.
- Missing : Lines of code not executed by the test suite.
- Excluded : Lines ignored in coverage analysis (e.g., comments, # pragma: no cover).
- Branches : Number of conditional branches (if, elif, else) in the code.
- Partial : Branches that were only partially executed.
- Coverage : Percentage of executed statements out of the total.

→ We got a total coverage of 68%, meaning only 68% of the code is covered in tests.
→ Out of 7994 lines, 2468 were not executed at all and out of 3780 branches, a sum of 250 remains untested (partial branches)
→ Some of the files had 100% coverage, while others have less coverage values.

→ Now for visualization, I generated a lcov report and then generated an html report for the same using the following commands.

```
(lab5) pakambo@Quagrinn:~/Documents/6th sem/cse202/lab5/algorithms$ coverage lcov -o cove
rage.info
Wrote LCOV report to coverage.info
(lab5) pakambo@Quagrinn:~/Documents/6th sem/cse202/lab5/algorithms$ genhtml coverage.info
 --output-directory coverage_html
Found 331 entries.
Found common filename prefix "/home/pakambo/Documents/6th sem/cse202/lab5/algorithms/algo
rithms"
Generating output.
Processing file algorithms/search/two_sum.py
  lines=29 hit=27 functions=3 hit=3
```

→ This will generate a folder named coverage_html with related files.
→ Opened the html page in local browser using the following command.

```
(lab5) pakambo@Quagrinn:~/Documents/6th sem/cse202/lab5/algorithms$ xdg-open coverage_htm
l/index.html
(lab5) pakambo@Quagrinn:~/Documents/6th sem/cse202/lab5/algorithms$ Opening in existing b
rowser session.
```

LCOV - code coverage report

			Coverage	Total	Hit
Current view:	top level	**Lines:**	69.1 %	7994	5526
Test:	coverage.info	**Functions:**	70.1 %	877	615
Test Date:	2025-03-23 23:01:00				

Directory	Line Coverage Rate	Line Coverage Total	Line Coverage Hit	Function Coverage Rate	Function Coverage Total	Function Coverage Hit
arrays/algorithms/arrays	91.6 %	405	371	83.0 %	47	39
automata/algorithms/automata	91.7 %	12	11	100.0 %	1	1
backtrack/algorithms/backtrack	96.1 %	285	274	97.3 %	37	36
bfs/algorithms/bfs	77.1 %	109	84	66.7 %	6	4
bit/algorithms/bit	99.5 %	204	203	100.0 %	30	30
compression/algorithms/compression	92.2 %	257	237	82.5 %	40	33
dfs/algorithms/dfs	90.9 %	176	160	88.9 %	18	16
distribution/algorithms/distribution	100.0 %	5	5	100.0 %	1	1
dp/algorithms/dp	62.0 %	440	273	62.2 %	45	28
graph/algorithms/graph	54.8 %	781	428	46.7 %	90	42
greedy/algorithms/greedy	91.7 %	12	11	100.0 %	1	1
heap/algorithms/heap	77.0 %	122	94	55.6 %	18	10
linkedlist/algorithms/linkedlist	31.4 %	593	186	34.0 %	50	17
map/algorithms/map	68.4 %	250	171	69.7 %	33	23
maths/algorithms/maths	73.3 %	961	704	78.1 %	105	82
matrix/algorithms/matrix	69.6 %	516	359	71.8 %	39	28
ml/algorithms/ml	100.0 %	20	20	100.0 %	2	2
queues/algorithms/queues	73.4 %	173	127	63.6 %	33	21
search/algorithms/search	94.7 %	207	196	100.0 %	20	20
set/algorithms/set	7.9 %	101	8	11.1 %	9	1
sort/algorithms/sort	79.1 %	460	364	87.5 %	40	35
stack/algorithms/stack	83.2 %	273	227	84.2 %	38	32
streaming/algorithms/streaming	100.0 %	51	51	100.0 %	5	5
strings/algorithms/strings	84.2 %	768	647	89.7 %	78	70
tree/algorithms/tree	31.8 %	528	168	32.2 %	59	19
tree/avl/algorithms/tree/avl	0.0 %	77		0.0 %	8	
tree/fenwick_tree/algorithms/tree/fenwick_tree	100.0 %	21	21	100.0 %	4	4
tree/segment_tree/algorithms/tree/segment_tree	100.0 %	25	25	100.0 %	4	4
tree/traversal/algorithms/tree/traversal	55.6 %	135	75	54.5 %	11	6
unix/path/algorithms/unix/path	96.3 %	27	26	100.0 %	5	5

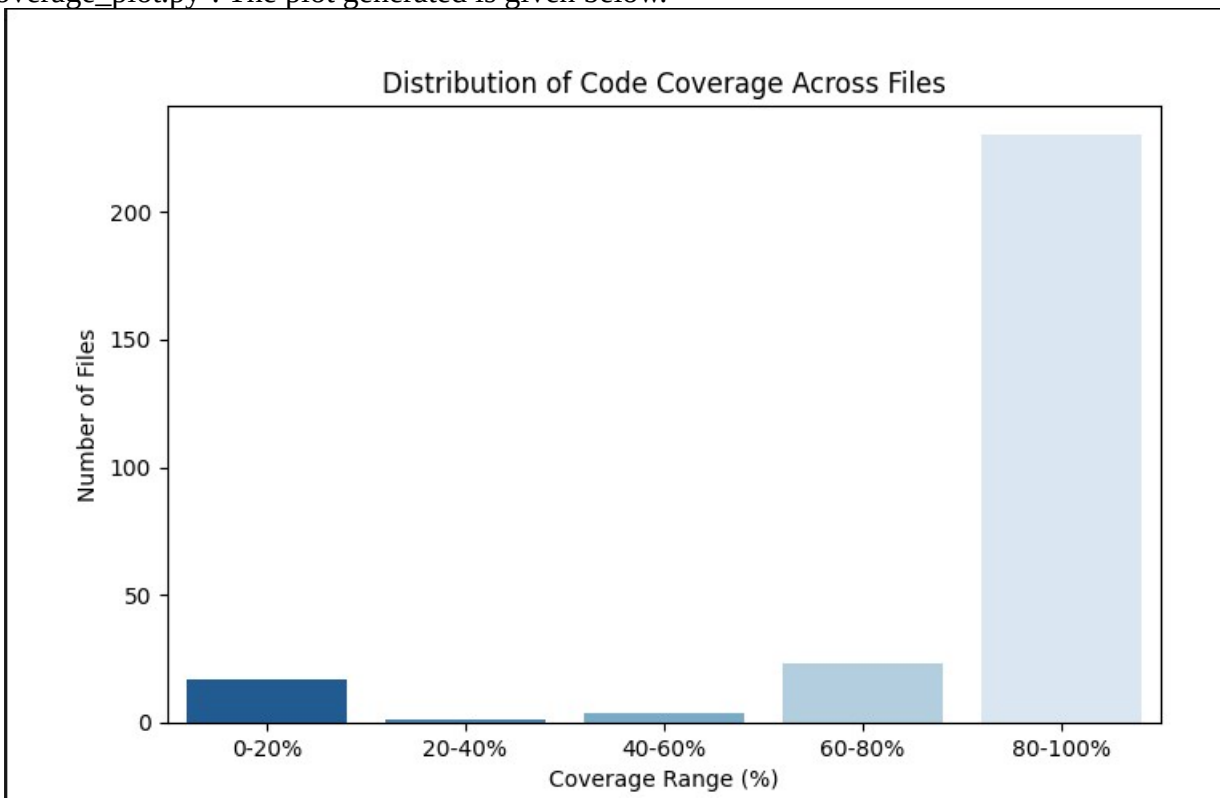
→ The above image shows shows clear visualizations of Line coverage and function coverage. It also gave us the metrics of line and function coverage.

Line coverage = 69.1%
Function Coverage = 70.1%
Branch Coverage = 93.38% [(3530/3780)*100]

→ Now stored this coverage data in .json format with the following command.

```
(lab5) pakambo@Quagrinn:~/Documents/6th sem/cse202/lab5/algorithms$ coverage json -o cove
rage_existing.json
Wrote JSON report to coverage_existing.json
(lab5) pakambo@Quagrinn:~/Documents/6th sem/cse202/lab5/algorithms$
```

→ Then wrote a python script to generate coverage distributions for files and named it as ‘coverage_plot.py’. The plot generated is given below.



(5) <u>Test suite B for uncovered files:</u>

→ Wrote a python script to list out the files that are not completely covered. Named the script as ‘low_coverage.py’.

→ Ran the ‘low_coverage.py’ file and got the following results. It listed many files in the algorithms repository. The results are also stored in a file named “low_coverage_files.txt”

```
(lab5) pakambo@Quagrinn:~/Documents/6th sem/cse202/lab5/algorithms$ python low_coverage.py
Files needing more tests: ['algorithms/arrays/garage.py', 'algorithms/arrays/limit.py', 'algorithms/arrays/longest_non_repeat.py', 'algorithms/arrays/merge_intervals.py', 'algorithms/arrays/missing_ranges.py', 'algorithms/arrays/n_sum.py', 'algorithms/arrays/rotate.py', 'algorithms/arrays/summarize_ranges.py', 'algorithms/arrays/three_sum.py', 'algorithms/automata/dfa.py', 'algorithms/backtrack/add_operators.py', 'algorithms/backtrack/letter_combination.py', 'algorithms/backtrack/palindrome_partitioning.py', 'algorithms/backtrack/pattern_match.py', 'algorithms/backtrack/permute.py', 'algorithms/bfs/maze_search.py', 'algorithms/bfs/shortest_distance_from_all_buildings.py', 'algorithms/bfs/word_ladder.py', 'algorithms/bit/flip_bit_longest_sequence.py', 'algorithms/bit/has_alternative_bit.py', 'algorithms/compression/huffman_coding.py', 'algorithms/compression/rle_compression.py', 'algorithms/dfs/maze_search.py', 'algorithms/dfs/pacific_atlantic.py', 'algorithms/dfs/sudoku_solver.py', 'algorithms/dp/fib.py', 'algorithms/dp/hosoya_triangle.py', 'algorithms/dp/job_scheduling.py', 'algorithms/dp/k_factor.py', 'algorithms/dp/longest_common_subsequence.py', 'algorithms/dp/longest_increasing.py', 'algorithms/dp/matrix_chain_order.py', 'algorithms/dp/max_product_subarray.py', 'algorithms/dp/min_cost_path.py', 'algorithms/dp/num_decodings.py', 'algorithms/dp/word_break.py', 'algorithms/graph/bellman_ford.py', 'algorithms/graph/check_bipartite.py', 'algorithms/graph/check_digraph_strongly_connected.py', 'algorithms/graph/clone_graph.py', 'algorithms/graph/count_connected_number_of_component.py', 'algorithms/graph/find_all_cliques.py', 'algorithms/graph/find_path.py', 'algorithms/graph/graph.py', 'algorithms/graph/markov_chain.py', 'algorithms/graph/maximum_flow.py', 'algorithms/graph/minimum_spanning_tree.py', 'algorithms/graph/satisfiability.py', 'algorithms/graph/strongly_connected_components_kosaraju.py', 'algorithms/graph/transitive_closure_dfs.py', 'algorithms/graph/traversal.py', 'algorithms/greedy/max_contiguous_subsequence_sum.py', 'algorithms/heap/binary_heap.py', 'algorithms/heap/merge_sorted_k_lists.py', 'algorithms/heap/sliding_window_max.py', 'algorithms/linkedlist/add_two_numbers.py', 'algorithms/linkedlist/delete_node.py', 'algorithms/linkedlist/first_cyclic_node.py', 'algorithms/linkedlist/intersection.py', 'algorithms/linkedlist/is_cyclic.py', 'algorithms/linkedlist/is_palindrome.py', 'algorithms/linkedlist/is_sorted.py', 'algorithms/linkedlist/kth_to_last.py', 'algorithms/linkedlist/linkedlist.py', 'algorithms/linkedlist/partition.py', 'algorithms/linkedlist/remove_duplicates.py', 'algorithms/linkedlist/remove_range.py', 'algorithms/linkedlist/reverse.py', 'algorithms/linkedlist/rotate_list.py', 'algorithms/linkedlist/swap_in_pairs.py', 'algorithms/map/hashtable.py', 'a
```

→ Now that we have the required file names, I set up the environment variable for pynguin tool.

```
(lab5) pakambo@Quagrinn:~/Documents/6th sem/cse202/lab5/algorithms$ export PYNGUIN_DANGER_AWARE=1
(lab5) pakambo@Quagrinn:~/Documents/6th sem/cse202/lab5/algorithms$
```

→ Setting this environment variable will enable the dangerous test generation, allowing Pynguin to generate extreme edge cases.

→ Wrote a bash script which generates test cases to all the low coverage files using pynguin and named it ‘testgeneration.sh’.

→ The bashscript will generate tests for all the file in the low_coverage_files.txt. A timeout of 210 seconds is set to generate the test cases due to time constraints. If the time limit exceeds the timeout or an error occurs while generating test cases for that particular file during the specified timeout, test case generation for those particular files will be skipped and these files will be recorded in a log file name ‘skipped_files.log’ along with the reason for skipping the files.

```
algorithms > $ testgeneration.sh
#!/bin/bash

# Set timeout duration in seconds
TIMEOUT_DURATION=210

# Log file for skipped files
SKIPPED_LOG="skipped_files.log"
echo "Skipped Files Log" > $SKIPPED_LOG
echo "=================" >> $SKIPPED_LOG

# Read file paths (skipping first line)
tail -n +2 low_coverage_files.txt | while read file; do
    module_name=$(echo $file | sed 's/\//./g' | sed 's/.py$//')
    echo "Generating tests for $module_name"

    # Run Pynguin with timeout and capture errors
    ERROR_OUTPUT=$(timeout $TIMEOUT_DURATION pynguin --project-path . --module-name $module_name --output-path generated_tests/ 2>&1)

    # Check exit status
    EXIT_CODE=$?

    if [[ $EXIT_CODE -eq 124 ]]; then
        echo "Skipping $module_name due to timeout!"
        echo "$module_name - Skipped due to timeout" >> $SKIPPED_LOG
    elif [[ $EXIT_CODE -ne 0 ]]; then
        echo "Skipping $module_name due to an error!"
        echo "$module_name - Skipped due to error: $ERROR_OUTPUT" >> $SKIPPED_LOG
    fi
done
```

```
(lab5) pakambo@Quagrinn:~/Documents/6th sem/cse202/lab5/algorithms$ export PYNGUIN_DANGER_AWARE=1
(lab5) pakambo@Quagrinn:~/Documents/6th sem/cse202/lab5/algorithms$ ./testgeneration.sh
Generating tests for algorithms.arrays.garage
Skipping algorithms.arrays.garage due to timeout!
Generating tests for algorithms.arrays.limit
Generating tests for algorithms.arrays.longest_non_repeat
Generating tests for algorithms.arrays.merge_intervals
```

→ The test cases generated are stored in the folder named ‘generated_tests’.
→ Were able to generate tests for only 27 files. Some files ran into “modulenotfound : no module named tree” error, but even after installing tree with “pip install tree”, same error repeated. And when rest of the skipped files were run without any time limit, some of them ran in to the error “RuntimeError: The current thread shall not be executed any more, thus I kill it.” and for rest of the files , pynguin generated the tests successfully.
→ The files for which test generation failed were written to skipped_files.log along with their reasons.

(6) Test suite A vs B:

→First run the generated test cases and measure the code coverage with the following command:

“pytest --cov=algorithms --cov-report=xml --cov-report=term generated_tests/”

```
FAILED generated_tests/test_algorithms_tree_traversal_inorder.py::test_case_2 - AttributeError: 'function' object has no attribute 'inorder'
FAILED generated_tests/test_algorithms_tree_traversal_inorder.py::test_case_3 - AttributeError: 'function' object has no attribute 'inorder_rec'
FAILED generated_tests/test_algorithms_tree_traversal_inorder.py::test_case_4 - AttributeError: 'function' object has no attribute 'Node'
FAILED generated_tests/test_algorithms_tree_traversal_postorder.py::test_case_1 - AttributeError: 'function' object has no attribute 'postorder'
FAILED generated_tests/test_algorithms_tree_traversal_postorder.py::test_case_2 - AttributeError: 'function' object has no attribute 'Node'
FAILED generated_tests/test_algorithms_tree_traversal_postorder.py::test_case_4 - AttributeError: 'function' object has no attribute 'Node'
FAILED generated_tests/test_algorithms_tree_traversal_postorder.py::test_case_7 - AttributeError: 'function' object has no attribute 'postorder'
FAILED generated_tests/test_algorithms_tree_traversal_preorder.py::test_case_2 - AttributeError: 'function' object has no attribute 'Node'
FAILED generated_tests/test_algorithms_tree_traversal_preorder.py::test_case_3 - AttributeError: 'function' object has no attribute 'Node'
========== 80 failed, 128 passed, 287 xfailed, 1 warning in 3.90s ==========
```

→ This will generate an XML report (coverage.xml) and display the coverage in the terminal.
→ The coverage for test suite B is as follows

```
----------
TOTAL                                                 7928   6724   3756     44    14%
Coverage XML written to file coverage.xml
```

→ Filter data for the files where tests were generated using the coverage XML report coverage.xml

```
(lab5) pakambo@Quagrinn:~/Documents/6th sem/cse202/lab5/algorithms$ coverage xml -o coverage_suite_B.xml
Wrote XML report to coverage_suite_B.xml
(lab5) pakambo@Quagrinn:~/Documents/6th sem/cse202/lab5/algorithms$
```

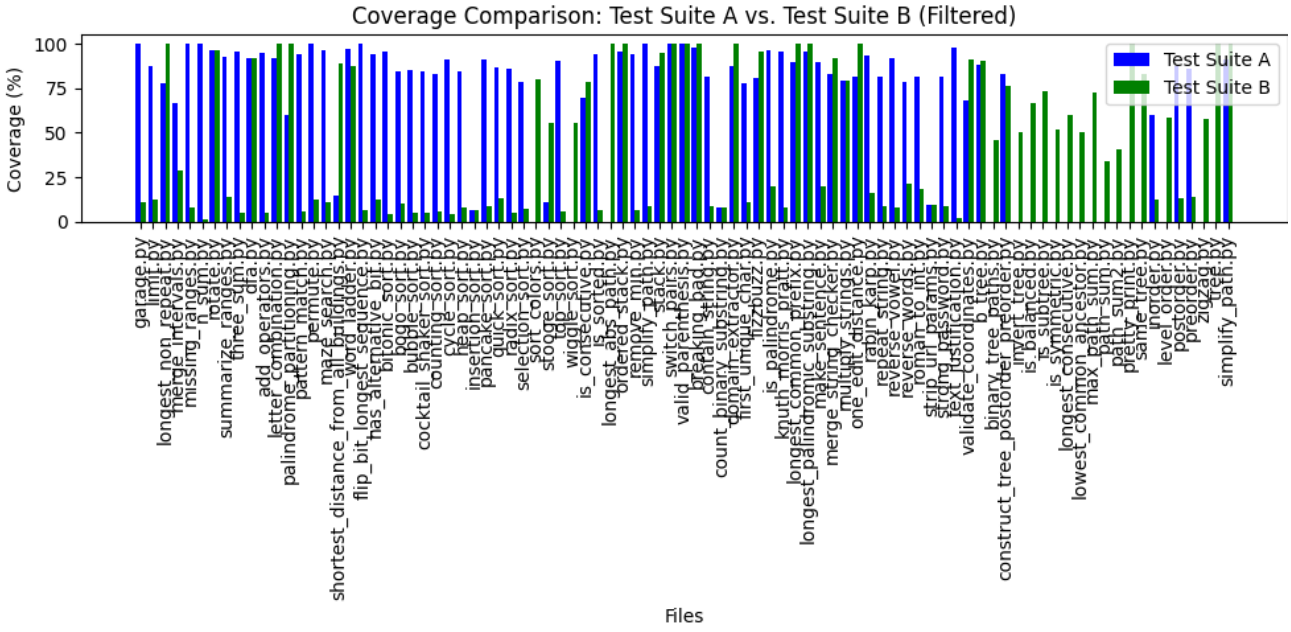
→ This will write the report to the file coverage_suite_B.xml
→ Then run the original test suite (test suite A) using “pytest --cov=algorithms --cov-report=xml –cov-report=term” for comparison and store the coverage data in a xml file.

```
(lab5) pakambo@Quagrinn:~/Documents/6th sem/cse202/lab5/algorithms$ pytest --cov=algorithms --cov-report=xml --cov-report=term
============================= test session starts ==============================
platform linux -- Python 3.10.16, pytest-8.3.5, pluggy-1.5.0
rootdir: /home/pakambo/Documents/6th sem/cse202/lab5/algorithms
configfile: pytest.ini
testpaths: tests
plugins: func-cov-0.2.3, cov-6.0.0
collected 416 items

tests/test_array.py .................F...F......                          [  6%]
tests/test_automata.py .                                                   [  7%]
tests/test_backtrack.py .......................                            [ 13%]
```

```
(lab5) pakambo@Quagrinn:~/Documents/6th sem/cse202/lab5/algorithms$ coverage xml -o coverage_suite_A.xml
Wrote XML report to coverage_suite_A.xml
```

→ This will write the test suite A coverage report to the file coverage_suite_A.xml.
→ Next I wrote a Python script that compares the code coverage of test suite A and test suite B, but only for the files that have tests generated in test suite B. This code is saved as "6_extract_cov_data.py" and it writes the comparison data to a file named coverage_results.txt.
→ Then wrote a python script to generate visualizations for the same and named it "A_vs_B_comp.py".



→ From the plot, we can observe that for majority of the files, test suite A has more coverage compared to Test suite B.

Overall Coverage change:
→ Now lets check how the overall coverage for the repository changed after generating test suite B.
→ First erased previous coverage data using the command "coverage erase"
→ Then ran all the test cases present in the repository i.e. both test suite A and B using the command "pytest generated_tests tests --cov=algorithms --cov-report=xml –cov-report=term".

```
FAILED generated_tests/test_algorithms_tree_traversal_postorder.py::test_case_2 - AttributeError: 'function' object has no attribu
FAILED generated_tests/test_algorithms_tree_traversal_postorder.py::test_case_4 - AttributeError: 'function' object has no attribu
FAILED generated_tests/test_algorithms_tree_traversal_postorder.py::test_case_7 - AttributeError: 'function' object has no attribu
FAILED generated_tests/test_algorithms_tree_traversal_preorder.py::test_case_2 - AttributeError: 'function' object has no attribut
FAILED generated_tests/test_algorithms_tree_traversal_preorder.py::test_case_3 - AttributeError: 'function' object has no attribut
FAILED tests/test_array.py::TestRemoveDuplicate::test_remove_duplicates - TypeError: TestCase.assertListEqual() missing 1 required
2'
FAILED tests/test_array.py::TestSummaryRanges::test_summarize_ranges - AssertionError: Lists differ: ['0-2', '4-5', '7'] != [(0, 2
========================================== 82 failed, 542 passed, 287 xfailed, 1 warning in 12.54s ==========================
```

```
----------- coverage: platform linux, python 3.10.16-final-0 -----------
TOTAL                                          7994   2187   3780    234    72%
Coverage XML written to file coverage.xml
```

→ Above figure shows the line coverage and branch coverage for combined tests.

→ Next converted the coverage data into lcov format using the comand ‘coverage lcov -o combined_coverage.info’ .

```
(lab5) pakambo@Quagrinn:~/Documents/6th sem/cse202/lab5/algorithms$ coverage lcov -o combined_coverage.info
Wrote LCOV report to combined_coverage.info
```

→ Then generated a html report for this combined coverage info using the command ‘genhtml combined_coverage.info --output-directory combined_coverage_html’. Then opened the html report using the command ‘xdg-open combined_coverage_html/index.html’.

```
(lab5) pakambo@Quagrinn:~/Documents/6th sem/cse202/lab5/algorithms$ genhtml comb
ined_coverage.info --output-directory combined_coverage_html
Found 331 entries.
Found common filename prefix "/home/pakambo/Documents/6th sem/cse202/lab5/algori
thms/algorithms"
Generating output.
Processing file algorithms/linkedlist/rotate_list.py
```

```
(lab5) pakambo@Quagrinn:~/Documents/6th sem/cse202/lab5/algorithms$ xdg-open com
bined_coverage_html/index.html
(lab5) pakambo@Quagrinn:~/Documents/6th sem/cse202/lab5/algorithms$ Opening in e
xisting browser session.
```

			Coverage	Total	Hit
Current view:	top level	Lines:	72.6 %	7994	5807
Test:	combined_coverage.info	Functions:	75.1 %	877	659
Test Date:	2025-03-25 01:30:07				

Directory	Line Coverage			Function Coverage		
	Rate	Total	Hit	Rate	Total	Hit
arrays/algorithms/arrays	95.3 %	405	386	85.1 %	47	40
automata/algorithms/automata	91.7 %	12	11	100.0 %	1	1
backtrack/algorithms/backtrack	99.3 %	285	283	100.0 %	37	37
bfs/algorithms/bfs	96.3 %	109	105	100.0 %	6	6
bit/algorithms/bit	99.5 %	204	203	100.0 %	30	30
compression/algorithms/compression	92.2 %	257	237	82.5 %	40	33
dfs/algorithms/dfs	90.9 %	176	160	88.9 %	18	16
distribution/algorithms/distribution	100.0 %	5	5	100.0 %	1	1
dp/algorithms/dp	62.0 %	440	273	62.2 %	45	28
graph/algorithms/graph	54.8 %	781	428	46.7 %	90	42
greedy/algorithms/greedy	91.7 %	12	11	100.0 %	1	1
heap/algorithms/heap	77.0 %	122	94	55.6 %	18	10
linkedlist/algorithms/linkedlist	31.4 %	593	186	34.0 %	50	17
map/algorithms/map	68.4 %	250	171	69.7 %	33	23
maths/algorithms/maths	73.3 %	961	704	78.1 %	105	82
matrix/algorithms/matrix	69.6 %	516	359	71.8 %	39	28
ml/algorithms/ml	100.0 %	20	20	100.0 %	2	2
queues/algorithms/queues	73.4 %	173	127	63.6 %	33	21
search/algorithms/search	94.7 %	207	196	100.0 %	20	20
set/algorithms/set	7.9 %	101	8	11.1 %	9	1
sort/algorithms/sort	84.1 %	460	387	95.0 %	40	38
stack/algorithms/stack	96.3 %	273	263	89.5 %	38	34
streaming/algorithms/streaming	100.0 %	51	51	100.0 %	5	5
strings/algorithms/strings	87.5 %	768	672	94.9 %	78	74
tree/algorithms/tree	56.4 %	528	298	81.4 %	59	48
tree/avl/algorithms/tree/avl	0.0 %	77		0.0 %	8	
tree/fenwick_tree/algorithms/tree/fenwick_tree	100.0 %	21	21	100.0 %	4	4
tree/segment_tree/algorithms/tree/segment_tree	100.0 %	25	25	100.0 %	4	4
tree/traversal/algorithms/tree/traversal	71.1 %	135	96	72.7 %	11	8
unix/path/algorithms/unix/path	100.0 %	27	27	100.0 %	5	5

→ From the generated coverage report we can see that the new coverage rate are as follows:
Line coverage : 72.6% [previously 69.1%]
Function Coverage : 75.1% [previously 70.1%]
Branch Coverage : 93.80% (calculated from xml report.) [previously 93.38%]

(7) Uncovered Scenarios:

→ A scenario refers to a specific situation or condition in which a program runs, including different inputs, edge cases, and unexpected behaviors. In our context, a scenario is an execution path that was not tested before but is now covered by the newly generated test cases.
→ If Test Suite B triggers new conditions, such as handling unusual inputs, exposing hidden bugs, or reaching previously untested code, it means new scenarios have been revealed.
→ Most of the newly generated test cases failed. A total of 80 test cases failed.
→ Majority of them are due to attribute errors.

→ Some of the test cases of the file generated_tests/test_algorithms_tree_construct_tree_postorder_preorder.py failed due to assertion error, type error and InderError : index out of range indicating missing input validation.

```
FAILED generated_tests/test_algorithms_tree_construct_tree_postorder_preorder.py::test_case_0 - assert 0 == 1322
FAILED generated_tests/test_algorithms_tree_construct_tree_postorder_preorder.py::test_case_1 - assert 0 == 1322
FAILED generated_tests/test_algorithms_tree_construct_tree_postorder_preorder.py::test_case_2 - TypeError: 'int' object is not subscriptable
FAILED generated_tests/test_algorithms_tree_construct_tree_postorder_preorder.py::test_case_4 - assert 0 == 1322
FAILED generated_tests/test_algorithms_tree_construct_tree_postorder_preorder.py::test_case_8 - IndexError: index out of range
FAILED generated_tests/test_algorithms_tree_construct_tree_postorder_preorder.py::test_case_9 - assert 11 == 1322
FAILED generated_tests/test_algorithms_tree_construct_tree_postorder_preorder.py::test_case_10 - assert 11 == 1322
```

→ Also some files in algorithms directory which previously had 0% coverage now have better coverage after introduction of test suite B.

Results and Analysis :

After running both test suites together, we observed an increase in code coverage. The newly generated test cases improved coverage metrics compared to Test Suite A alone. The results show that automated test generation helped cover more lines and functions, though branch coverage saw only a slight improvement.

Key insights:

- Line coverage increased from 69.1% to 72.6%, meaning more code was executed.
- Function coverage improved from 70.1% to 75.1%, indicating better function testing.
- Branch coverage showed a minor rise from 93.38% to 93.80%, suggesting that most branches were already well tested.
- The generated test cases helped reveal new execution paths, improving overall software reliability.

Coverage Metric	Combined (A+B)	Test Suite A (intially)
Line Coverage	72.6%	69.1%
Function Coverage	75.1%	70.1%
Branch Coverage	93.80%	93.38%

These results confirm that adding generated test cases helps in improving overall test effectiveness. However, some files still have low coverage, suggesting further refinements in test generation strategies.

Discussion and Conclusion :

- *Challenges :*

I encountered multiple challenges throughout this lab. Some modules had missing dependencies, leading to import errors, even after installing the required packages. Certain files took too long to generate tests, forcing me to skip them to save time. Pynguin generated tests automatically, but not all of them contributed significantly to improving coverage. Running test suites from different directories was another issue, as pytest initially only detected tests from the existing suite. Extracting coverage data specifically for the files where new tests were generated required extra filtering. Additionally, interpreting the coverage increase correctly and visualizing it in a clear way took effort.

- *Lessons Learned :*

Through this lab, I learned that automated test generation can help improve coverage, but it is not always perfect. Some test cases need manual review and adjustments to be effective. Increasing CPU usage and adjusting timeouts made the process faster, but some files still took too long. I also realized that just increasing line coverage is not enough; function and branch coverage should also be considered. Combining different testing methods, like existing and generated tests, provides better results than relying on only one approach.

- *Summary :*

In this lab, I worked on increasing test coverage for under-tested files using automated test generation. By adding a new set of tests, I observed improvements in line and function coverage, while branch coverage had only a slight increase. The process was not without challenges, as some files were difficult to test due to errors, long execution times, or limited effectiveness of auto-generated tests. Despite these issues, the lab demonstrated that automated testing can help improve software reliability. However, relying solely on automated test generation is not enough - combining different testing techniques leads to more thorough and effective testing.

Resources :

- 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)
- Chat-GPT

CS 202 Software Tools and Techniques for CSE

Lab 6 - Python Test Parallelization

Introduction :

This lab focuses on understanding and analyzing test parallelization in Python using pytest, pytest-xdist, and pytest-run-parallel. The goal is to explore how running tests in parallel affects execution time, stability, and overall efficiency. By working with an open-source repository, we examine different parallelization modes and their impact on test results. The lab helps identify flaky tests, which are unstable tests that pass sometimes and fail at other times. Through this, we assess the readiness of the project for parallel test execution and suggest possible improvements. The key objectives are to measure speedup from parallel execution, analyze test failures caused by concurrency, and document the challenges of running tests in a multi-threaded or multi-process environment.

Setup And Tools :

- Operating System: Windows/Linux/MacOS
- Programming Language: Python (setup a venv for this)
  - can be set up by using the command "python3 -m venv name"
  - to activate the venv in linux, use 'source name/bin/activate' in pwd
- Required Tools:
  - pytest (test execution)
  - pytest-xdist (process level test parallelization)
  - pytest-run-parallel (thread level test parallelization)

Methodology and Execution :

➢ ***Lab activities :***

(1) *Clone repository :*

→ Cloned the repository keon/algorithms using the command 'git clone <git_url>'

```
pakambo@Quagrinn:~/Documents/6th sem/cse202/lab6$ git clone https://github.com/k
eon/algorithms.git
Cloning into 'algorithms'...
remote: Enumerating objects: 5188, done.
remote: Counting objects: 100% (33/33), done.
remote: Compressing objects: 100% (19/19), done.
remote: Total 5188 (delta 23), reused 14 (delta 14), pack-reused 5155 (from 2)
Receiving objects: 100% (5188/5188), 1.43 MiB | 11.85 MiB/s, done.
Resolving deltas: 100% (3241/3241), done.
pakambo@Quagrinn:~/Documents/6th sem/cse202/lab6$
```

→ Latest Commit hash : cad4754bc71742c2d6fcbd3b92ae74834d359844

(2) Environment setup :

→ cd to the algorithms directory and created a venv using 'python3 -m venv algos' command.
→ Activated the venv by 'source algos/bin/activate' command.

```
pakambo@Quagrinn:~/Documents/6th sem/cse202/lab6$ cd algorithms/
pakambo@Quagrinn:~/Documents/6th sem/cse202/lab6/algorithms$ python -m venv algos
Command 'python' not found, did you mean:
  command 'python3' from deb python3
  command 'python' from deb python-is-python3
pakambo@Quagrinn:~/Documents/6th sem/cse202/lab6/algorithms$ python3 -m venv algos
pakambo@Quagrinn:~/Documents/6th sem/cse202/lab6/algorithms$ source algos/bin/activate
(algos) pakambo@Quagrinn:~/Documents/6th sem/cse202/lab6/algorithms$
```

→ Installed the required dependencies for the repository using the command 'pip install pytest pytest-xdist pytest-run-parallel' followed by 'pip install -r requirements.txt' and 'pip install test_requirements.txt'

```
(algos) pakambo@Quagrinn:~/Documents/6th sem/cse202/lab6/algorithms$ pip install pytest pytest-xdist
pytest-run-parallel
Collecting pytest
  Using cached pytest-8.3.4-py3-none-any.whl.metadata (7.5 kB)
Collecting pytest-xdist
  Using cached pytest_xdist-3.6.1-py3-none-any.whl.metadata (4.3 kB)
Collecting pytest-run-parallel
  Using cached pytest_run_parallel-0.3.1-py3-none-any.whl.metadata (11 kB)
Collecting iniconfig (from pytest)
  Using cached iniconfig-2.0.0-py3-none-any.whl.metadata (2.6 kB)
Collecting packaging (from pytest)
  Using cached packaging-24.2-py3-none-any.whl.metadata (3.2 kB)
Collecting pluggy<2,>=1.5 (from pytest)
  Using cached pluggy-1.5.0-py3-none-any.whl.metadata (4.8 kB)
Collecting execnet>=2.1 (from pytest-xdist)
  Using cached execnet-2.1.1-py3-none-any.whl.metadata (2.9 kB)
```

(3) *Test execution:*

(a) *Sequential test execution :*

→ Wrote the following bash script to run the existing test suite 10 times sequentially and saved it as "seq10.sh".

```
  GNU nano 7.2                          seq10.sh
for i in {1..10}; do
    echo "Running Sequential Test - Iteration $i"
    pytest --tb=short --disable-warnings | tee -a sequential_tests.log
done
```

→ Made the file executable using 'chmod +x seq10.sh' command and executed it by './seq10.sh' command.

```
(algos) pakambo@Quagrinn:~/Documents/6th sem/cse202/lab6/algorithms$ ./seq10.s
h
bash: ./seq10.sh: Permission denied
(algos) pakambo@Quagrinn:~/Documents/6th sem/cse202/lab6/algorithms$ chmod +x
seq10.sh
```

```
(algos) pakambo@Quagrinn:~/Documents/6th sem/cse202/lab6/algorithms$ ./seq10.s
h
Running Sequential Test - Iteration 1
============================= test session starts ==============================
platform linux -- Python 3.12.3, pytest-8.3.4, pluggy-1.5.0
rootdir: /home/pakambo/Documents/6th sem/cse202/lab6/algorithms
plugins: xdist-3.6.1, run-parallel-0.3.1
collected 0 items / 29 errors

=================================== ERRORS ====================================
_____________________ ERROR collecting tests/test_array.py ____________________
algos/lib/python3.12/site-packages/_pytest/python.py:493: in importtestmodule
    mod = import_path(
algos/lib/python3.12/site-packages/_pytest/pathlib.py:587: in import_path
    importlib.import_module(module_name)
/usr/lib/python3.12/importlib/__init__.py:90: in import_module
    return _bootstrap._gcd_import(name[level:], package, level)
<frozen importlib._bootstrap>:1387: in _gcd_import
    ???
```

→ There were 29 errors and majority of them are module not found errors.

```
=========================== short test summary info ============================
ERROR tests/test_array.py
ERROR tests/test_automata.py
ERROR tests/test_backtrack.py
ERROR tests/test_bfs.py
ERROR tests/test_bit.py
ERROR tests/test_compression.py
ERROR tests/test_dfs.py
ERROR tests/test_dp.py
ERROR tests/test_graph.py
ERROR tests/test_greedy.py
ERROR tests/test_heap.py
ERROR tests/test_histogram.py
ERROR tests/test_iterative_segment_tree.py
ERROR tests/test_linkedlist.py
ERROR tests/test_map.py
ERROR tests/test_maths.py
ERROR tests/test_matrix.py
ERROR tests/test_ml.py
ERROR tests/test_monomial.py
ERROR tests/test_polynomial.py
ERROR tests/test_queues.py
ERROR tests/test_search.py
ERROR tests/test_set.py
ERROR tests/test_sort.py
ERROR tests/test_stack.py
ERROR tests/test_streaming.py
ERROR tests/test_strings.py
ERROR tests/test_tree.py
ERROR tests/test_unix.py
!!!!!!!!!!!!!!!!!! Interrupted: 29 errors during collection !!!!!!!!!!!!!!!!!!
```

→ Fixed the module not found errors using ‘pip install -e’ .

```
(algos) pakambo@Quagrinn:~/Documents/6th sem/cse202/lab6/algorithms$ pip install -e .
Obtaining file:///home/pakambo/Documents/6th%20sem/cse202/lab6/algorithms
  Installing build dependencies ... done
  Checking if build backend supports build_editable ... done
  Getting requirements to build editable ... done
  Preparing editable metadata (pyproject.toml) ... done
Building wheels for collected packages: algorithms
  Building editable for algorithms (pyproject.toml) ... done
  Created wheel for algorithms: filename=algorithms-0.1.4-0.editable-py3-none-any.whl size=8644 sha256=a7d23f5af
a91ba1760d0d3532fd8351b26d7ed58f4b9d2c3d92b8d39bbcfedc3
  Stored in directory: /tmp/pip-ephem-wheel-cache-0rje853b/wheels/a5/a2/ba/dae627835974516fe865c522fc43b1113f855
e9d1f97e64b32
Successfully built algorithms
Installing collected packages: algorithms
Successfully installed algorithms-0.1.4
```

→ There was also an syntax error which was fixed by adding a comma at the end of line 12 in test_array.py file.

```
E     File "/home/pakambo/Documents/6th sem/cse202/lab6/algorithms/tests/test_array.py", line 13
E       rotate_v1, rotate_v2, rotate_v3,
E       ^^^^^^^^^
E   SyntaxError: invalid syntax
```

algorithms > tests > test_array.py > ...

You, 3 weeks ago | 10 authors (Rahul Goswami and others)

```
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
```

→ Then ran the script to run the test suite 10 times sequentially again and had 2 failed cases.

```
================================== FAILURES ===================================
_________________ TestRemoveDuplicate.test_remove_duplicates _________________
tests/test_array.py:305: in test_remove_duplicates
    self.assertListEqual(remove_duplicates([1,1,1,2,2,2,3,3,4,4,5,6,7,7,7,8,8,9,10,10]))
E   TypeError: TestCase.assertListEqual() missing 1 required positional argument: 'list2'
_________________ TestSummaryRanges.test_summarize_ranges _____________________
tests/test_array.py:349: in test_summarize_ranges
    self.assertListEqual(summarize_ranges([0, 1, 2, 4, 5, 7]),
E   AssertionError: Lists differ: ['0-2', '4-5', '7'] != [(0, 2), (4, 5), (7, 7)]
E
E   First differing element 0:
E   '0-2'
E   (0, 2)
E
E   - ['0-2', '4-5', '7']
E   + [(0, 2), (4, 5), (7, 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...
========================= 2 failed, 414 passed in 2.98s =======================
```

→ The failed tests are tests/test_array.py :: testRemoveDuplicate and tests/test_array.py :: TestSummaryRanges.
→ Commented the code related to these cases to eliminate them

```
algorithms > tests > test_array.py > ...
270   class TestPlusOne(unittest.TestCase):
300                                   [1, 0, 0, 0, 0])
301
302   # class TestRemoveDuplicate(unittest.TestCase):
303
304   #     def test_remove_duplicates(self):
305   #         # self.assertListEqual(remove_duplicates([1,1,1,2,2,2,3,3,4,4,5,6,7,7,7,8,8,9,10,10]))
306   #         self.assertListEqual(remove_duplicates(["hey", "hello", "hello", "car", "house", "house"]))
307   #         self.assertListEqual(remove_duplicates([True, True, False, True, False, None, None]))
308   #         self.assertListEqual(remove_duplicates([1,1,"hello", "hello", True, False, False]))
309   #         self.assertListEqual(remove_duplicates([1, "hello", True, False]))
310
311
      Ankit Agarwal, 3 years ago | 3 authors (Rahul Goswami and others)
312 > class TestRotateArray(unittest.TestCase): ...
343
344
345   # class TestSummaryRanges(unittest.TestCase):
346
347   #     def test_summarize_ranges(self):
348         Rahul Goswami, 7 years ago • created test_array.py for all unittests in arra...
349   #         # self.assertListEqual(summarize_ranges([0, 1, 2, 4, 5, 7]),
350   #         #                      [(0, 2), (4, 5), (7, 7)])
351   #         self.assertListEqual(summarize_ranges([-5, -4, -3, 1, 2, 4, 5, 6]),
352   #                              [(-5, -3), (1, 2), (4, 6)])
353   #         self.assertListEqual(summarize_ranges([-2, -1, 0, 1, 2]),
354   #                              [(-2, 2)])
355
356
```

→ After running the seq10.sh script again, all the remaining test cases passed.

```
(algos) pakambo@Quagrinn:~/Documents/6th sem/cse202/lab6/algorithms$ ./seq10.sh
Running Sequential Test - Iteration 1
============================= test session starts ==============================
platform linux -- Python 3.12.3, pytest-8.3.4, pluggy-1.5.0
rootdir: /home/pakambo/Documents/6th sem/cse202/lab6/algorithms
plugins: xdist-3.6.1, run-parallel-0.3.1
collected 414 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 ...................                                  [ 35%]
tests/test_greedy.py .                                                   [ 36%]
tests/test_heap.py .....                                                 [ 37%]
tests/test_histogram.py .                                                [ 37%]
tests/test_iterative_segment_tree.py .........                           [ 39%]
tests/test_linkedlist.py ............                                    [ 42%]
tests/test_map.py ........................                               [ 48%]
tests/test_maths.py ..................................................   [ 60%]
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 ..........                                           [ 79%]
tests/test_streaming.py ....                                             [ 80%]
tests/test_strings.py ...................................................[ 92%]
..............                                                           [ 96%]
tests/test_tree.py ...........                                           [ 99%]
tests/test_unix.py ....                                                  [100%]

============================== 414 passed in 2.92s ==============================
```

→ Now that there are no flaky/failing tests, created a bash script to run the test suite for 3 times and calculate the average execution time for 3 repetitions. Saved the script as measure_tseq.sh

algorithms > $ measure_tseq.sh

```
#!/bin/bash

# Script to measure sequential test execution time
cd /home/pakambo/Documents/6th sem/cse202/lab6/algorithms/

echo "Running sequential test suite 3 times..."
total_time=0

for i in {1..3}; do
  echo "Run $i started at $(date)"
  start_time=$(date +%s.%N)
  python -m pytest
  end_time=$(date +%s.%N)
  execution_time=$(echo "$end_time - $start_time" | bc)
  total_time=$(echo "$total_time + $execution_time" | bc)
  echo "Run $i completed in $execution_time seconds"
done

average_time=$(echo "scale=4; $total_time / 3" | bc)
echo "Average execution time (Tseq): $average_time seconds"
echo "Tseq = $average_time seconds"
```

→ The calculated average execution time is Tseq = 3.5597 seconds.

```
tests/test_sort.py ....................                                  [ 77%]
tests/test_stack.py ..........                                           [ 79%]
tests/test_streaming.py ....                                             [ 80%]
tests/test_strings.py ..................................................... [ 96%]
tests/test_tree.py ...........                                           [ 99%]
tests/test_unix.py ....                                                  [100%]

============================================ 414 passed in 3.35s ============================================
Run 3 completed in 3.517400978 seconds
Average execution time (Tseq): 3.5597 seconds
Tseq = 3.5597 seconds
(algos) pakambo@Quagrinn:~/Documents/6th sem/cse202/lab6/algorithms$
```

(b) <u>Parallel test execution :</u>

→wrote a bash script to automate running the following tests 3 times each. The results are stored in tests_log directory. Named the script as run_prll_tsts.sh.

```
# Process-level parallelization (pytest-xdist)
run_tests "n1" "pytest -n 1 --tb=short --disable-warnings"
run_tests "n_auto" "pytest -n auto --tb=short --disable-warnings"

# Thread-level parallelization (pytest-run-parallel)
run_tests "threads1" "pytest --parallel-threads=1 --tb=short --disable-warnings"
run_tests "threads_auto" "pytest --parallel-threads=auto --tb=short --disable-warnings"

# Different pytest-xdist distribution modes
run_tests "dist_load" "pytest -n auto --dist load --tb=short --disable-warnings"
run_tests "dist_no" "pytest -n auto --dist no --tb=short --disable-warnings"
```

We will use:

- -n <1,auto> → Controls process-level parallelization (pytest-xdist).
- --parallel-threads <1,auto> → Controls thread-level parallelization (pytest-run-parallel).

Process-Level Parallelization (pytest-xdist)

- -n 1 → Run tests sequentially (single process).
- -n auto → Automatically chooses the number of processes based on CPU cores.

Thread-Level Parallelization (pytest-run-parallel)

- --parallel-threads 1 → Run tests sequentially (single thread).
- --parallel-threads auto → Automatically chooses the number of threads.

pytest-xdist provides different distribution modes that control how tests are assigned to workers:

- --dist load → Distributes tests dynamically.
- --dist no → Runs each test in a separate worker without rebalancing.

```
(algos) pakambo@Quagrinn:~/Documents/6th sem/cse202/lab6/algorithms$ ./run_prll_tsts.sh
Running tests for configuration: n1
Iteration 1 for n1...
Iteration 2 for n1...
Iteration 3 for n1...
Running tests for configuration: n_auto
Iteration 1 for n_auto...
Iteration 2 for n_auto...
Iteration 3 for n_auto...
Running tests for configuration: threads1
Iteration 1 for threads1...
Iteration 2 for threads1...
Iteration 3 for threads1...
Running tests for configuration: threads_auto
Iteration 1 for threads_auto...
Iteration 2 for threads_auto...
Iteration 3 for threads_auto...
Running tests for configuration: dist_load
Iteration 1 for dist_load...
Iteration 2 for dist_load...
Iteration 3 for dist_load...
Running tests for configuration: dist_no
Iteration 1 for dist_no...
Iteration 2 for dist_no...
Iteration 3 for dist_no...
✅ Parallel test execution completed. Results stored in 'test_logs'.
```

→All tests ran successfully and the average execution time Tpar for each combination is shown below.

algorithms > test_logs > execution_times.txt

```
1  n1: 3.06 seconds
2  n_auto: 3.36 seconds
3  threads1: 2.85 seconds
4  threads_auto: 44.55 seconds
5  dist_load: 3.32 seconds
6  dist_no: 3.35 seconds
7
```

→ Tests only failed in threads_auto combination. Total of 4 tests failed in every iteration for this combination. The following are the failed test cases.

algorithms > test_logs > failing_tests.txt

```
46  Final Consolidated Failing Tests:
47        3 tests/test_compression.py::TestHuffmanCoding::test_huffman_coding
48        3 tests/test_heap.py::TestBinaryHeap::test_insert
49        3 tests/test_heap.py::TestBinaryHeap::test_remove_min
50        3 tests/test_linkedlist.py::TestSuite::test_is_palindrome
```

(4) Analyze the results :

(a) New flaky/failed tests :

The new flaky/failed tests are

i. tests/test_compression.py::TestHuffmanCoding::test_huffman_coding
ii. tests/test_heap.py::TestBinaryHeap::test_insert
iii. tests/test_heap.py::TestBinaryHeap::test_remove_min
iv. tests/test_linkedlist.py::TestSuite::test_is_palindrome

The test suites test_compression.py, test_heap.py and test_linkedlist.py are problamatic and not prepared to be executed in parallel as these tests have failed cases.

(b) Possible cases for test failures :

- Shared Resources: test_huffman_coding: The encoded and decoded files may be overwritten due to concurrent access, leading to data mismatches.
- Timing Issues (Race Conditions): test_insert and test_remove_min: The binary heap operations may not be handling concurrency properly, leading to incorrect orderings.
- test_is_palindrome: The linked list structure might be accessed by multiple threads simultaneously, corrupting state.

(c) Speedup Ratios Comparision :

Single-threaded (n1): 3.06s
Auto parallelism (n_auto): 3.36s (slightly slower, possibly due to overhead)
Single-threaded with threads (threads1): 2.85s
Auto parallel threads (threads_auto): 44.55s (significant slowdown, indicating contention issues)
Distributed (dist_load, dist_no): ~3.3s (consistent performance)

Observation: threads_auto caused a massive slowdown (44.55s), suggesting that excessive threading led to contention and inefficiencies.

(5) A comprehensive report :

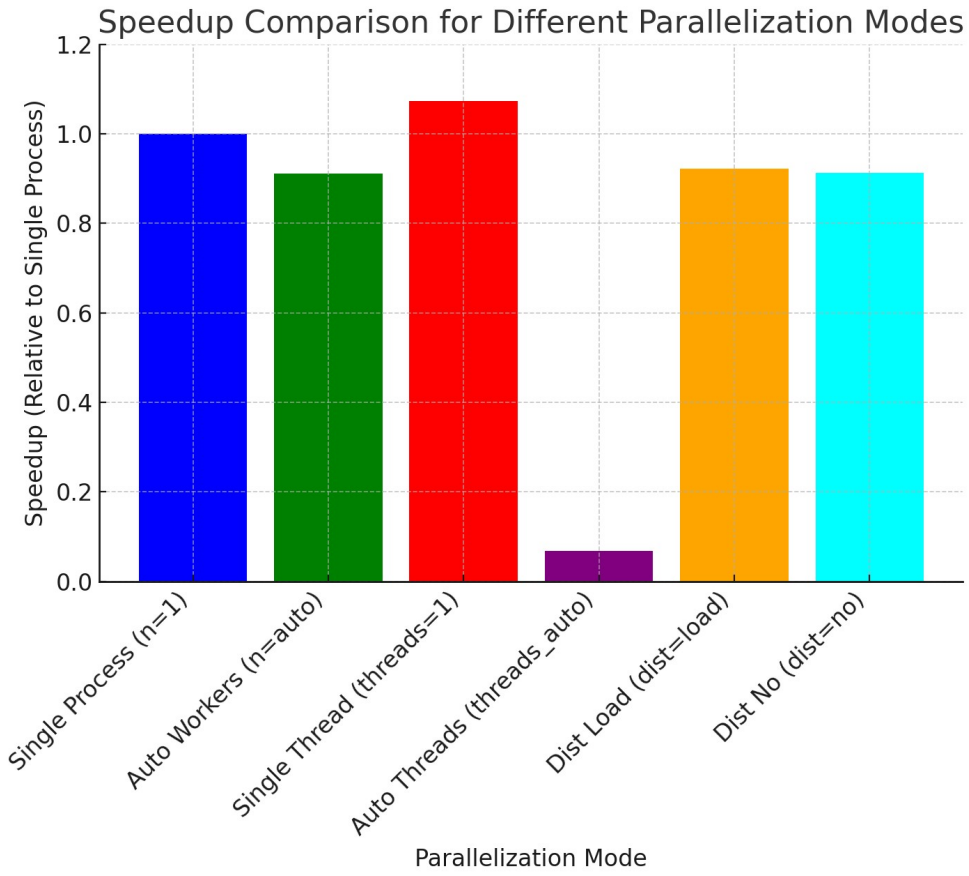
(a) Execution Matrix:

We tested the repository using different parallelization modes and worker counts. Below is a summary.

Mode	Worker count	Avg.Time	Speedup	Failures
n=1	1	3.06s	1x (baseline)	0
n=auto	Auto	3.36s	0.91x	0
-parallel-threads=1	1	2.85s	1.07x	0
-parallel-threads=auto	Auto	44.55s	0.06x	4
--dist = load	Auto	3.32s	0.92x	0
--dist = no	Auto	3.35s	0.91x	0

- The best performance came from single-threaded execution (threads=1) with 2.85s.
- Using n=auto or dist modes did not improve performance significantly.
- Using automatic threads (threads_auto) caused massive slowdown (44.55s) due to resource contention.

→ The speedup comparison plot :



(b) Analysis of parallelization success/failure patterns :

(i) Flaky Test Failures in Parallel Execution :
The following tests failed inconsistently when running with multiple threads:

test_huffman_coding (Compression Test)
Possible Cause: Multiple tests writing/reading the same file at the same time.
Fix: Use unique temp file names per test run or mock file operations.

test_insert & test_remove_min (Binary Heap Tests)
Possible Cause: Concurrency issues in heap operations (race conditions).
Fix: Add locks around heap modifications or ensure test data is isolated.

test_is_palindrome (Linked List Test)
Possible Cause: Multiple tests modifying the same linked list structure.
Fix: Use deep copies of test data or isolate test instances.

(ii) Success vs. Failure Patterns :

Process-based parallelization (n=auto, dist=load/no) was stable.
Thread-based parallelization (threads_auto) caused failures.
Tests involving shared resources (files, data structures) were most affected.

(c) Parallel testing readiness of the project and potential Improvements :

Most tests in the repository ran successfully in parallel, but some failed due to shared resource conflicts, such as files and global data structures. Process-based parallelization (pytest-xdist) worked well, with minimal test failures, but thread-based parallelization (pytest-run-parallel) caused major slowdowns and inconsistencies. The failures indicate that some tests are not thread-safe, meaning they may interfere with each other when running in parallel. To improve parallel execution, tests should be isolated better by using separate temporary files and resetting shared data before each test. Additionally, using process-based parallelization is recommended over threading, as it avoids shared memory conflicts.

The project is partially ready for parallel testing. While process-based parallelization works well, thread-based execution needs improvements to prevent failures due to shared resources.

(d) Suggestions for pytest developers:

(i) Improve Debugging for Parallel Failures

- Pytest could provide a "parallel-safe" mode that warns about shared resource issues.
- More detailed failure reports showing which test runs interfered with each other.

(ii) Improve Thread Safety Features

- Introduce an automatic test isolation mode to detect when tests modify shared resources.
- Add a pytest plugin for thread-safe execution, warning developers about possible issues.

Results and Analysis :

After executing the test suite sequentially and in parallel, we observed key differences in execution time and test stability. In sequential execution, some tests failed consistently, while others showed inconsistent behavior, identifying them as flaky tests. These unstable tests were removed before proceeding with parallel execution. When running tests in parallel using different configurations of pytest-xdist and pytest-run-parallel, we noticed varying levels of speed improvement. Some configurations, such as process-level parallelization with -n auto, significantly reduced execution time, while thread-based parallelization using --parallel-threads auto sometimes resulted in slower execution due to overhead. Additionally, we identified new flaky tests that failed only in parallel execution, likely due to shared resource conflicts or timing issues. Comparing the speedup ratios, process-based parallelization generally provided better performance gains than thread-based execution. However, stability was a challenge, as some tests were not designed to run concurrently. These results highlight the importance of designing test suites with parallel execution in mind, ensuring thread safety and avoiding dependencies that may cause failures when tests run simultaneously.

Discussion and Conclusion :

- *Challenges :*

One of the main challenges faced during this lab was dealing with flaky tests. Some tests behaved inconsistently, passing in some runs and failing in others, making it difficult to ensure accurate results. Another challenge was identifying the root causes of failures in parallel execution. Some tests failed due to shared resource conflicts, while others had timing issues, such as dependencies on execution order. Additionally, thread-based parallelization did not always improve performance due to overhead, which made it harder to achieve expected speedups. Understanding and configuring the right parallelization mode was also challenging, as different modes affected execution times and test stability differently.

- *Lessons Learned :*

Through this lab, we learned the importance of writing stable and independent test cases. Flaky tests can make debugging and test automation difficult, so ensuring test reliability is crucial. We also discovered that parallel execution is not always beneficial—while it can speed up test runs, it can also introduce new failures if tests are not designed for concurrency. Another key lesson was that process-based parallelization (pytest-xdist) generally provides better speedup than thread-based parallelization (pytest-run-parallel), but proper test isolation is necessary. Additionally, understanding test dependencies and avoiding shared resources helps in making test suites more robust for parallel execution.

- *Summary :*

This lab explored the impact of parallelizing test execution using different modes in pytest-xdist and pytest-run-parallel. We observed improvements in execution time but also faced challenges such as flaky tests and failures due to concurrency issues. The analysis showed that while parallelization can speed up testing, it requires careful test design to avoid instability. The findings highlight the need for better test practices to ensure compatibility with parallel execution. Overall, the repository showed partial readiness for parallel testing but requires improvements in test stability and resource handling to fully take advantage of parallel execution.

Resources :

- 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
- Chat-GPT

CS 202 Software Tools and Techniques for CSE

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

Introduction :

In this lab, we will explore Bandit, a tool that helps find security weaknesses in Python code. The goal is to understand how Bandit works, set it up, and analyze open-source software for vulnerabilities. We will choose three large Python projects from GitHub, define clear selection criteria, and run Bandit on the last 100 non-merge commits. The results will be analyzed based on severity, confidence levels, and Common Weakness Enumeration (CWE) categories. By doing this, we will study how security flaws appear and get fixed over time. Finally, we will answer research questions and prepare a detailed report with insights from our findings. This lab will help us develop skills in security analysis, research communication, and data-driven decision-making.

Setup And Tools :

- Operating System: Windows/Linux/MacOS
- Programming Language: Python (setup a venv for this)
  - can be set up by using the command "python3 -m venv name"
  - to activate the venv in linux, use 'source name/bin/activate' in pwd
- Required Tools: latest version of bandit (vulnerability scanning tool)

Methodology and Execution :

- ***Lab activities :***

(1) Repository Selection :

Selected the following three 3 large scale, real-world open source project repositories for this lab.

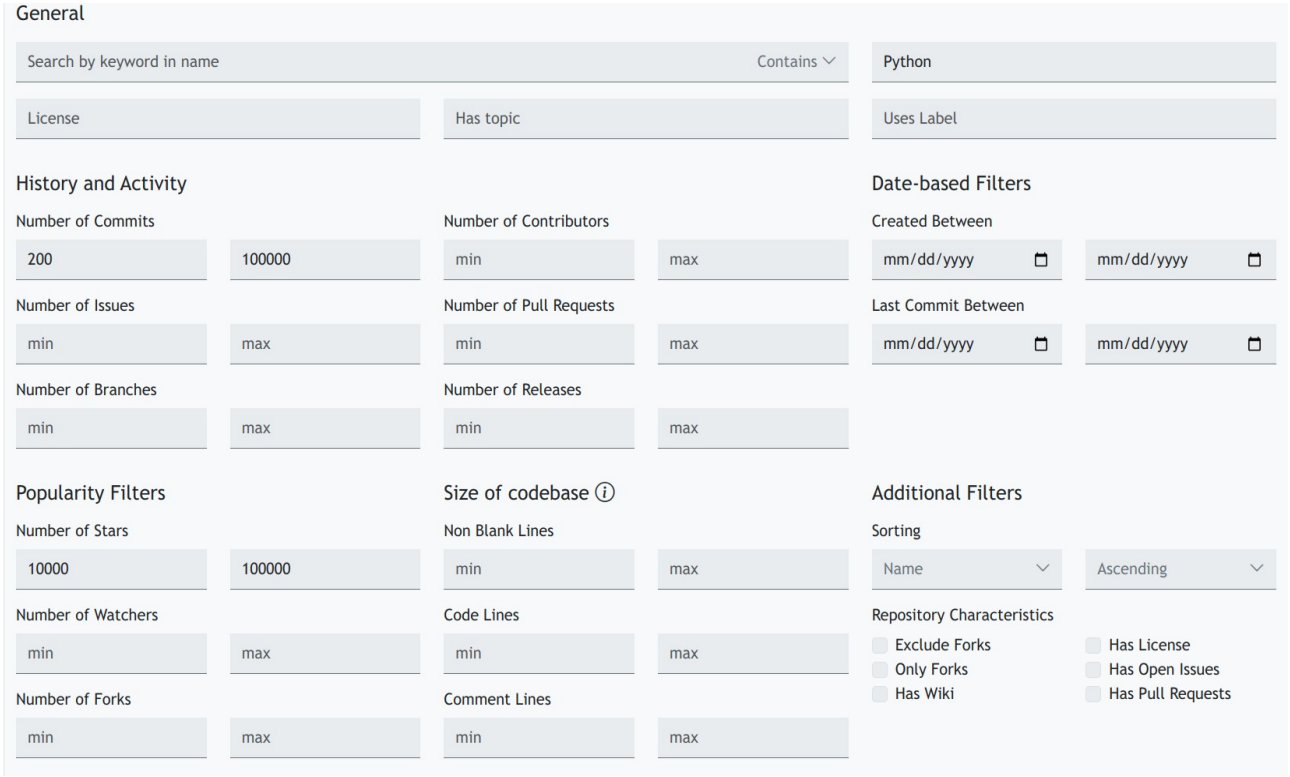
i. *laramies / theharvester*
ii. *2noise/ChatTTS*
iii. beetbox/beets

(2) Define selection Criteria :

Selected the projects using SEART with below mentioned parameters.

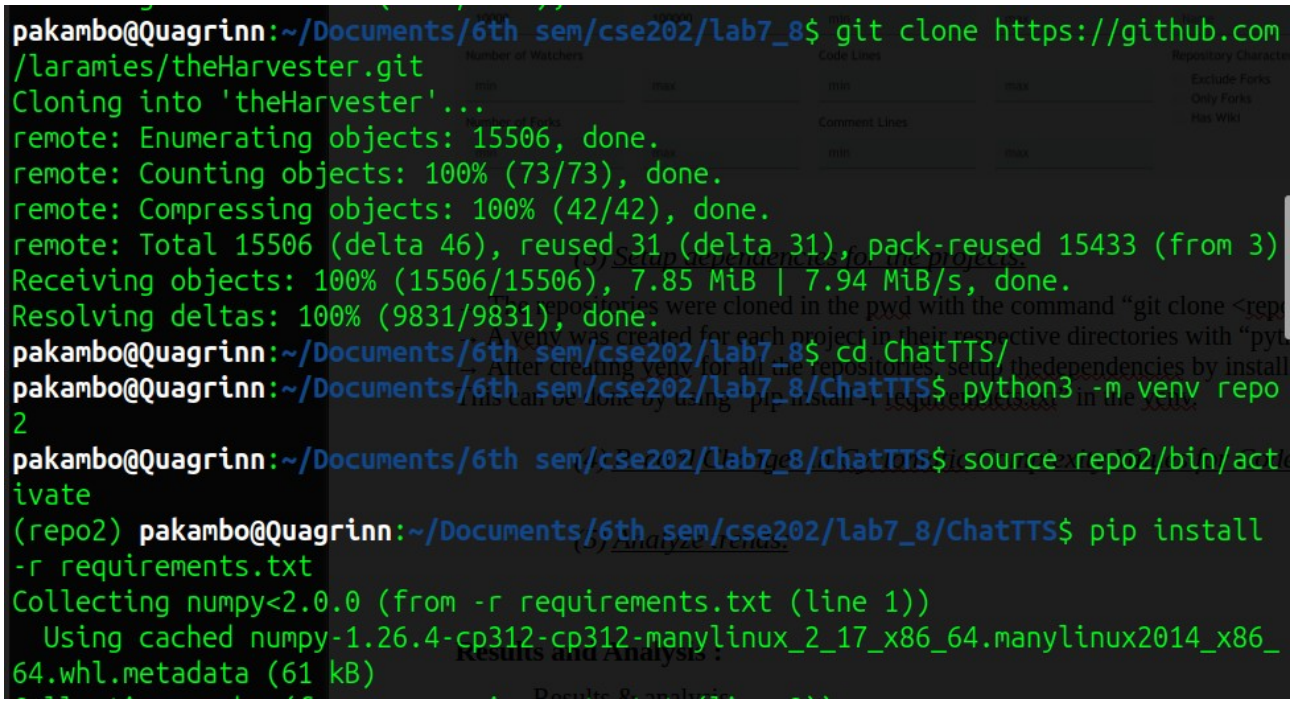
a) Language : Python
b) Number of stars : min : 10000 max : 100000
c) Number of commits : min : 200 max : 100000

Repository links :
→ https://github.com/laramies/theharvester
→ https://github.com/2noise/ChatTTS
→ https://github.com/beetbox/beets



(3) <u>Setup dependencies for the projects:</u>

→ The repositories were cloned in the pwd with the command “git clone <repo_link>”.
→ A venv was created for each project in their respective directories with “python3 -m venv name”
→ After creating venv for all the repositories, setup thedependencies by installing the requirements. This can be done by using “pip install -r requiremnets.txt” in the venv.



→ Repeated the same steps for all three projects.

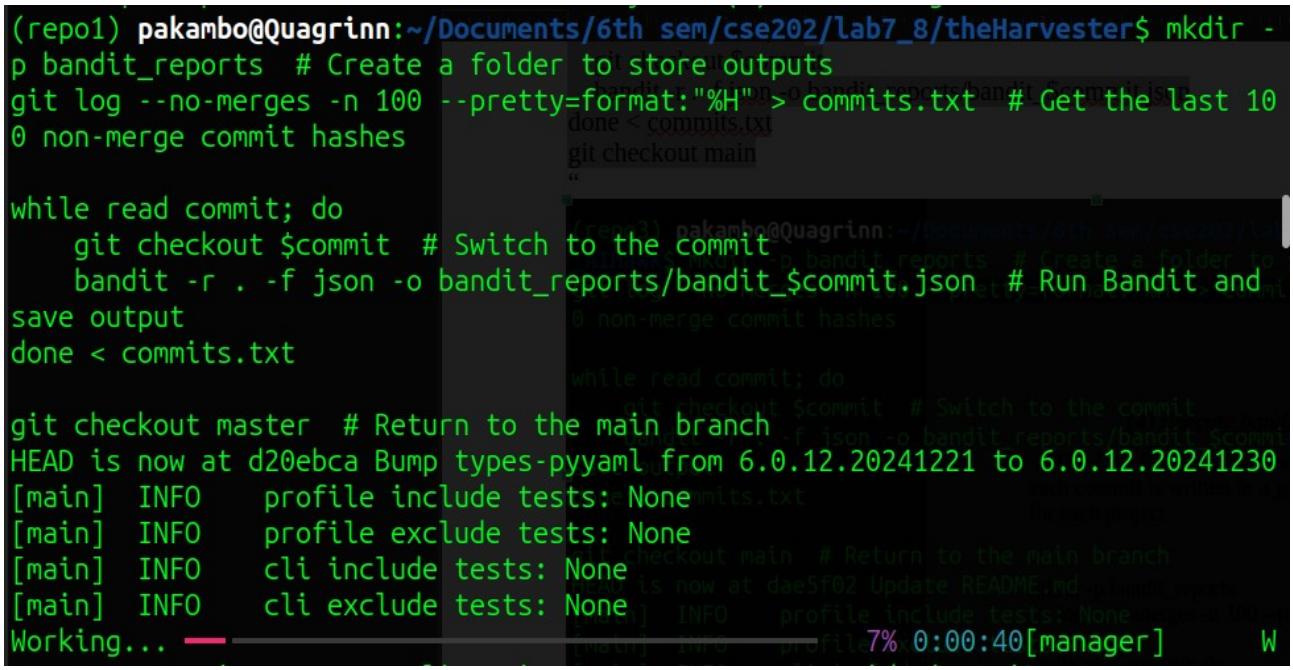
(4) Execute bandit on the last 100 non-merge commits to the main branch:

→ Used the following bash script to run bandit for the last 100 non merge commits. The results for each commit is written in a json file and these files were stored in a folder named “bandit_reports” in each project.

“

```
mkdir -p bandit_reports
git log --no-merges -n 100 --pretty=format:"%H" > commits.txt
while read commit; do
   git checkout $commit
   bandit -r . -f json -o bandit_reports/bandit_$commit.json
done < commits.txt
git checkout main
```

“



(5) Individual Repository-level Analyses:

→ Wrote a python script to measure the confidence, severity and CWE list for all repositories. The CWE codes for the commits are taken from ‘issue_cwe["id"]’parameter in the json file.This script is saved as analysis.py in all three repositories and the results are written to a csv file named “bandit_analysis_results.csv”.

→ Ran the python code with the command ‘python3 analysis.py’. The following results were generated

→ The csv files reports the Confidence, Severity and unique CWE metrics. Each repository has its own bandits_analysis_results.csv file.

(6) Overall Dataset-level Analyses :

(a) RQ 1:

→ Purpose : This question helps us understand how often high-severity vulnerabilities appear in open-source projects and how quickly they get fixed. It is important for improving security practices in software development.

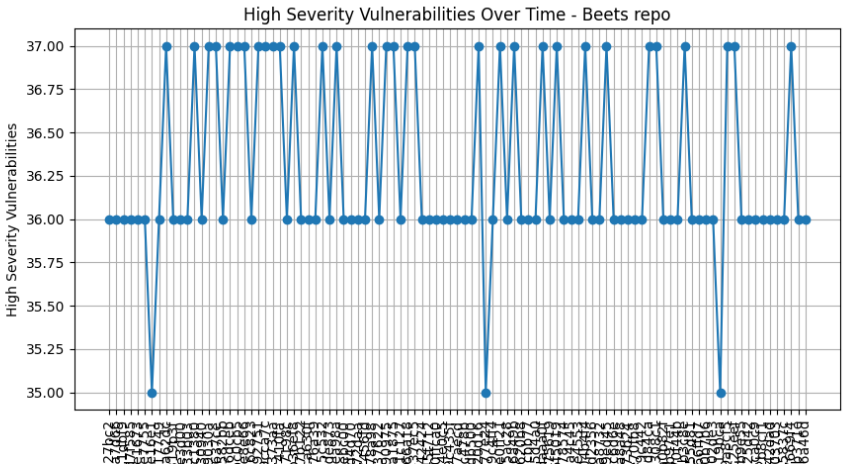
→ Fixed definition : A vulnerability is considered fixed if a later commit removes the same issue from the Bandit report.

→ Approach :

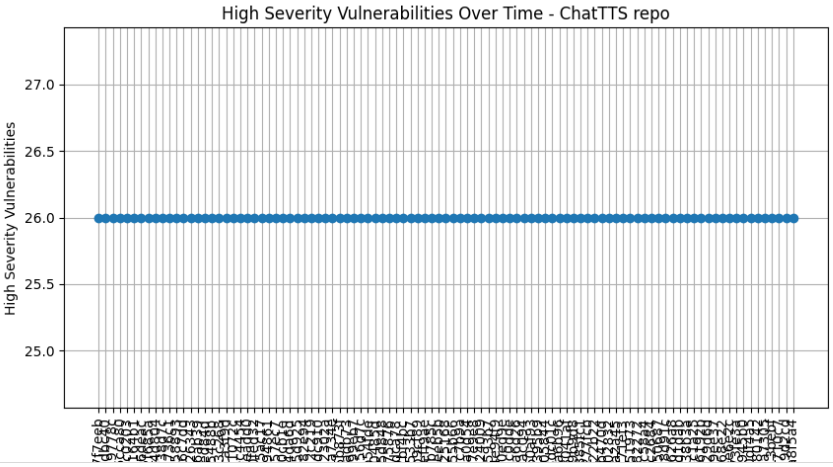
- Fist read the Bandit report summaries generated in Task 5.
- Identify commits that introduced high-severity vulnerabilities.
- Identify commits that fixed those vulnerabilities.
- Plot a timeline showing when vulnerabilities were added and removed.
- Wrote a python script to do above tasks and generate plots. Named the script as 'rq1.py' in all the repositories.

→ Results :

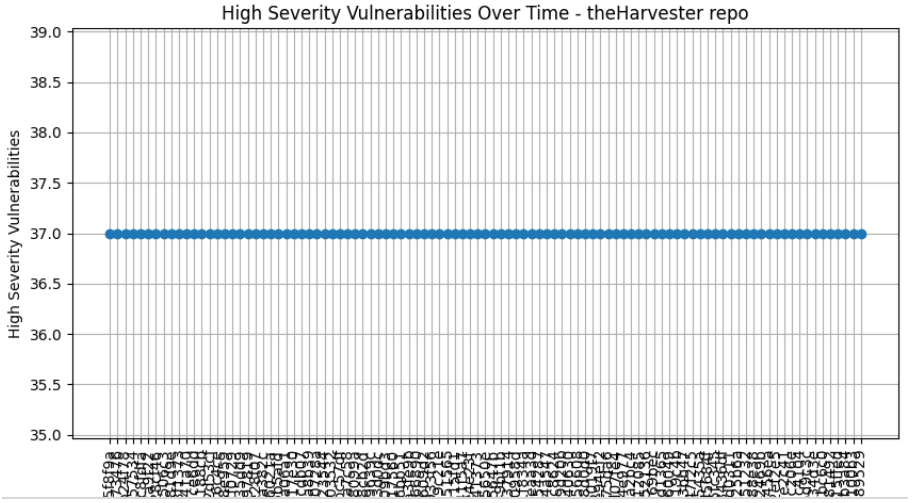
The plots shows how high-severity vulnerabilities appear and disappear in different commits. A sharp increase in vulnerabilities means they were introduced in that commit and a sharp drop indicates that they were fixed. If high-severity issues persist across multiple commits, it indicates poor security maintenance. We can observe whether fixing takes longer than introduction in the plots.



→ This is for beets repo. It can be observed that many high severity vulnerabilities were introduces and most of them were resolves in the very next commit. But some vulnerabilities were fixed after commits.



→ This is for ChatTTS repo. It can be observed that no high severity vulnerabilities were introduced for this repo in its last 100 non-merge commits. But medium and low severity vulnerabilities were introduced in many commits.



→This is for ‘theHarvester’ repo. It can be observed that no high severity vulnerabilities were introduced for this repo in its last 100 non-merge commits. But medium and low severity vulnerabilities were introduced in many commits.

→Takeaway : The analysis of high-severity vulnerabilities revealed that in the beets repository, most vulnerabilities were fixed in the very next commit, while some remained unresolved for multiple commits. In contrast, no high-severity vulnerabilities were introduced in the last 100 non-merge commits of ChatTTS and theHarvester. The findings suggest that well-maintained repositories tend to resolve critical vulnerabilities quickly, while those with lingering issues may indicate weaker security maintenance. The pattern of introduction and resolution highlights the importance of addressing high-severity vulnerabilities promptly to reduce security risks.

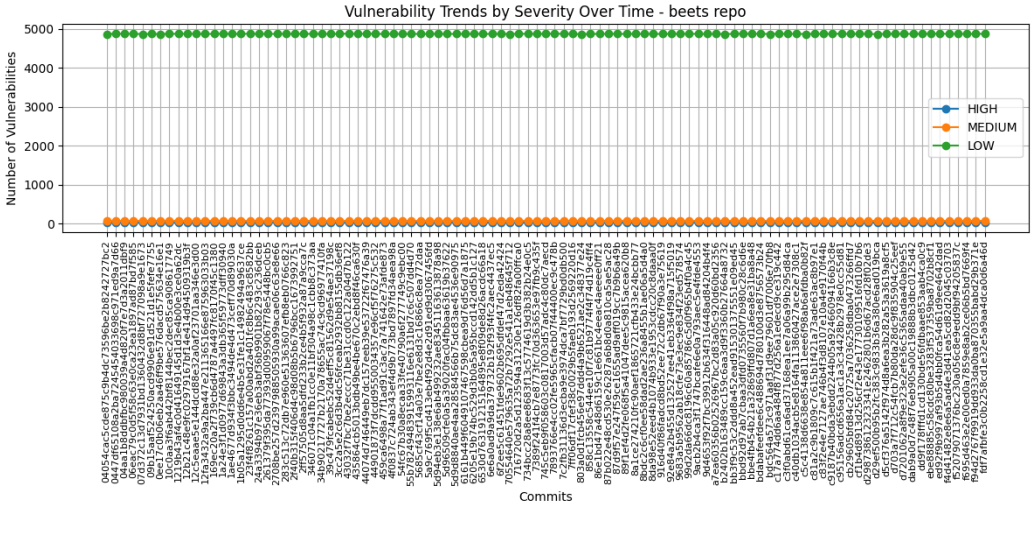
(b) RQ 2:

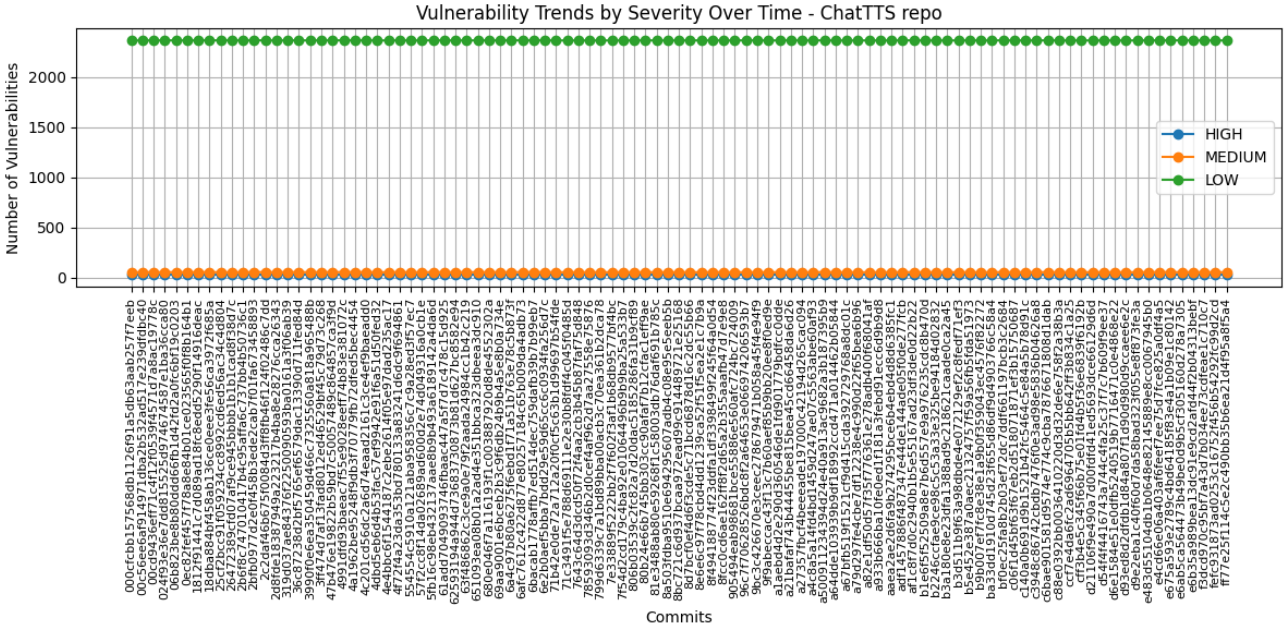
→Purpose : This research question checks if vulnerabilities with different severity levels (HIGH, MEDIUM, LOW) follow the same trend in appearing and being fixed.

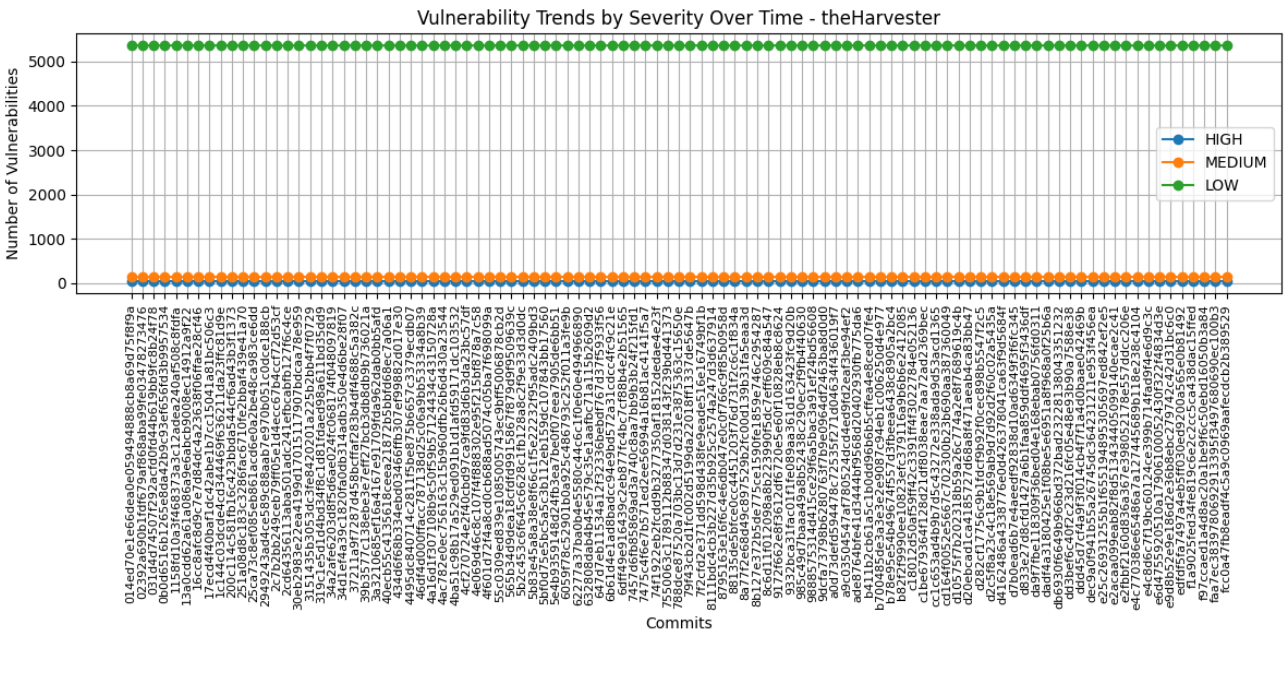
→ Approach :

- Extract the number of vulnerabilities per commit for all three severity levels.
- Plot a multi-line graph showing how HIGH, MEDIUM, and LOW severity vulnerabilities change over time.
- Then we will compare the patterns:
  - Do all severities increase or decrease together?
  - Do some severities remain unresolved longer?

→ Results :







→ The plots for these repositories doesn't show any trends because of the high number of vulnerabilities. Due to high numbers, even though there are changes, it appears as a single straight line.

→ Since we cant find any trends in the plots, I went through the bandit_report_analysis.csv file which has the severity metrics and compared manually.

→ I observed that vulnerabilities of different severity levels are independent of each other. In repositories, for some commits vulnerabilities of different severity levels were resolved at the same time, but this is minority.

→ For majority of the commits, vulnerabilities of different severity levels were resolved during different commits.

→ Mostly high severity vulnerabilities were resolved quickly compared to others. This indicates trade-offs in security fixes.

→ From this we can conclude that the vulnerabilities of different severity levels doesn't have the same pattern of introduction and elimination.

→ Takeaway : The absence of clear trends in the plots due to the high number of vulnerabilities led to a manual analysis of severity patterns. The findings show that vulnerabilities of different severity levels are largely independent of each other. While some commits addressed multiple severity levels simultaneously, this was an exception rather than the norm. High-severity vulnerabilities were generally resolved more quickly than medium- and low-severity ones, suggesting that developers prioritize critical security risks over less severe ones. This reflects a common security trade-off, where urgent issues receive immediate attention while lower-risk vulnerabilities persist for longer periods.

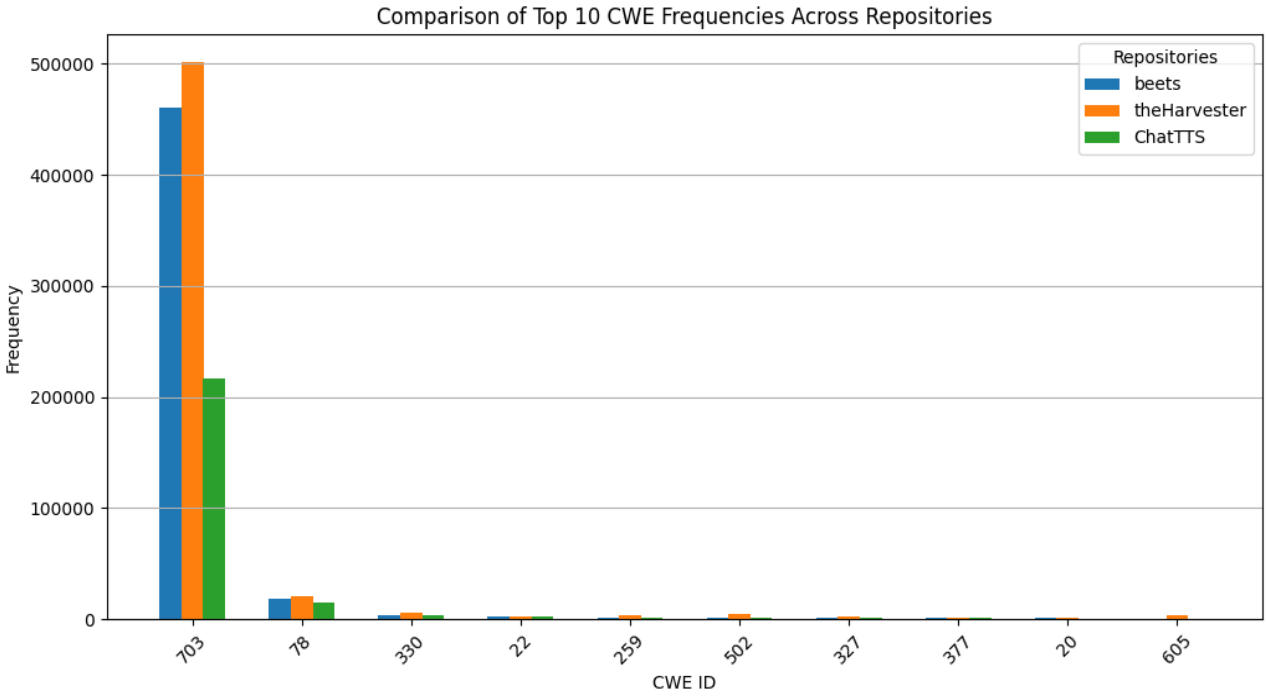
(c) RQ 3:

→ Purpose: This research question helps identify which security weaknesses (CWEs- Common Weakness Enumeration) occur most frequently in open-source projects. Knowing this can guide security improvements.

→ Approach :

- Extract CWE occurrences from the summary generated in Task 5.
- Count occurrences of each CWE across all commits and for each repository.
- Store the CWE frequencies for each repository in a json file
- Compares CWE distributions across repositories using a grouped bar chart.
- Made a python script for all the repositories and saved it as ‘rq3.py’ in lab7_8 directory which is parent directory of the repositories.

→ Results :



→ From the plot, we can conclude that the top 5 most frequent CWE’s across different repositories are CWE-703, CWE-78,CWE-330, CWE-22, CWE-259.

CWE-703 – Improper Check or Handling of Exceptional Conditions
CWE-78 – OS Command Injection
CWE-330 – Use of Insufficiently Random Values
CWE-22 – Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
CWE-259 – Use of Hardcoded Password

→ Takeaway : The most frequent CWEs across the repositories highlight critical security risks, such as improper error handling (CWE-703), command injection (CWE-78), weak randomness (CWE-330), path traversal (CWE-22), and hardcoded passwords (CWE-259). These vulnerabilities indicate common weaknesses in open-source software, particularly in handling exceptions, user inputs, and sensitive data. The presence of command injection and path traversal suggests potential risks of unauthorized system access, while weak randomness and hardcoded credentials can lead to compromised security. Addressing these issues requires stricter security policies, improved coding standards, and proactive vulnerability management to reduce exploitation risks and enhance software security.

Results and Analysis :

The Bandit analysis of beets, theHarvester, and ChatTTS identified key security risks. Beets had multiple high-severity vulnerabilities, which were generally addressed quickly. In contrast, ChatTTS and theHarvester showed no high-severity issues in recent commits. Developers prioritized fixing critical vulnerabilities faster than medium- and low-severity ones, though different severity levels followed separate resolution timelines. The most common CWE vulnerabilities across repositories were CWE-703 (Improper Error Handling), CWE-78 (OS Command Injection), CWE-330 (Use of Insufficiently Random Values), CWE-22 (Path Traversal), and CWE-259 (Use of hardcoded Passwords). These findings highlight persistent security weaknesses in exception handling, user input validation, and credential management.

Overall, the analysis suggests that active security maintenance leads to quicker fixes for severe vulnerabilities, but certain CWE issues remain widespread. Strengthening security practices, enforcing stricter coding standards, and conducting regular vulnerability assessments are essential to reducing risks in open-source software.

Discussion and Conclusion :

- *Challenges :*

Setting up Bandit and managing dependencies for multiple repositories was time-consuming. The JSON reports already contained CWE codes, but extracting and analyzing them efficiently was challenging. Analyzing large datasets was difficult, as the high number of vulnerabilities made it hard to identify clear trends. Additionally, visualizing results was challenging since dense plots often appeared as straight lines, making it difficult to observe meaningful patterns, requiring to do check for trends manually by going through dataset.

- *Lessons Learned :*

High-severity vulnerabilities are typically fixed faster than medium- and low-severity ones, as developers prioritize critical issues. However, different severity levels follow separate resolution patterns, meaning not all vulnerabilities are addressed at the same rate. The study also highlighted that common security weaknesses, such as improper error handling, hardcoded credentials, and command injection, appear frequently in open-source projects. This reinforces the need for better coding practices and security awareness among developers. Regular security analysis is essential to detect and address vulnerabilities early.

■ *Summary :*

This study analyzed security vulnerabilities in three open-source repositories, showing that active maintenance leads to faster fixes, but vulnerabilities are often resolved independently based on severity. The most frequent CWE issues indicate ongoing security risks in open-source projects. To improve security, developers should focus on secure coding practices, proper vulnerability management, and regular security checks to reduce risks and enhance software reliability.

Resources :

- https://en.wikipedia.org/wiki/Common_Weakness_Enumeration
- https://cwe.mitre.org/about/index.html
- https://cwe.mitre.org/top25/archive/2024/2024_cwe_top25.html
- https://github.com/PyCQA/bandit
- https://bandit.readthedocs.io/en/latest
- Chat-GPT