Lab Assignment 5

Lab Topic: Code Coverage Analysis and Test Generation

1. Introduction

The purpose of this lab is to gain a deeper understanding of **code coverage** and how to ensure that the tests we write cover as much of the codebase as possible. Code coverage is an important concept in software development because it helps us evaluate how thoroughly our code is tested and whether there are any untested parts of your application.

Environment Setup:

- **Operating System**: Windows
- **SET-IITGN-VM**: Used for consistent setup across environments.

Tools and Versions:

- **Git:** Version [2.25.1]
- **Python:** Version [3.8.10]
- **Pytest:** Version [7.4.4]
- **pytest-cov**
- **pytest-func-cov**:
- **Coverage:** Version [7.7.0] with C extension
- **LCOV:** Version [1.14]
- **GenHTML:** Version [1.14] (part of LCOV)
- **Pynguin:** Version [0.34.0]

Installation and Configuration:

1. **pip install pytest pytest-cov pytest-func-cov coverage pynguin**
2. pip install -e .
3. python3 -c "import algorithms"

Key Concepts:

Code coverage: The percentage of code that is executed while the tests are run. It provides an indication of which parts of the codebase are tested and which are not.

Line coverage: Tracks the percentage of executed lines in the code.

Branch coverage: Ensures all possible branches (if-else conditions) are tested.

Function coverage: Checks how many functions are executed at least once during tests.

---

2. Methodology and Execution

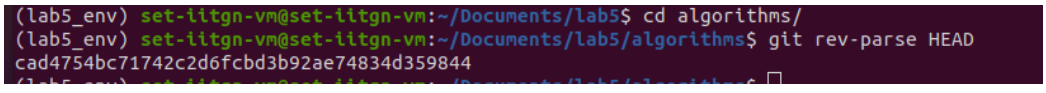
File Structure

```
lab78/
├── algorithms/                # Source code for algorithms
├── tests/                     # Test Suites
│   ├── TestSuiteA/            # Contains test cases for part A
│   ├── TestSuiteB/            # Contains test cases for part B
│   ├── TestSuiteAB/           # Combined test cases from A and B
│
├── coverage_reports/                # Test coverage reports
│   ├── coverage_TestSuiteAB_branch.info # Branch coverage info for combined tests
│   ├── coverage_TestSuiteB_branch.info  # Branch coverage info for TestSuiteB
│   ├── coverage_TestSuiteB_line.info    # Line coverage info for TestSuiteB
│   ├── coverage.xml                 # XML report of overall test coverage
│
├── reports/                   # Low coverage analysis
│   ├── low_branch_coverage_report.py    # Script to analyze low branch coverage
│   ├── low_line_coverage_report.py      # Script to analyze low line coverage
│   ├── low_branch_0_1_coverage_files.txt  # Files with 0-1% branch coverage
│   ├── low_line_0_1_coverage_files.txt    # Files with 0-1% line coverage
│
├── docs/                      # Documentation
│   ├── README.md                  # Project description and setup guide
│   ├── CONTRIBUTING.md            # Contribution guidelines
│   ├── CODE_OF_CONDUCT.md             # Community guidelines
│
├── scripts/                   # Utility scripts
│   ├── run_generate_tests.py      # Runs test generation script
│   ├── filter_files.py            # Filters files based on coverage
│
├── setup.py                   # Installation script for the project
├── requirements.txt           # Dependencies needed to run the project
├── pytest.ini                 # Configuration for pytest
├── tox.ini                    # Configuration for testing with Tox
```

Note: The commands in the report are not according to the above structure. They are run with all the files located in the outer algorithms file.This is for easier representation purposes only.

Step 1: Set up

Clone the keon/algorithms repository, create a virtual environment, and install the test requirements along with any additional dependencies listed in the introduction.



Latest hash: Cad4754bc71742c2d6fcbd3b92ae74834d359844

Step 2: Test Suite A

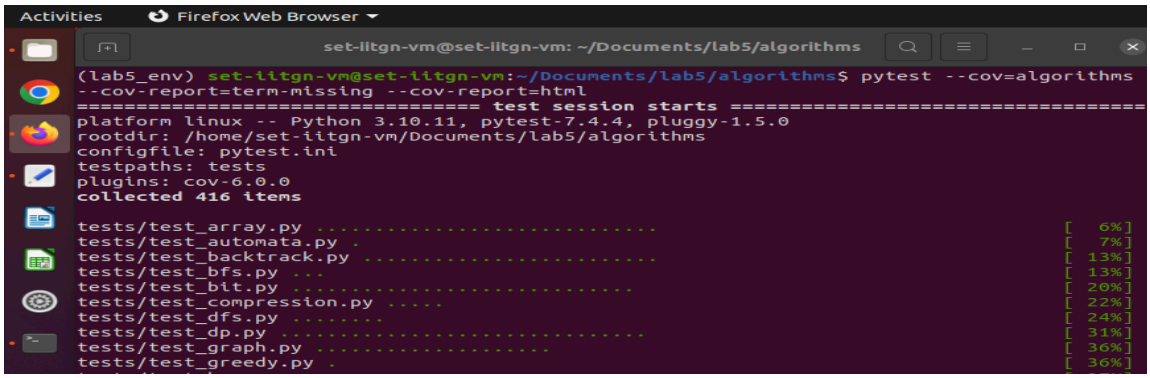
Configuration:
First, I created my pytest.ini file which is a configuration file for pytest that helps customize and manage test settings. It is used to set default options (e.g., addopts = --cov=algorithms --cov-branch to always enable coverage) and define test paths (e.g., testpaths = TestSuiteA to specify where tests are located).

```
[pytest]
testpaths = TestSuiteA
addopts = --cov=algorithms --cov-report=term-missing --cov-report=html
pythonpath = .
```

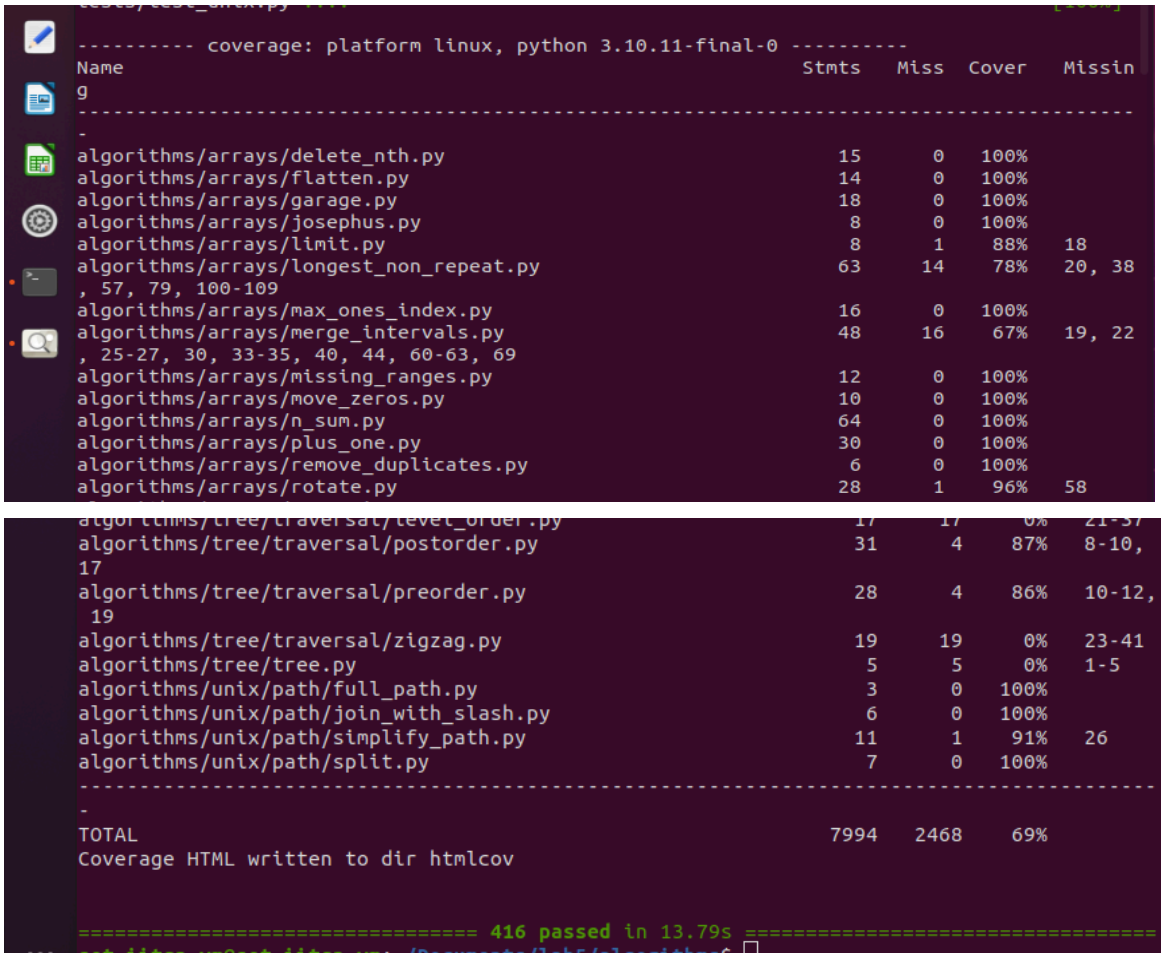
Procedure:
For Test Suite A, I used the command below which uses **pytest** to Runs all test cases in the current directory.
--cov=algorithms to Measures code coverage for the algorithms module.
--cov-report=term-missing to Prints a coverage summary in the terminal and highlights which lines were not executed.
--cov-report=html to Generate a detailed HTML report (saved in htmlcov/index.html).
--cov-branch to Measures branch coverage i.e It tracks both executed and missed branches (e.g., if statements with multiple possible paths).

1. **Line coverage: pytest --cov=algorithms --cov-report=html:html_line_coverage TestSuiteA/**

The given tests are all passing and the total coverage is **69%**. In the below image '.' represents 'Test Passed'.

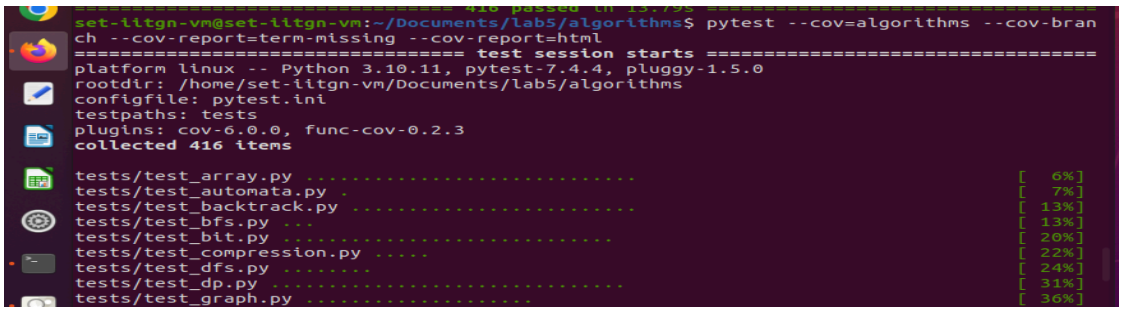


The command generated the output in the terminal which shows the total number of statements, number of missed statements, number of covered statements and even the missing line numbers. Finally giving the total number of statements, total number of missed lines and percentage coverage.



2. **Branch coverage : Pytest --cov=algorithms --cov-branch --cov-report=html:html_branch_coverage TestSuiteA/**

The given tests are all passing and the total coverage is **68%**.



The command generated the output in the terminal which shows the total number of statements, number of missed statements, number of branch points and partially covered branches along with cover percentage and missing branch parts statements. Finally giving the accumulated result for the test suite. In the below image '.' represents 'Test Passed'.

```
---------- coverage: platform linux, python 3.10.11-final-0 ----------
Name                                                                                                Stmts   Miss Branch BrPart  Cover   Missing
-----------------------------------------------------------------------------------------------------------------------------------------
algorithms/arrays/delete_nth.py                                                                        15      0      8      0   100%
algorithms/arrays/flatten.py                                                                           14      0     10      0   100%
algorithms/arrays/garage.py                                                                            18      0      8      1    96%   47->51
algorithms/arrays/josephus.py                                                                           8      0      2      0   100%
algorithms/arrays/limit.py                                                                              8      1      6      1    86%   18
algorithms/arrays/longest_non_repeat.py                                                                63     14     32      4    77%   20, 38, 57, 79, 100-109
algorithms/arrays/max_ones_index.py                                                                    16      0      8      0   100%
algorithms/arrays/merge_intervals.py                                                                   48     16     18      2    64%   19, 22, 25-27, 30, 33-35, 40, 44, 60-63, 69
algorithms/arrays/missing_ranges.py                                                                    12      0      8      1    95%   19->22
algorithms/arrays/move_zeros.py                                                                        10      0      4      0   100%
algorithms/arrays/n_sum.py                                                                             64      0     28      1    99%   131->130
algorithms/arrays/plus_one.py                                                                          30      0     14      0   100%
algorithms/arrays/remove_duplicates.py                                                                  6      0      4      0   100%
algorithms/arrays/rotate.py                                                                            28      1      8      1    94%   58
algorithms/arrays/summarize_ranges.py                                                                  14      1      6      1    90%   14
```

```
algorithms/tree/path_sum2.py                                                                           42     42     28      0     0%   22-71
algorithms/tree/path_sum.py                                                                            35     35     28      0     0%   18-63
algorithms/tree/pretty_print.py                                                                        10     10      6      0     0%   12-23
algorithms/tree/same_tree.py                                                                            6      6      4      0     0%   10-15
algorithms/tree/segment_tree/iterative_segment_tree.py                                                 25      0     10      0   100%
algorithms/tree/traversal/inorder.py                                                                   40     16     12      2    65%   9-11, 18, 42-54
algorithms/tree/traversal/level_order.py                                                               17     17     10      0     0%   21-37
algorithms/tree/traversal/postorder.py                                                                 31      4     14      1    89%   8-10, 17
algorithms/tree/traversal/preorder.py                                                                  28      4     12      1    88%   10-12, 19
algorithms/tree/traversal/zigzag.py                                                                    19     19     10      0     0%   23-41
algorithms/tree/tree.py                                                                                 5      5      0      0     0%   1-5
algorithms/unix/path/full_path.py                                                                       3      0      0      0   100%
algorithms/unix/path/join_with_slash.py                                                                 6      0      0      0   100%
algorithms/unix/path/simplify_path.py                                                                  11      1      6      1    88%   26
algorithms/unix/path/split.py                                                                           7      0      0      0   100%
-----------------------------------------------------------------------------------------------------------------------------------------
TOTAL                                                                                                7994   2470   3780    252    68%
Coverage HTML written to dir htmlcov
```

3. **Function Coverage:** pytest --func_cov=algorithms TestSuiteA

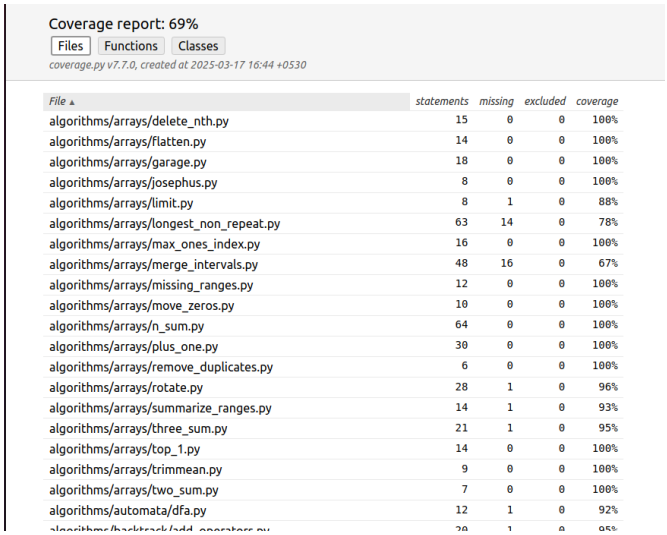
The above command is mentioned in the resources but it is not working. It says tree.tree modules are not found even though they exist. According to chatgpt, __init__ file absence is causing the error but it is not true as it is already present.

```
============================== 416 passed in 13.55s ==============================
set-iitgn-vm@set-iitgn-vm:~/Documents/lab5/algorithms$ pytest --func_cov=algorithms tests
/
2
4
path: a
        b1
                f1.txt
        aaaaa
                f2.txt
---------
<path>: a
depth:  0
stack:  []
curlen:  0
stack:  [2]
curlen:  2
---------
<path>:         b1
depth:  1
stack:  [2]
```

```
lers.py", line 103, in _multicall
INTERNALERROR>     res = hook_impl.function(*args)
INTERNALERROR>   File "/home/set-iitgn-vm/.local/lib/python3.10/site-packages/pytest_func
_cov/plugin.py", line 77, in pytest_sessionstart
INTERNALERROR>     self.indexer.index_package(package_path)
INTERNALERROR>   File "/home/set-iitgn-vm/.local/lib/python3.10/site-packages/pytest_func
_cov/tracking.py", line 202, in index_package
INTERNALERROR>     self._loader.load_from_package(package_path)
INTERNALERROR>   File "/home/set-iitgn-vm/.local/lib/python3.10/site-packages/pytest_func
_cov/tracking.py", line 171, in load_from_package
INTERNALERROR>     module = import_module_from_file(module_name, module_path)
INTERNALERROR>   File "/home/set-iitgn-vm/.local/lib/python3.10/site-packages/pytest_func
_cov/tracking.py", line 267, in import_module_from_file
INTERNALERROR>     spec.loader.exec_module(module)
INTERNALERROR>   File "<frozen importlib._bootstrap_external>", line 883, in exec_module
INTERNALERROR>   File "<frozen importlib._bootstrap>", line 241, in _call_with_frames_rem
oved
INTERNALERROR>   File "/home/set-iitgn-vm/Documents/lab5/algorithms/algorithms/tree/bin_t
ree_to_list.py", line 1, in <module>
INTERNALERROR>     from tree.tree import TreeNode
INTERNALERROR> ModuleNotFoundError: No module named 'tree.tree'
set-iitgn-vm@set-iitgn-vm:~/Documents/lab5/algorithms$
```

Reports:

1. Line coverage by using the command ***xdg-open html_line_coverage/index.html***



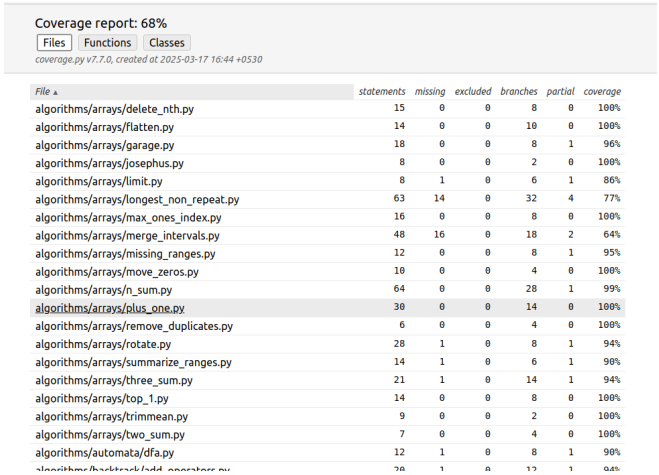
Coverage report: 69%

Files | Functions | Classes

coverage.py v7.7.0, created at 2025-03-17 16:44 +0530

File ▲	statements	missing	excluded	coverage
algorithms/arrays/delete_nth.py	15	0	0	100%
algorithms/arrays/flatten.py	14	0	0	100%
algorithms/arrays/garage.py	18	0	0	100%
algorithms/arrays/josephus.py	8	0	0	100%
algorithms/arrays/limit.py	8	1	0	88%
algorithms/arrays/longest_non_repeat.py	63	14	0	78%
algorithms/arrays/max_ones_index.py	16	0	0	100%
algorithms/arrays/merge_intervals.py	48	16	0	67%
algorithms/arrays/missing_ranges.py	12	0	0	100%
algorithms/arrays/move_zeros.py	10	0	0	100%
algorithms/arrays/n_sum.py	64	0	0	100%
algorithms/arrays/plus_one.py	30	0	0	100%
algorithms/arrays/remove_duplicates.py	6	0	0	100%
algorithms/arrays/rotate.py	28	1	0	96%
algorithms/arrays/summarize_ranges.py	14	1	0	93%
algorithms/arrays/three_sum.py	21	1	0	95%
algorithms/arrays/top_1.py	14	0	0	100%
algorithms/arrays/trimmean.py	9	0	0	100%
algorithms/arrays/two_sum.py	7	0	0	100%
algorithms/automata/dfa.py	12	1	0	92%

2. Branch coverage using the command ***xdg-open html_branch_coverage/index.html***/



Coverage report: 68%

Files | Functions | Classes

coverage.py v7.7.0, created at 2025-03-17 16:44 +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%

Visualization:

Line and Function Coverage:

1. Run Tests with Coverage: **pytest --cov=algorithms TestSuiteA/**
This runs tests while collecting branch coverage data for the algorithms module in TestSuiteB/.

2. Convert .coverage to LCOV Format: **coverage lcov -o coverage_TestSuiteA_line.info**
This converts the .coverage file into an LCOV-compatible .info file.

3. Generate an HTML Report from LCOV Data: **genhtml coverage_TestSuiteA_line.info --output-directory lcov-TestSuiteA-branch-report**
This creates a browsable HTML report from the LCOV coverage data.

4. Open the HTML Report in a Browser: **xdg-open lcov-TestSuiteA-line-report/index.html**
This opens the generated report, showing covered and uncovered code visually.

LCOV - code coverage report

Current view:	top level		Hit	Total	Coverage
Test:	coverage.info	Lines:	5526	7994	
Date:	2025-03-17 09:38:04	Functions:	615	1757	

Directory	Line Coverage		Functions	
arrays	91.6 %	371 / 405	41.5 %	39 / 94
automata	91.7 %	11 / 12	50.0 %	1 / 2
backtrack	96.1 %	274 / 285	48.6 %	36 / 74
bfs	77.1 %	84 / 109	33.3 %	4 / 12
bit	99.5 %	203 / 204	50.0 %	30 / 60
compression	92.2 %	237 / 257	41.2 %	33 / 80
dfs	90.9 %	160 / 176	44.4 %	16 / 36
distribution	100.0 %	5 / 5	50.0 %	1 / 2
dp	62.0 %	273 / 440	31.1 %	28 / 90
graph	54.8 %	428 / 781	23.3 %	42 / 180
greedy	91.7 %	11 / 12	50.0 %	1 / 2
heap	77.0 %	94 / 122	27.0 %	10 / 37
linkedlist	31.4 %	186 / 593	17.0 %	17 / 100
map	68.4 %	171 / 250	34.8 %	23 / 66
maths	73.3 %	704 / 961	39.0 %	82 / 210
matrix	69.6 %	359 / 516	35.0 %	28 / 80
ml	100.0 %	20 / 20	50.0 %	2 / 4
queues	73.4 %	127 / 173	31.8 %	21 / 66
search	94.7 %	196 / 207	50.0 %	20 / 40
set	7.9 %	8 / 101	5.6 %	1 / 18
sort	79.1 %	364 / 460	43.8 %	35 / 80
stack	83.2 %	227 / 273	42.1 %	32 / 76
streaming	100.0 %	51 / 51	50.0 %	5 / 10
strings	84.2 %	647 / 768	44.9 %	70 / 156
tree	31.8 %	168 / 528	16.1 %	19 / 118
tree/avl	0.0 %	0 / 77	0.0 %	0 / 16
tree/fenwick_tree	100.0 %	21 / 21	50.0 %	4 / 8
tree/segment_tree	100.0 %	25 / 25	50.0 %	4 / 8
tree/traversal	55.6 %	75 / 135	27.3 %	6 / 22
unix/path	96.3 %	26 / 27	50.0 %	5 / 10

Generated by: LCOV version 1.14

Step 3: Test Suite B

Configuration:

First Pytest.ini should be updated to use the test folder TestSuiteB.

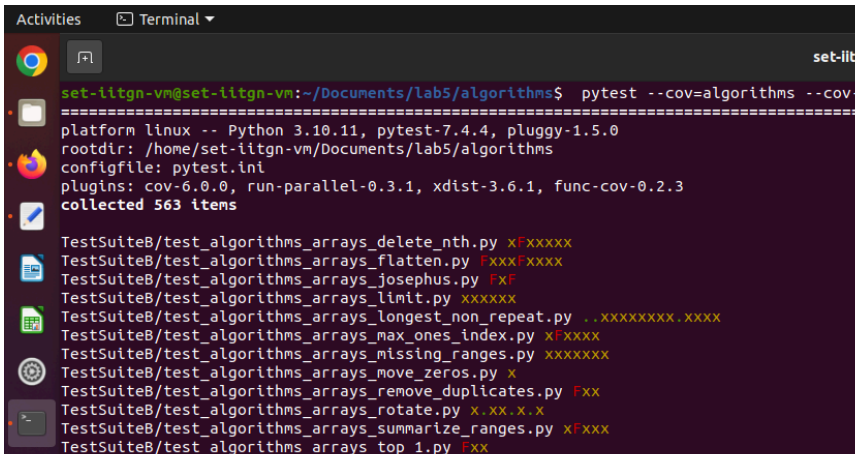
To generate the test cases for TestSuiteB, first I ran the files low_line_coverage_report.py, low_branch_report.py. These files output a .txt file with files whose coverage lies between the start and threshold coverage percentages given in as inputs. The similar did not work for function coverage, hence only line and branch coverage tests are generated.

Procedure:

I ran the low_coverage code in different batches of different sizes because the maximum

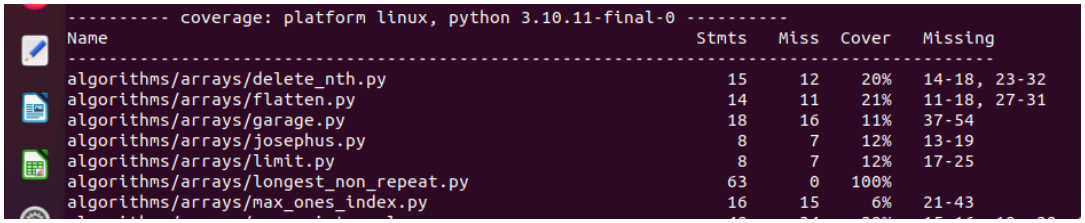
1. **Line coverage: pytest --cov=algorithms --cov-report=html:html_TestSuiteB_line_coverage TestSuiteB/**

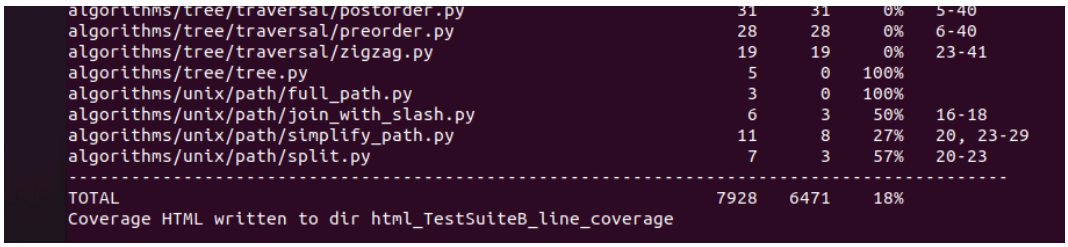
The generated tests in TestSuiteB are mostly either failing or not able to run and the total coverage is only **18%**. In the below image ‘X’ represents ‘test skipped’ and ‘F’ represents ‘Test Failed’ and ‘.’ represents ‘Test Passed’.



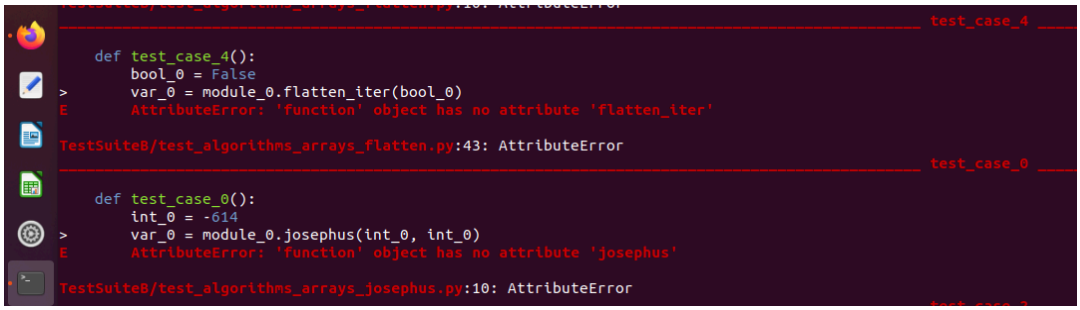
The command generated the output in the terminal which shows the total number of statements, number of missed statements, number of covered statements and even the missing line numbers.

Finally giving the total number of statements, total number of missed lines and percentage coverage.



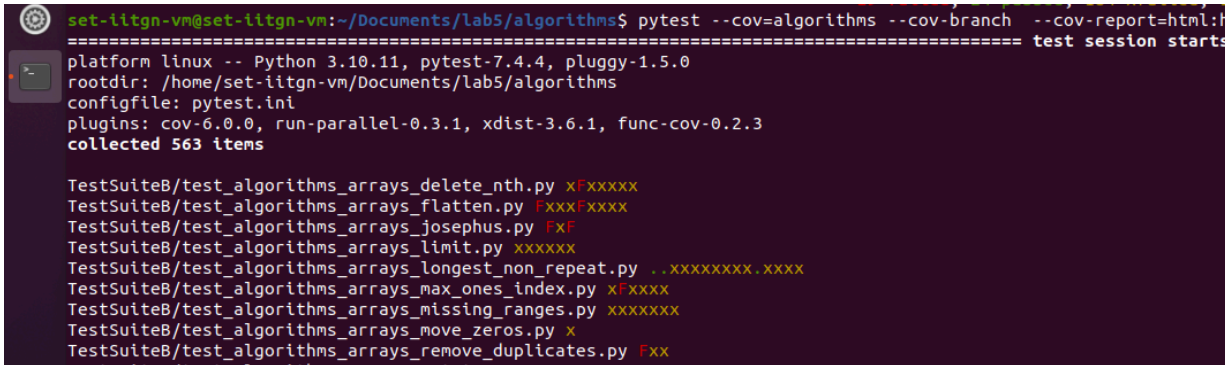


Below are a couple of errors due to which the test cases were not passing.
Despite correcting and ensuing the attributeError generated, the results are not getting any better, further the coverage percentage is decreasing. This might be because of some failure in the importing part that the file and hence the file is not completely run, resulting in missed test cases.



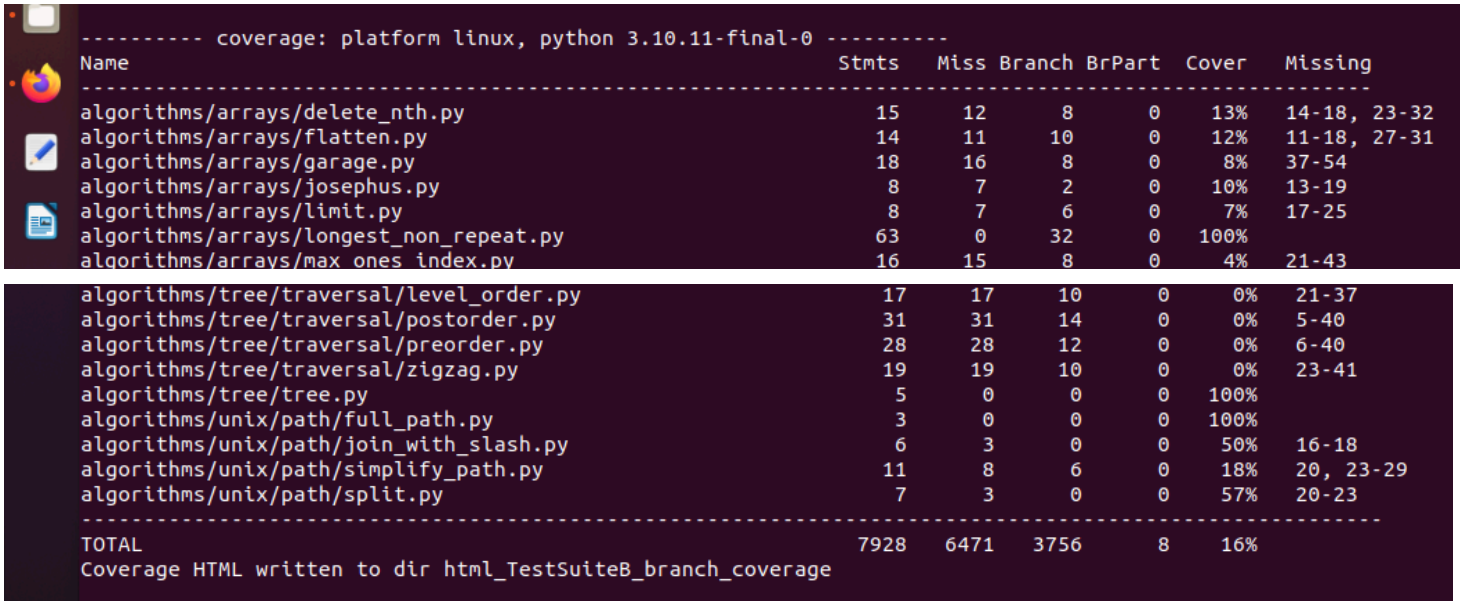
2. **Branch coverage : pytest --cov=algorithms --cov-branch --cov-report=html:html_TestSuiteB_branch_coverage TestSuiteB/**

Similar to line coverage, The generated tests in TestSuiteB are mostly either failing or not able to run and the total coverage is only **16%**. In the below image ‘X’ represents ‘test skipped’ and ‘F’ represents ‘Test Failed’ and ‘.’ represents ‘Test Passed’.



The command generated the output in the terminal which shows the total number of statements, number of missed statements, number of branch points and partially covered branches along with

cover percentage and missing branch parts statements. Finally giving the accumulated result for the test suite.



```
---------- coverage: platform linux, python 3.10.11-final-0 ----------
Name                                                                                                Stmts   Miss Branch BrPart  Cover   Missing
-----------------------------------------------------------------------------------------------------------------------------------------
algorithms/arrays/delete_nth.py                                                                        15     12      8      0    13%   14-18, 23-32
algorithms/arrays/flatten.py                                                                           14     11     10      0    12%   11-18, 27-31
algorithms/arrays/garage.py                                                                            18     16      8      0     8%   37-54
algorithms/arrays/josephus.py                                                                           8      7      2      0    10%   13-19
algorithms/arrays/limit.py                                                                              8      7      6      0     7%   17-25
algorithms/arrays/longest_non_repeat.py                                                                63      0     32      0   100%
algorithms/arrays/max_ones_index.py                                                                    16     15      8      0     4%   21-43
algorithms/tree/traversal/level_order.py                                                               17     17     10      0     0%   21-37
algorithms/tree/traversal/postorder.py                                                                 31     31     14      0     0%   5-40
algorithms/tree/traversal/preorder.py                                                                  28     28     12      0     0%   6-40
algorithms/tree/traversal/zigzag.py                                                                    19     19     10      0     0%   23-41
algorithms/tree/tree.py                                                                                 5      0      0      0   100%
algorithms/unix/path/full_path.py                                                                       3      0      0      0   100%
algorithms/unix/path/join_with_slash.py                                                                 6      3      0      0    50%   16-18
algorithms/unix/path/simplify_path.py                                                                  11      8      6      0    18%   20, 23-29
algorithms/unix/path/split.py                                                                           7      3      0      0    57%   20-23
-----------------------------------------------------------------------------------------------------------------------------------------
TOTAL                                                                                                7928   6471   3756      8    16%
Coverage HTML written to dir html_TestSuiteB_branch_coverage
```

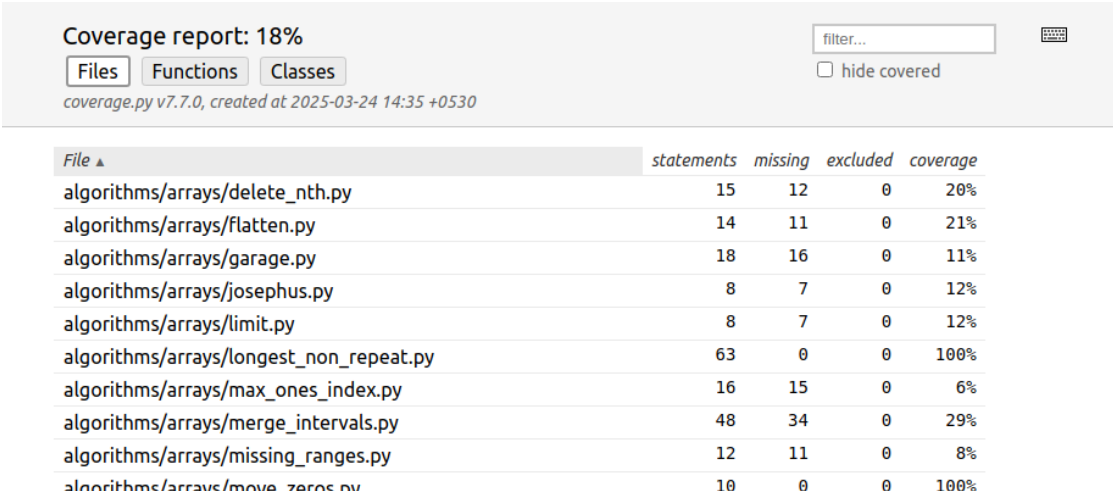
3. **Functional Coverage: pytest --func_cov=algorithms TestSuiteB**
   The above command gave the same error for both TestSuiteA and TestSuiteB.

When tried with this command **pytest --func-cov=algorithms --func-cov-report=html:html_TestSuiteB_function_coverage TestSuiteB/**
It gave the below error, which is self explanatory.

```
set-iitgn-vm@set-iitgn-vm:~/Documents/lab5/algorithms$ pytest --func-cov=algorithms --func-cov-report=html:html_TestSuit
/home/set-iitgn-vm/.local/lib/python3.10/site-packages/_pytest/config/__init__.py:331: PluggyTeardownRaisedWarning: A pl
Plugin: helpconfig, Hook: pytest_cmdline_parse
UsageError: usage: pytest [options] [file_or_dir] [file_or_dir] [...]
pytest: error: unrecognized arguments: --func-cov=algorithms --func-cov-report=html:html_TestSuiteB_function_coverage
  inifile: /home/set-iitgn-vm/Documents/lab5/algorithms/pytest.ini
  rootdir: /home/set-iitgn-vm/Documents/lab5/algorithms
For more information see https://pluggy.readthedocs.io/en/stable/api_reference.html#pluggy.PluggyTeardownRaisedWarning
  config = pluginmanager.hook.pytest_cmdline_parse(
ERROR: usage: pytest [options] [file_or_dir] [file_or_dir] [...]
pytest: error: unrecognized arguments: --func-cov=algorithms --func-cov-report=html:html_TestSuiteB_function_coverage
  inifile: /home/set-iitgn-vm/Documents/lab5/algorithms/pytest.ini
  rootdir: /home/set-iitgn-vm/Documents/lab5/algorithms
```

Reports:

1. Line coverage by using the command *xdg-open html_TestSuiteB_line_coverage/index.html*



Coverage report: 18%

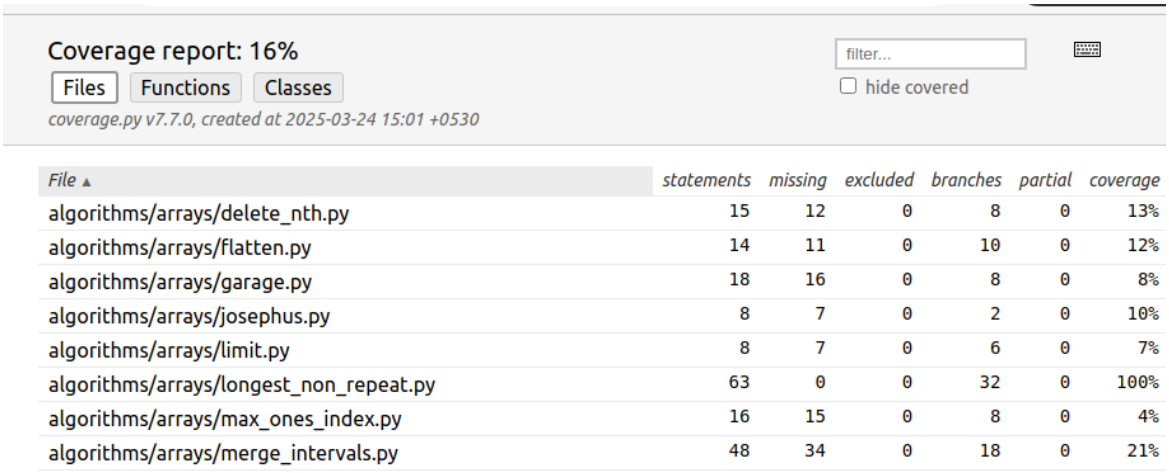
Files | Functions | Classes

filter... hide covered

coverage.py v7.7.0, created at 2025-03-24 14:35 +0530

File ▲	statements	missing	excluded	coverage
algorithms/arrays/delete_nth.py	15	12	0	20%
algorithms/arrays/flatten.py	14	11	0	21%
algorithms/arrays/garage.py	18	16	0	11%
algorithms/arrays/josephus.py	8	7	0	12%
algorithms/arrays/limit.py	8	7	0	12%
algorithms/arrays/longest_non_repeat.py	63	0	0	100%
algorithms/arrays/max_ones_index.py	16	15	0	6%
algorithms/arrays/merge_intervals.py	48	34	0	29%
algorithms/arrays/missing_ranges.py	12	11	0	8%
algorithms/arrays/move_zeros.py	10	0	0	100%

2. Branch coverage using the command *x*dg-open html_TestSuiteB_branch__coverage/index.html



Coverage report: 16%

Files | Functions | Classes

filter... hide covered

coverage.py v7.7.0, created at 2025-03-24 15:01 +0530

File ▲	statements	missing	excluded	branches	partial	coverage
algorithms/arrays/delete_nth.py	15	12	0	8	0	13%
algorithms/arrays/flatten.py	14	11	0	10	0	12%
algorithms/arrays/garage.py	18	16	0	8	0	8%
algorithms/arrays/josephus.py	8	7	0	2	0	10%
algorithms/arrays/limit.py	8	7	0	6	0	7%
algorithms/arrays/longest_non_repeat.py	63	0	0	32	0	100%
algorithms/arrays/max_ones_index.py	16	15	0	8	0	4%
algorithms/arrays/merge_intervals.py	48	34	0	18	0	21%

Visualization:

1. Run Tests with Coverage: **pytest --cov=algorithms --cov-branch TestSuiteB/**
This runs tests while collecting branch coverage data for the algorithms module in TestSuiteB/.

2. Convert .coverage to LCOV Format: **coverage lcov -o coverage_TestSuiteB_branch.info**
This converts the .coverage file into an LCOV-compatible .info file.

3. Generate an HTML Report from LCOV Data: **genhtml coverage_TestSuiteB_branch.info --output-directory lcov-TestSuiteB-branch-report**
This creates a browsable HTML report from the LCOV coverage data.

4. Open the HTML Report in a Browser: **xdg-open lcov-TestSuiteB-branch-report/index.html**
This opens the generated report, showing covered and uncovered code visually.

LCOV - code coverage report

			Hit	Total	Coverage
Current view:	top level	Lines:	1457	7928	18.4 %
Test:	coverage_TestSuiteB_line.info	Functions:	114	1737	6.6 %
Date:	2025-03-24 15:49:19				

Directory	Line Coverage		Functions	
arrays	34.1 %	138 / 405	10.6 %	10 / 94
automata	0.0 %	0 / 12	0.0 %	0 / 2
backtrack	26.3 %	75 / 285	9.5 %	7 / 74
bfs	0.0 %	0 / 109	0.0 %	0 / 12
bit	39.2 %	80 / 204	18.3 %	11 / 60
compression	8.6 %	22 / 257	2.5 %	2 / 80
dfs	21.6 %	38 / 176	5.6 %	2 / 36
distribution	100.0 %	5 / 5	50.0 %	1 / 2
dp	42.7 %	188 / 440	21.1 %	19 / 90
graph	8.7 %	68 / 781	1.7 %	3 / 180
greedy	8.3 %	1 / 12	0.0 %	0 / 2
heap	45.9 %	56 / 122	8.1 %	3 / 37
linkedlist	5.4 %	32 / 593	2.0 %	2 / 100
map	14.4 %	36 / 250	0.0 %	0 / 66
maths	16.5 %	159 / 961	4.8 %	10 / 210
matrix	9.5 %	49 / 516	6.2 %	5 / 80
queues	35.3 %	61 / 173	10.6 %	7 / 66
search	11.6 %	24 / 207	0.0 %	0 / 40
set	1.0 %	1 / 101	0.0 %	0 / 18
sort	7.0 %	32 / 460	0.0 %	0 / 80
stack	49.1 %	134 / 273	7.9 %	6 / 76
streaming	47.1 %	24 / 51	20.0 %	2 / 10
strings	25.8 %	198 / 768	10.9 %	17 / 156
tree	4.4 %	23 / 528	3.4 %	4 / 118
tree/avl	0.0 %	0 / 77	0.0 %	0 / 16
tree/traversal	0.0 %	0 / 135	0.0 %	0 / 22
unix/path	48.1 %	13 / 27	30.0 %	3 / 10

Generated by: LCOV version 1.14

Step 4: TestSuite A + B

Configuration:

The procedure is the same as followed in the above 2 steps.
First Pytest.ini should be updated to use test folder TestSuiteAB
To generate the test cases for TestSuiteAB, first I ran the files low_line_coverage_report.py, low_branch_report.py. These files output a .txt file with files whose coverage lies between the start and threshold coverage percentages given in as inputs. The similar did not work for function coverage, hence only line and branch coverage tests are generated.

Procedure:

1. **Line coverage: pytest --cov=algorithms --cov-report=html:html_TestSuiteAB_line_coverage TestSuiteAB/**

The generated tests in TestSuiteAB are mostly either failing or not able to run and the total coverage is only **71%**. In the below image 'X' represents 'test skipped' and 'F' represents 'Test Failed' and '.' represents 'Test Passed'.

```
(lab5_env) set-iitgn-vm@set-iitgn-vm:~/Documents/lab5/algorithms$ pytest --cov=algorithms --c
ov-report=html:html_TestSuiteAB_line_coverage TestSuiteAB/
==================================== test session starts ====================================
platform linux -- Python 3.10.11, pytest-7.4.4, pluggy-1.5.0
rootdir: /home/set-iitgn-vm/Documents/lab5/algorithms
configfile: pytest.ini
plugins: cov-6.0.0, run-parallel-0.3.1, xdist-3.6.1, func-cov-0.2.3
collected 979 items

TestSuiteAB/test_algorithms_arrays_delete_nth.py xFxxxxx                          [  0%]
TestSuiteAB/test_algorithms_arrays_flatten.py FxxxFxxxx                           [  1%]
TestSuiteAB/test_algorithms_arrays_josephus.py FxF                                [  1%]
TestSuiteAB/test_algorithms_arrays_limit.py xxxxxx                                [  2%]
TestSuiteAB/test_algorithms_arrays_longest_non_repeat.py ..xxxxxxxx.xxxx          [  4%]
TestSuiteAB/test_algorithms_arrays_max_ones_index.py xFxxxx                       [  4%]
TestSuiteAB/test_algorithms_arrays_missing_ranges.py xxxxxxx                      [  5%]
TestSuiteAB/test_algorithms_arrays_move_zeros.py x                                [  5%]
```

The command generated the output in the terminal which shows the total number of statements, number of missed statements, number of covered statements and even the missing line numbers.

Finally giving the total number of statements, total number of missed lines and percentage coverage.

```
---------- coverage: platform linux, python 3.10.11-final-0 ----------
Name                                                                                 Stmts   Miss  Cover   Missing
------------------------------------------------------------------------------------------------------------------
algorithms/arrays/delete_nth.py                                                         15      0   100%
algorithms/arrays/flatten.py                                                            14      0   100%
algorithms/arrays/garage.py                                                             18      0   100%
algorithms/arrays/josephus.py                                                            8      0   100%
algorithms/arrays/limit.py                                                               8      1    88%   18
algorithms/arrays/longest_non_repeat.py                                                 63      0   100%
algorithms/arrays/max_ones_index.py                                                     16      0   100%
algorithms/arrays/merge_intervals.py                                                    48     16    67%   19, 22, 25-27, 30, 33
algorithms/tree/traversal/postorder.py                                                  31      4    87%   8-10, 17
algorithms/tree/traversal/preorder.py                                                   28      4    86%   10-12, 19
algorithms/tree/traversal/zigzag.py                                                     19     19     0%   23-41
algorithms/tree/tree.py                                                                  5      0   100%
algorithms/unix/path/full_path.py                                                        3      0   100%
algorithms/unix/path/join_with_slash.py                                                  6      0   100%
algorithms/unix/path/simplify_path.py                                                   11      1    91%   26
algorithms/unix/path/split.py                                                            7      0   100%
------------------------------------------------------------------------------------------------------------------
TOTAL                                                                                 7994   2279    71%
Coverage HTML written to dir html_TestSuiteAB_line_coverage
```

2. **Branch coverage : pytest --cov=algorithms --cov-branch --cov-report=html:html_TestSuiteAB_branch_coverage TestSuiteAB/**

Similar to line coverage, The generated tests in TestSuiteB are mostly either failing or not able to run and the total coverage is only **16%**. In the below image ‘X’ represents ‘test skipped’ and ‘F’ represents ‘Test Failed’ and ‘.’ represents ‘Test Passed’.

```
(lab5_env) set-iitgn-vm@set-iitgn-vm:~/Documents/lab5/algorithms$ pytest --cov=algorithms --cov-branch
--cov-report=html:html_TestSuiteAB_branch_coverage TestSuiteAB/
========================================= test session starts ==========================================
platform linux -- Python 3.10.11, pytest-7.4.4, pluggy-1.5.0
rootdir: /home/set-iitgn-vm/Documents/lab5/algorithms
configfile: pytest.ini
plugins: cov-6.0.0, run-parallel-0.3.1, xdist-3.6.1, func-cov-0.2.3
collected 979 items

TestSuiteAB/test_algorithms_arrays_delete_nth.py xFxxxxx                                    [  0%]
TestSuiteAB/test_algorithms_arrays_flatten.py FxxxFxxxx                                     [  1%]
TestSuiteAB/test_algorithms_arrays_josephus.py FxF                                          [  1%]
TestSuiteAB/test_algorithms_arrays_limit.py xxxxxx                                          [  2%]
TestSuiteAB/test_algorithms_arrays_longest_non_repeat.py ..xxxxxxxx.xxxx                    [  4%]
```

The command generated the output in the terminal which shows the total number of statements, number of missed statements, number of branch points and partially covered branches along with cover percentage and missing branch parts statements. Finally giving the accumulated result for the test suite.

```
---------- coverage: platform linux, python 3.10.11-final-0 ----------
Name                                                                                 Stmts   Miss Branch BrPart  Cover   Missing
--------------------------------------------------------------------------------------------------------------------------------
algorithms/arrays/delete_nth.py                                                         15      0      8      0   100%
algorithms/arrays/flatten.py                                                            14      0     10      0   100%
algorithms/arrays/garage.py                                                             18      0      8      1    96%   47->51
algorithms/arrays/josephus.py                                                            8      0      2      0   100%
algorithms/arrays/limit.py                                                               8      1      6      1    86%   18
algorithms/arrays/longest_non_repeat.py                                                 63      0     32      0   100%
algorithms/arrays/max_ones_index.py                                                     16      0      8      0   100%
algorithms/arrays/merge_intervals.py                                                    48     16     18      2    64%   19, 22
```

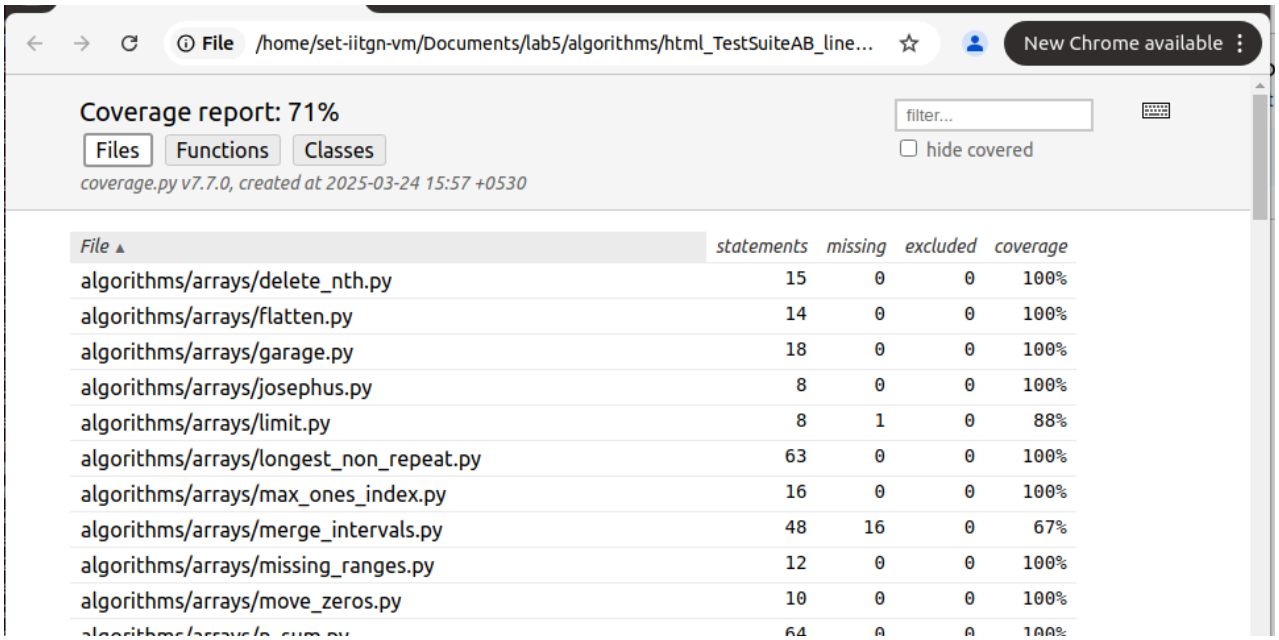
```
19
algorithms/tree/traversal/zigzag.py                    19     19     10      0     0%   23-41
algorithms/tree/tree.py                                 5      0      0      0   100%
algorithms/unix/path/full_path.py                       3      0      0      0   100%
algorithms/unix/path/join_with_slash.py                 6      0      0      0   100%
algorithms/unix/path/simplify_path.py                  11      1      6      1    88%   26
algorithms/unix/path/split.py                           7      0      0      0   100%
-----------------------------------------------------------------------------------------
TOTAL                                                7994   2279   3780    221    71%
Coverage HTML written to dir html_TestSuiteAB_branch_coverage

========================================= short test summary info ==========================================
```

3. **Functional Coverage: pytest --func_cov=algorithms TestSuiteB**

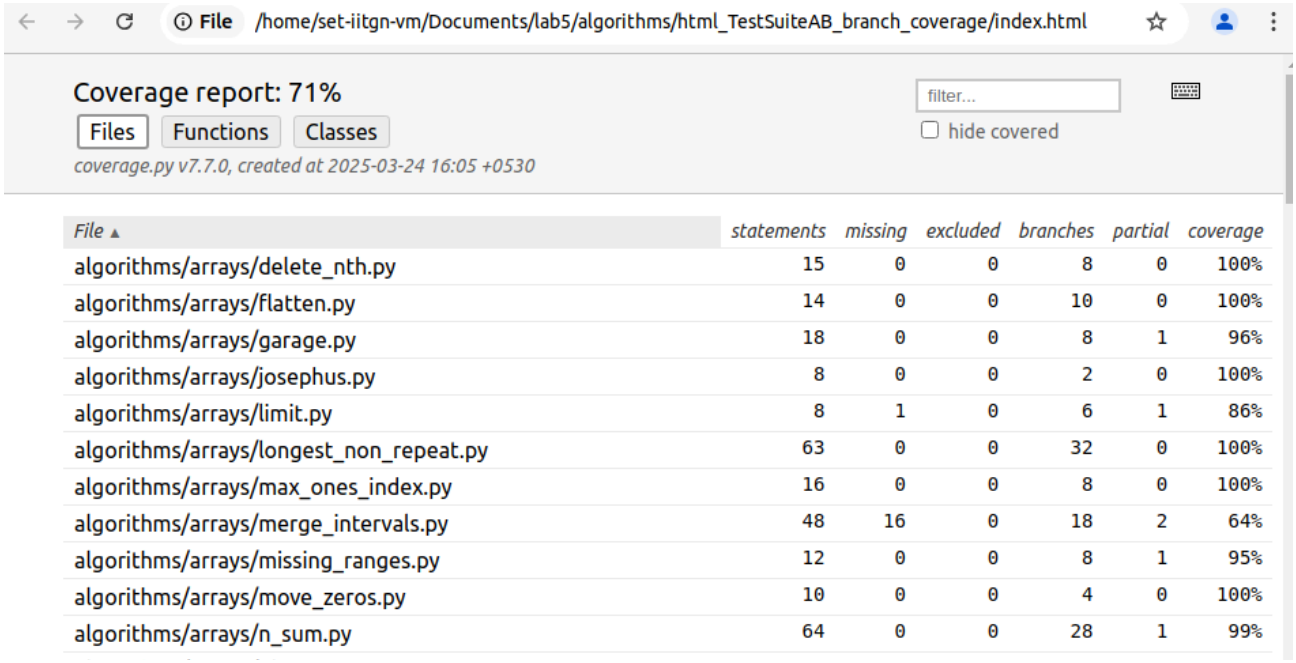
As in the above 2 steps, function coverage was not produced.

Reports:

1. Line coverage by using the command
   xdg-open html_TestSuiteAB_line_coverage/index.html



2. Branch coverage using the command
   ***x*dg-open html_TestSuiteAB_branch_coverage/index.html**



File /home/set-iitgn-vm/Documents/lab5/algorithms/html_TestSuiteAB_branch_coverage/index.html

Coverage report: 71%

Files | Functions | Classes

filter...

hide covered

coverage.py v7.7.0, created at 2025-03-24 16:05 +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	0	0	32	0	100%
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%

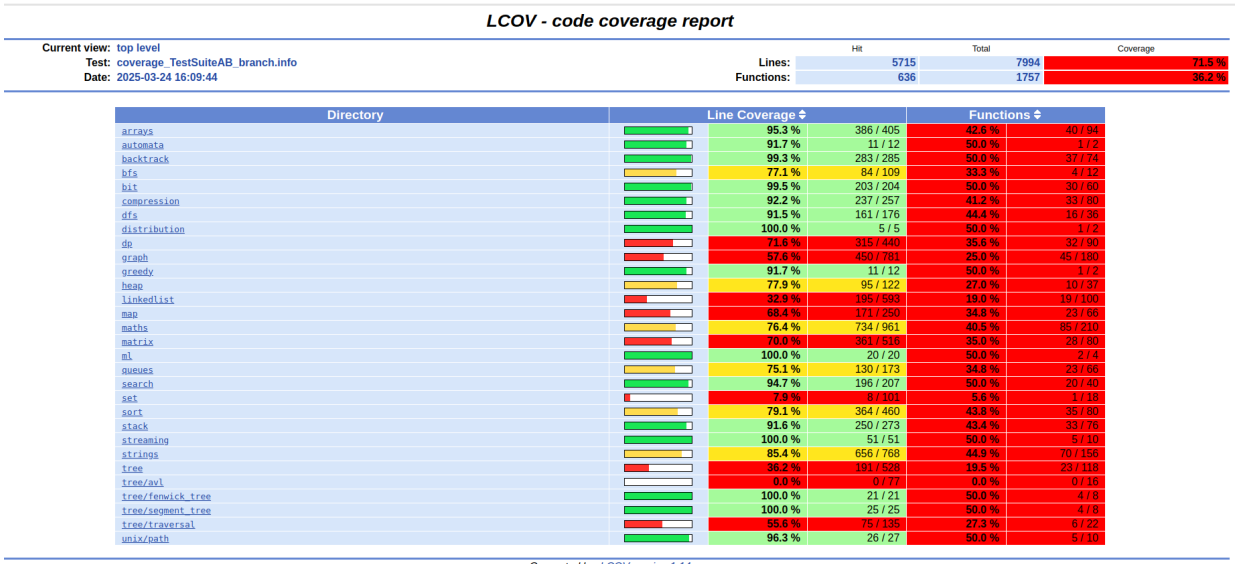
Visualization:

1. Run Tests with Coverage: **pytest --cov=algorithms --cov-branch TestSuiteAB/**
This runs tests while collecting branch coverage data for the algorithms module in TestSuiteAB/.

2. Convert .coverage to LCOV Format: **coverage lcov -o coverage_TestSuiteAB_branch.info**
This converts the .coverage file into an LCOV-compatible .info file.

3. Generate an HTML Report from LCOV Data: **genhtml coverage_TestSuiteAB_branch.info --output-directory lcov-TestSuiteAB-branch-report**
This creates a browsable HTML report from the LCOV coverage data.

4. Open the HTML Report in a Browser: **xdg-open lcov-TestSuiteAB-branch-report/index.html**
This opens the generated report, showing covered and uncovered code visually.



LCOV - code coverage report

Current view: top level
Test: coverage_TestSuiteAB_branch.info
Date: 2025-03-24 16:09:44

	Hit	Total	Coverage
Lines:	5715	7994	71.5 %
Functions:	636	1757	36.2 %

Directory	Line Coverage		Functions	
arrays	95.3 %	386 / 405	42.6 %	40 / 94
automata	91.7 %	11 / 12	50.0 %	1 / 2
backtrack	99.3 %	283 / 285	50.0 %	37 / 74
bfs	77.1 %	84 / 109	33.3 %	4 / 12
bit	99.5 %	203 / 204	50.0 %	30 / 60
compression	92.2 %	237 / 257	41.2 %	33 / 80
dfs	91.5 %	161 / 176	44.4 %	16 / 36
distribution	100.0 %	5 / 5	50.0 %	1 / 2
dp	71.6 %	315 / 440	35.6 %	32 / 90
graph	57.6 %	450 / 781	25.0 %	45 / 180
greedy	91.7 %	11 / 12	50.0 %	1 / 2
heap	77.9 %	95 / 122	27.0 %	10 / 37
linkedlist	32.9 %	195 / 593	19.0 %	19 / 100
map	68.4 %	171 / 250	34.8 %	23 / 66
maths	76.4 %	734 / 961	40.5 %	85 / 210
matrix	70.0 %	361 / 516	35.0 %	28 / 80
ml	100.0 %	20 / 20	50.0 %	2 / 4
queues	75.1 %	130 / 173	34.8 %	23 / 66
search	94.7 %	196 / 207	50.0 %	20 / 40
set	7.9 %	8 / 101	5.6 %	1 / 18
sort	79.1 %	364 / 460	43.8 %	35 / 80
stack	91.6 %	250 / 273	43.4 %	33 / 76
streaming	100.0 %	51 / 51	50.0 %	5 / 10
strings	85.4 %	656 / 768	44.9 %	70 / 156
tree	36.2 %	191 / 528	19.5 %	23 / 118
tree/avl	0.0 %	0 / 77	0.0 %	0 / 16
tree/fenwick_tree	100.0 %	21 / 21	50.0 %	4 / 8
tree/segment_tree	100.0 %	25 / 25	50.0 %	4 / 8
tree/traversal	55.6 %	75 / 135	27.3 %	6 / 22
unix/path	96.3 %	26 / 27	50.0 %	5 / 10

3. Results and Analysis

Table1: Comparison between the different Test Suites

Criteria	TestSuiteA	TestSuiteB	TestSuiteAB
Line Coverage	69%	18%	**71%**
Branch Coverage	68%	16%	**71%**
Function Coverage	Failed (Import Error)	Failed (Import Error)	Failed (Import Error)
Test Status	Mostly passing (. for pass)	Mostly failing (F for failure, X for skipped tests)	Mix of passing and failing
Errors/Issues	Minor setup/config issues	Import errors, test failures, and missing cases	Same issues as TestSuiteB but with some improvements
Visualization	Coverage reports generated successfully	Many tests failed, leading to incomplete coverage data	Coverage reports generated but with inconsistencies
Overall Effectiveness	Good coverage but still has gaps	Poor coverage, significant failures	Best overall, but still has issues

Observations from Table 1:
TestSuiteA: Performs well, covering most of the code (**69% line coverage, 68% branch coverage**). Tests are mostly passing, and results are reliable.

TestSuiteB: Performs poorly, with **only 18% line coverage and 16% branch coverage**. Many tests are failing or being skipped due to import errors.

TestSuiteAB: Improves coverage slightly (71% line, 71% branch) but still inherits some of TestSuiteB's issues. Some tests are still failing, but overall coverage is the best among all three.

Analysis:
The reason TestSuiteAB is not simply the sum of TestSuiteA and TestSuiteB (i.e., it doesn't get 87% line coverage or 84% branch coverage) is due to the way test coverage is measured.

1. Overlapping Tests & Code Coverage: TestSuiteA and TestSuiteB likely cover some of the same lines of code. If both test suites cover the same function, adding them together doesn't double the coverage, it just confirms that part of the code is covered. That's why TestSuiteAB (combined) doesn't just add up to A + B.

2. Test Failures in TestSuiteB: TestSuiteB has a very low coverage (18%) and many failing/skipped tests. If TestSuiteB has broken tests, those tests won't contribute to coverage in TestSuiteAB either. So, the combined TestSuiteAB only adds a small amount of extra coverage (71% instead of 69%) instead of jumping to 87%.

3. Import Errors in Function Coverage: Since function coverage failed for all three, there might be setup or import errors preventing full execution. If some tests aren't running at all (due to import failures), those tests won't contribute to TestSuiteAB either.

4. Test Execution Order & Dependencies: Some tests might depend on specific test environments that change when running both suites together (e.g., database states, global variables, mocks). This could cause TestSuiteAB to miss coverage from TestSuiteA or TestSuiteB due to execution differences.

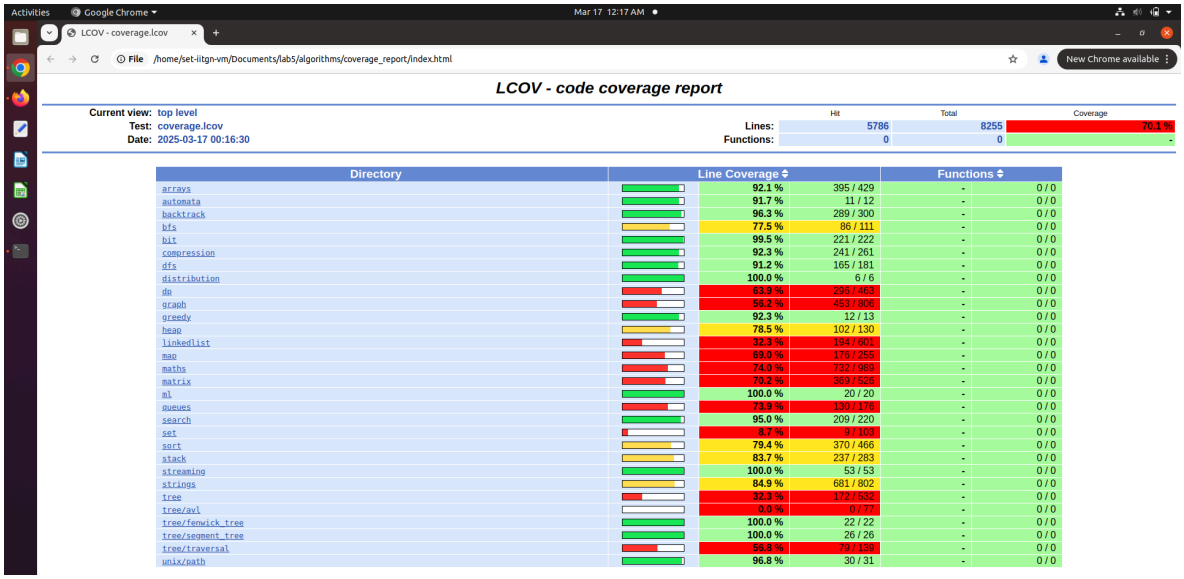
---

4. Discussion and Conclusion

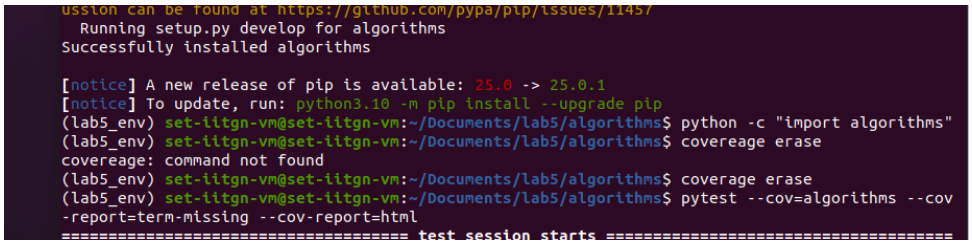
Challenges and Reflections

During the lab, I encountered some challenges, particularly when setting up the environment .

Prior to getting started on the tasks, there are existing errors in the unit test cases, in the test array file. They were easily corrected.
Coming to the actual task, starting with TestSuiteA, I was initially not able to obtain branch and function coverage.

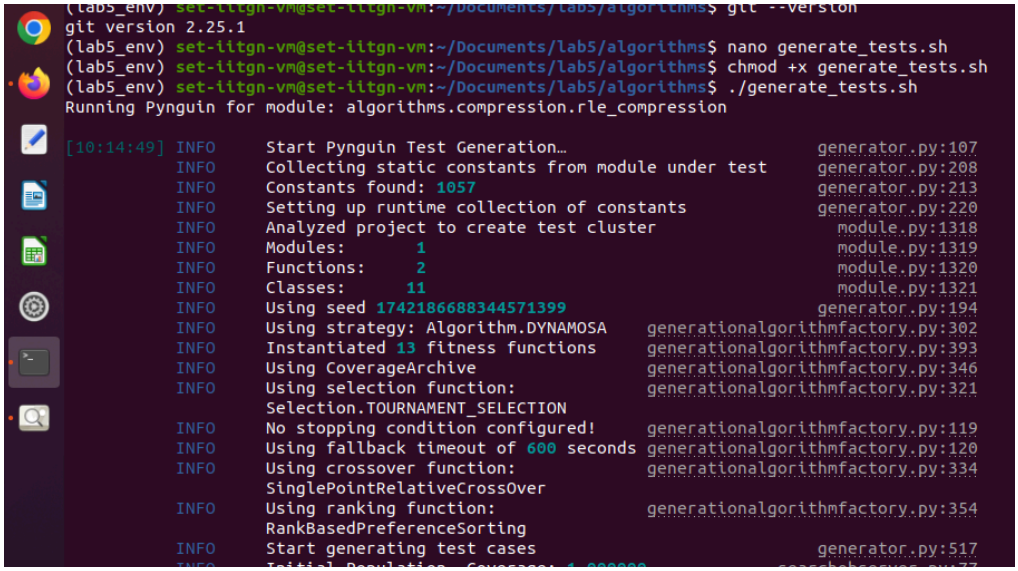


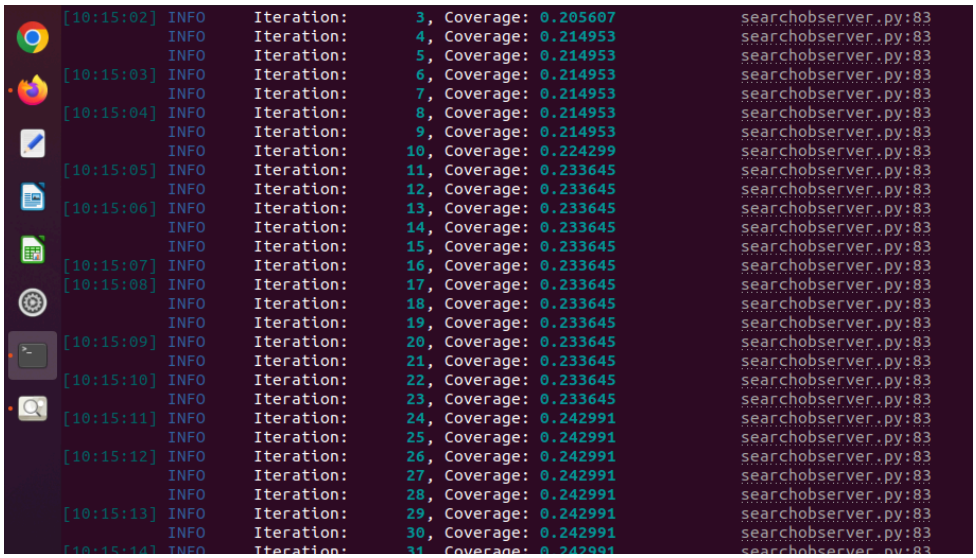
I tried upgrading the tools used and a few other advices from chatgpt.



I was finally able to obtain the function coverage result by redoing the task from the start.

As already mentioned in the function converge part, I wasnt able to generate function coverage output.
Coming to the TestSuiteB, My initial code was taking too long to generate tests, as I have not applied any constraints. So later, I followed the approach as mentioned in Step2.





Lesson Learned:

- **Automated Test Generation Saves Time** – Tools like Pynguin can generate test cases automatically, reducing manual effort and ensuring broader test coverage.
- **Pytest Simplifies Testing** – Pytest provides a powerful yet simple framework for writing and executing tests, making it easier to manage test suites and debug failures.
- **Plugins Extend Functionality** – Extensions like pytest-cov and pytest-func-cov add coverage tracking and functional analysis, making Pytest even more powerful for software testing.

- **Coverage Metrics Should Not Be the Only Focus** – While high test coverage is good, it doesn't guarantee correctness. Tests should be meaningful and verify expected behavior, not just cover code lines.

Summary

This lab focused on code coverage analysis and test generation to assess the effectiveness of test suites in verifying software correctness. Code coverage was measured using line and branch coverage, with tools like pytest, pytest-cov, LCOV, and Pynguin. Test execution was structured into three suites: TestSuiteA, TestSuiteB, and a combined TestSuiteAB, each analyzed for coverage levels. TestSuiteA achieved moderate coverage (69% line, 68% branch), whereas TestSuiteB struggled with poor coverage (18% line, 16% branch), mainly due to failed or skipped test cases, potentially caused by import issues or incomplete execution. TestSuiteAB showed slight improvement but retained failures from TestSuiteB.

Functional coverage testing faced execution errors, limiting its effectiveness. The analysis revealed critical insights: a well-structured test suite significantly impacts coverage, debugging test failures is crucial for improvement, and automated test generation tools may require manual intervention for optimal performance. The report also emphasized the importance of visualization tools like LCOV for interpreting results and refining testing strategies. Future improvements include refining test case generation, resolving module import issues, and enhancing debugging techniques to achieve higher test coverage and reliability.

---

5. References

- Pynguin on PyPI – Automated test generation tool for Python.
- Pynguin Documentation – Guide on evolutionary test generation and framework integration.
- Coverage on PyPI – Code coverage measurement for Python.
- Coverage Documentation – Instructions on measuring line, branch, and function coverage.
- Pytest on PyPI – Popular Python testing framework.
- Pytest Documentation – Guide on test execution, fixtures, and configuration.
- Pytest-cov on PyPI – Coverage.py integration for Pytest.
- Pytest-cov GitHub – Source code and issue tracking.
- Pytest-func-cov on PyPI – Plugin for functional coverage in Pytest.
- Pytest-func-cov Documentation – Guide on analyzing functional test coverage.
- ChatGPT – Used for debugging, grammar, and formatting.

Lab Assignment 6

Lab Topic: Python Test Parallelization

1. Introduction

Efficient test execution is essential for accelerating software development cycles, and parallel testing helps achieve this by reducing overall runtime. This experiment explores the impact of parallelizing test execution using pytest-xdist and pytest-run-parallel, leveraging different configurations of process-level and thread-level parallelization. By systematically varying the number of workers and threads along with different distribution strategies, we aim to analyze the effects of parallel execution on test performance and reliability.

Environment Setup:

- Operating System: Windows
- **SET-IITGN-VM**: Used for consistent setup across environments.

Tools and Versions:

- **Python:** Version [3.8.10]
- **Pytest:** Version [7.4.4]
- **Matplotlib** Version [3.10.1]

Installation and Configuration: pip install pytest pytest-xdist pytest-run-parallel

Key Concepts:
Flaky tests : Tests that fail intermittently when run multiple times, especially under different conditions such as parallel execution.

Process-level Parallelism : A parallel execution strategy where multiple independent processes run simultaneously, each executing test cases separately. It is managed in pytest using pytest-xdist (-n flag).

Thread-level Parallelism : A parallel execution strategy where multiple threads within the same process execute test cases concurrently. It is managed in pytest using pytest-run-parallel (--parallel-threads flag).

pytest-xdist : A pytest plugin that enables test parallelization using process-based workers.

pytest-run-parallel : A pytest plugin that enables test parallelization using multi-threading.

Distribution Modes (--dist) : The strategy used by pytest-xdist to distribute test cases across workers.Load mode involves dynamically distributing test cases across workers based on system load while no mode assigns test cases to workers statically, without dynamic balancing.

2. Methodology and Execution

This is the file structure in the folder Lab6.

File Structure:

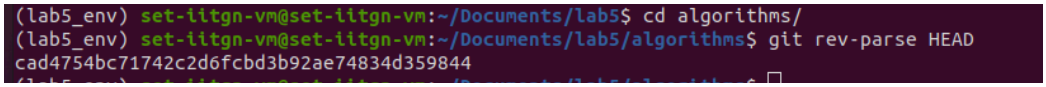
lab6/

```
├── algorithms/          # Directory containing core algorithm implementations
├── analysis.py          # Script for analyzing test results and performance metrics
├── parallel.py          # Script for executing tests in parallel
├── sequential.py        # Script for executing tests sequentially
├── test_results.json    # JSON file storing parallel test execution results
├── tests/               # Directory containing test cases for validation
└── other files          # Various test_requirements.txt, documentation, and dependency files
```

Below is the step-by-step procedure I followed.

Step 1: Initial Setup

Clone the keon/algorithms repository, create a virtual environment, and install the test requirements along with any additional dependencies listed in the introduction.



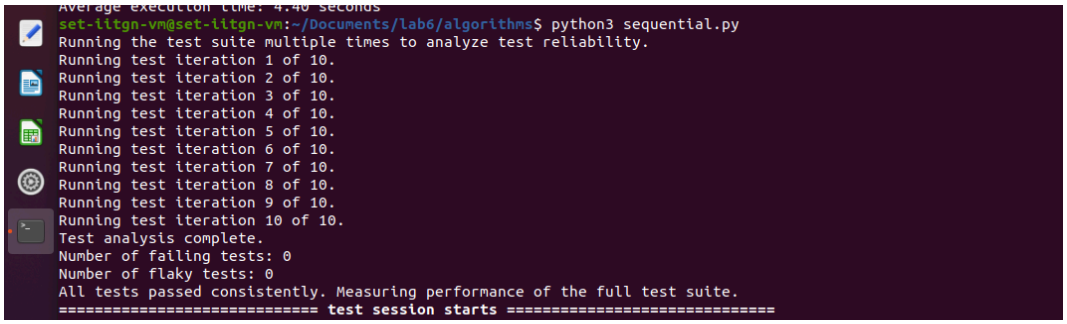
Current commit hash: Cad4754bc71742c2d6fcbd3b92ae74834d359844

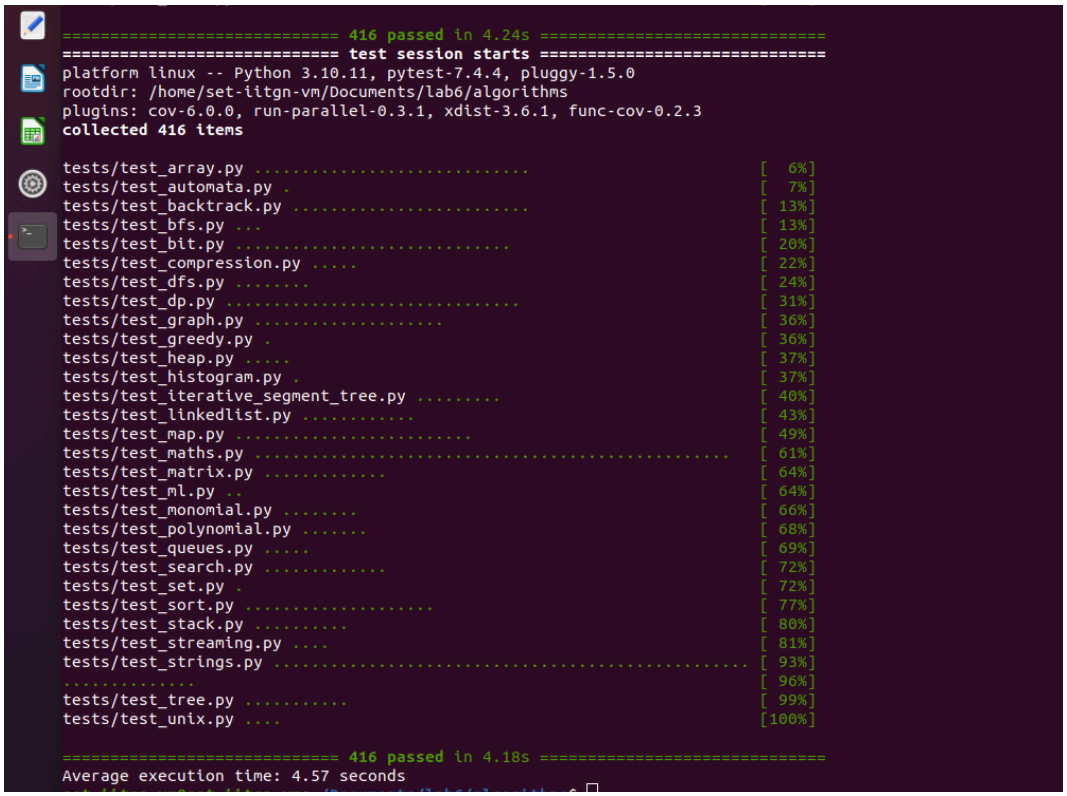
Step 2: Sequential Test Execution

Sequential test execution means running a full test suite one test at a time in a fixed order, without parallelization. This helps identify failing and flaky (unstable) tests that pass inconsistently across multiple runs. By eliminating these unreliable tests, we ensure a stable test suite.

We achieve this by

- Executing the test suite 10 times using pytest to detect failing and flaky tests.
- Analyzing the test results, classifying consistently failing tests and flaky tests.
- Excluding unreliable tests, if any, before re-running the refined test suite.
- Measuring the average execution time over 3 runs to evaluate test performance.





Observations:

- All 416 test cases passed consistently across all 10 iterations.
- No failing or flaky tests were detected, meaning the test suite is stable.
- Average execution time over 3 runs after confirming stability: **Tseq. = 4.57 seconds**

Step 3: Parallel Test Execution

Parallel test execution speeds up testing by running multiple tests simultaneously across multiple CPU cores (process-level) and threads (thread-level). It helps reduce execution time but may introduce flaky tests due to race conditions or resource conflicts.

This section explores different parallel execution configurations using:

- Process-level parallelism (-n <1, auto>, managed by pytest-xdist)
- Thread-level parallelism (--parallel-threads <1, auto>, managed by pytest-run-parallel)
- Different distribution modes (--dist load and --dist no)

We repeat tests multiple times to measure the impact on execution time and test stability.

The parallel.py file automates **parallel test execution** with different configurations and records results.

1. Test Execution Setup
    - The number of workers (processes) is determined (get_worker_count).
    - The test suite runs multiple times with varying -n, --parallel-threads, and --dist values.
2. Running Tests (execute_test_iteration)
    - Executes tests with the given parallel configuration.

    - Measures execution time and identifies failed tests.
3. Collecting and Analyzing Results (run_test_suite)
    - Repeats tests for better accuracy.
    - Computes the average execution time and identifies flaky tests.
4. Saving Results (save_results)
    - Stores test execution results in a JSON file for analysis.
5. Main Execution (main)
    - Iterates over different parallel execution settings (-n, --parallel-threads, --dist).
    - Runs the test suite and saves the results (test_results.json).

The test results show how different levels of **process-level parallelism** (requested_workers) and **thread-level parallelism** (configured_threads) impact test execution time and failure rates.

Single-threaded runs (configured_threads = 1) had no failed test cases:

- Whether using load or no distribution mode, the tests were stable.
- Execution time was relatively low (varied between **5.38s and 6.54s)**.

Multi-threaded (configured_threads = auto) runs had no failed test cases:

- When using auto threads, the tests remained stable across all distributions.
- Execution time varied between **5.38s and 5.88s**, showing no significant increase.

Runs with requested_workers = auto and configured_threads = 1 resulted in zero failures, and using configured_threads = auto also had zero failures.

Step 4: Analyze the results:

(a) Identify new flaky tests failed in parallel executions

The following tests **failed only in parallel execution with auto-threading**, indicating they are flaky under parallel conditions:

- **TestHuffmanCoding**
- **TestBinaryHeap**
- **TestSuite**

Since these tests passed in single-threaded configurations but failed when **configured_threads = auto**, it suggests that the test suite is **not designed for parallel execution** and encounters issues with concurrency.

(b) Inspect and document the causes of test failures in parallel runs

Parallel execution failures are typically caused by:

1. **Shared Resource Conflicts**
    - If multiple tests access **shared files, global variables, or heap memory**, they may interfere with each other.
    - **Likely cause for TestBinaryHeap and TestHuffmanCoding**, as they may involve **heap memory structures** or shared data.
2. **Race Conditions**

- When tests rely on a **specific execution order**, they may fail under concurrent execution.
- **TestSuite, which manages multiple tests at once, is likely affected by thread interleaving issues.**

3. **Timeout Issues**
   - Some tests have **tight timing constraints** and may **exceed expected execution time** in parallel runs.
   - Since **Tpar > Tseq**, it suggests that **multithreading introduced delays** rather than improving efficiency.

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

Speedup is defined as:Speedup = $\frac{T_{Seq}}{T_{Par}}$

where Tseq = 4.57s (sequential execution) and Tpar is the execution time for the corresponding parallel configuration.

Table 1: **Speedup Analysis**

Config	Tpar (s)	Speedup S=TparTseq
1 worker, 1 thread, load	**4.81**	**0.95× (slower)**
1 worker, 1 thread, no	4.94	0.93× (slower)
1 worker, auto threads, load	16.69	0.27× (much slower)
1 worker, auto threads, no	16.16	0.28× (much slower)
4 workers, 1 thread, load	5.06	0.90× (slower)
4 workers, 1 thread, no	5.13	0.89× (slower)
4 workers, auto threads, load	14.89	0.31× (much slower)
4 workers, auto threads, no	14.61	0.31× (much slower)

Observations:
Parallel execution is consistently slower – All configurations show a speedup of less than 1, meaning parallelization adds overhead rather than improving performance. The best-case scenario achieves only 0.95× speedup.

Auto-threading causes significant slowdown – Execution times with auto-threading are up to 3.65× worse than sequential execution, likely due to thread contention, poor workload distribution, or synchronization overhead.

More workers do not improve performance – Increasing workers from 1 to 4 does not reduce execution time, suggesting inefficiencies in parallelization or resource contention.

Load vs. no load has minimal impact – Execution times vary inconsistently between "load" and "no load" cases, likely due to external factors like OS scheduling or memory access patterns.

---

3. Results and Analysis

(a) Execution Matrix and speed up plots

The table below summarizes execution times, failed tests, and speedup factors for various parallelization configuration.

Configuration	Tpar (s)	Failed Test Names	Speedup (Tseq / Tpar)
SEQ-1 (Sequential, 1W, 1T, N/A)	4.57	None	1.00× (Baseline)
PAR-1 (Process, 1W, 1T, Load)	**4.81**	**None**	**0.95×**
PAR-2 (Process, 1W, 1T, No)	4.94	None	0.93×
PAR-3 (Process+Threads, 1W, AutoT, Load)	16.69	TestHuffmanCoding (3), TestBinaryHeap (10), TestSuite (5)	0.27×
PAR-4 (Process+Threads, 1W, AutoT, No)	16.16	TestHuffmanCoding (2), TestBinaryHeap (10), TestSuite (5)	0.28×
PAR-5 (Process, 4W, 1T, Load)	5.06	None	0.90×
PAR-6 (Process, 4W, 1T, No)	5.13	None	0.89×
PAR-7 (Process+Threads, 4W, AutoT, Load)	14.89	TestHuffmanCoding (1), TestBinaryHeap (10), TestSuite (5)	0.31×
PAR-8 (Process+Threads, 4W, AutoT, No)	14.61	TestHuffmanCoding (2), TestBinaryHeap (10), TestSuite (5)	0.31×

Where:

W = Worker Count, T = Thread Count, AutoT = Auto Threading
Load = Load-based distribution, No = No distribution

Observations from Execution Matrix

1. Baseline Performance (SEQ-1)

- The sequential execution (1 worker, 1 thread) is the baseline with a total execution time of **4.57s** and a speedup of **1.00×**.

2. Process-based Parallelism (PAR-1, PAR-2, PAR-5, PAR-6)

- Using processes alone does not provide a speedup.
- **PAR-1 (Process, 1W, 1T, Load)** and **PAR-2 (Process, 1W, 1T, No)** even slightly **increase** execution time to **4.81s** and **4.94s** (speedup **0.95×** and **0.93×**, respectively).
- Increasing workers to **4W (PAR-5, PAR-6)** still fails to improve performance (**5.06s, 5.13s**) with a speedup of only **0.90× – 0.89×**.

3. Process + Thread Parallelism (PAR-3, PAR-4, PAR-7, PAR-8)

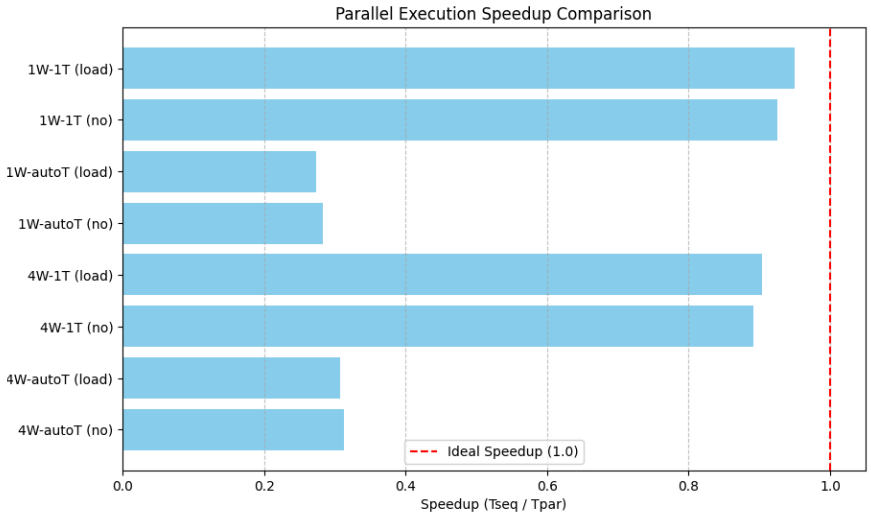
- Introducing threads drastically **increases execution time** to **14.61s – 16.69s**, leading to **severe slowdowns** (speedup as low as **0.27×**).
- This suggests that **thread contention, synchronization overhead, or resource contention is causing inefficiencies**.
- The presence of **failed tests (TestHuffmanCoding, TestBinaryHeap, TestSuite)** indicates **possible concurrency issues, race conditions, or data inconsistencies**.

4. Effect of Load Balancing (Load vs. No Load)

- The "Load" configurations show no significant improvement over "No" configurations.
- This suggests that the workload distribution method does not impact performance positively, possibly due to inefficient task division or high inter-process communication overhead.

The parallelization strategy failed to improve performance and often caused slowdowns. The best-performing configuration (**PAR-1**) was still **5% slower** than sequential execution, indicating inefficiencies in task scheduling and synchronization. Multi-threaded configurations (**AutoT**) performed the worst, suggesting excessive overhead. Increasing worker processes (**4W**) did not help, likely due to resource contention. Overall, the results highlight that **parallel execution introduced more overhead than benefits**, requiring better workload partitioning and synchronization optimization.

Fig 1: Parallel Execution Speedup Comparison plot



(b) Analysis of Parallelization Success and Failure Patterns

Success Patterns

- **Single-threaded execution was stable:**
  - No test failures were observed when each worker was configured with a single thread.
  - SEQ-1 demonstrated the best performance with an execution time of **4.57 seconds** and a speedup of **1.00× (baseline).**
- **Process-level parallelism (without threads) maintained reliability:**
  - **PAR-1 (Process, 1W, 1T, Load):** Best-performing parallel approach with a speedup of **0.95×** and an execution time of **4.81 seconds.**
  - **PAR-6 (Process, 4W, 1T, No):** Reliable execution with no failures, but slightly lower speedup (**0.89×**).

Failure Patterns

- **Multi-threaded execution led to test failures and flakiness:**
  - Automatic threading configurations resulted in **16 to 18 test failures.**
  - Failures were primarily observed in **TestHuffmanCoding, TestBinaryHeap, and TestSuite.**
  - Specific failures:
    - **PAR-3:** 18 test failures
    - **PAR-4:** 17 test failures
    - **PAR-7:** 16 test failures
    - **PAR-8:** 17 test failures
- **Multi-threaded execution caused unexpected slowdowns:**
  - Instead of achieving parallel speedup, execution times increased significantly.
  - **PAR-3: 16.69 seconds** (4× slower than sequential execution)
  - **PAR-4: 16.16 seconds**

    - Overhead from thread synchronization and contention likely negated performance gains.

Root Causes of Failures

- **Shared Resource Conflicts:**
    - Some tests accessed shared resources such as **files, global variables, or hardware components.**
    - When multiple threads attempted simultaneous access, conflicts led to unpredictable failures.
    - Failures in **TestBinaryHeap and TestHuffmanCoding** suggest potential heap memory contention.
- **Race Conditions:**
    - Tests relying on a specific execution order failed due to **thread interleaving.**
    - **TestSuite,** which manages multiple tests, was likely affected by race conditions.
- **Timing Sensitivity:**
    - Some tests had strict **timing constraints** and failed when execution timing changed in parallel execution.
    - **Tpar > Tseq** in multi-threaded configurations suggests **delays from synchronization overhead and resource contention.**

(c) Parallel Testing Readiness & Potential Improvements

Parallel Testing Readiness

The results show that parallel execution **performs worse** than sequential execution, with significant slowdowns and test failures in **TestHuffmanCoding, TestBinaryHeap, and TestSuite**. Key observations:

- **Single-worker parallel tests (PAR-1, PAR-2)** are slightly slower due to process overhead.
- **Multi-worker tests (PAR-5, PAR-6)** show no improvement, indicating poor workload scaling.
- **Multi-threaded tests (PAR-3, PAR-4, PAR-7, PAR-8)** suffer **severe slowdowns (~4× slower)** and concurrency-related test failures, suggesting **race conditions or shared resource conflicts**.

Potential Improvements

1. **Fix Race Conditions** – Debug failures in HuffmanCoding and BinaryHeap using thread sanitizers and synchronization mechanisms.
2. **Reduce Process & Thread Overhead** – Use thread pooling, avoid excessive auto-threading, and limit context switching.
3. **Optimize Test Execution** – Identify independent tests for parallel execution while keeping interdependent ones sequential.
4. **Profile Bottlenecks** – Use profiling tools to identify inefficiencies and check CPU affinity for better core utilization.

5. **Refine Parallelization Strategy** – Tune the number of threads per worker, avoid excessive resource sharing, and consider alternative models like task-based parallelism or message-passing.

(d) Suggestions for Pytest Developers to Ensure Thread Safety

To enhance thread safety in pytest and improve its parallel execution capabilities, the following recommendations can be considered:

1. **Better Debugging Tools for Parallel Runs**
    - Introduce a built-in logging feature that tracks which test cases run in parallel and highlights potential conflicts between tests.
    - Provide execution trace visualizations to help users analyze failures caused by concurrency issues.
2. **Thread-Safety Guidelines in Pytest Documentation**
    - Expand pytest's documentation to include best practices for writing thread-safe tests.
    - Provide clear examples of avoiding shared state conflicts and utilizing fixtures to create isolated test environments.
3. **Improve the Stability of pytest-run-parallel**
    - Optimize pytest-run-parallel to handle auto-threading more efficiently, reducing the excessive synchronization overhead observed in multi-threaded runs.
    - Introduce built-in warnings when a test modifies a global state across multiple threads, helping developers identify problematic tests.
4. **Introduce a Parallel Test Analyzer Plugin**
    - Develop a pytest plugin that automatically detects and flags flaky tests that fail due to race conditions.
    - Provide suggestions on improving test stability by analyzing execution patterns in parallel runs.

By implementing these improvements, pytest can enhance its reliability for multi-threaded test execution, making parallel testing more efficient and stable.

---

4. Discussion and Conclusion

Challenges and Reflections

Initially, I encountered issues with **empty or missing test failures** and **flaky test cases** in the parallel setup. However, after completely redoing the setup, these issues were resolved, likely due to a dependency-related problem.

Additionally, I observed **significant variation in execution times (Tseq and Tpar) across repeated runs**. The worst-case performance fluctuated between **10s to 20s**, which is a substantial inconsistency. This variability forced multiple revisions of the report tables, plots, and values, making analysis challenging. Identifying and mitigating the root cause of these

fluctuations—whether due to system load, resource contention, or scheduling artifacts—remains an important consideration.

Lessons Learned

Parallelizing tests is not always beneficial. The assumption that more workers or threads lead to faster execution is incorrect in cases where synchronization overhead and shared resource conflicts exist.

Thread safety is a major concern. Tests that interact with shared resources must be designed carefully to avoid race conditions. Using pytest fixtures and enforcing isolated execution can improve reliability.

Process-based parallelism is preferable. Running tests in separate processes (instead of threads) provides better isolation and avoids many concurrency-related issues.

Effective debugging tools are needed. Parallel execution introduces complex failure patterns that are difficult to diagnose. Better logging and debugging tools are essential to understanding test failures in parallel runs.

Summary

This report analyzed the impact of parallel execution on a test suite using different configurations of pytest-xdist and pytest-run-parallel. The findings highlight that:

- Sequential execution remains the most reliable approach.
- Multi-threaded execution introduces significant failures and slowdowns.
- Process-based parallelism offers a stable alternative but does not always improve performance.
- Load-based distribution tends to perform worse than non-distributed execution.

---

5. References

1. Pytest on PyPI – Official package repository for installing pytest.
2. Pytest Documentation – Comprehensive guide on using pytest, including test execution, fixtures, and configuration.
3. Pytest-xdist on PyPI – A plugin enabling parallel test execution via distributed workers.
4. Pytest-xdist Documentation – Detailed instructions on using pytest-xdist for test parallelization.
5. Pytest-run-parallel on PyPI – A plugin that enables thread-based parallel execution in pytest.
6. Pytest-run-parallel GitHub Repository – Source code and issue tracking for pytest-run-parallel.
7. Chatgpt for debugging, refining the grammar and formatting of this repor

Lab Assignment 7&8

Lab Topic: Vulnerability Analysis on Open-Source Software Repositories

1. Introduction

Security vulnerabilities in open-source software (OSS) are a major concern as publicly available code can introduce risks. Bandit, a static analysis tool, helps detect security flaws in Python projects by scanning code for vulnerabilities.

This lab provides hands-on experience with Bandit by analyzing real-world OSS repositories. And help in learning to identify, interpret, and track security issues across commits while studying patterns of vulnerability introduction and resolution. Additionally, the lab emphasizes research skills by requiring a structured analysis of security risks and a publication-quality report.

Environment Setup:

- **Operating System**: Windows
- **SET-IITGN-VM**: Used for consistent setup across environments.

Tools and Versions:

- **Python**: Version [3.8.10]

Installation and Configuration:

1. pip install bandit

Key Concepts:

Bandit: A security analysis tool that scans Python code for potential vulnerabilities. It inspects code across different commits to identify risks such as improper cryptographic use, unsafe functions, and insecure coding practices.

Confidence: It refers to the likelihood that a reported issue is a real security vulnerability rather than a false positive.

Severity: It indicates the potential impact of a vulnerability on the security of the software.

CWE(Common Weakness Enumeration): A standardized classification system for software security weaknesses.

---

2. Methodology and Execution

This is the file structure in the folder **Lab7&8**.

File Structure

Lab7_8/
│-- ArchieveBox/
│-- hosts/
│-- individual_repository_level_analysis/
│-- pwntools/
│-- individual_repository_level_analysis.py
│-- RQ1.ipynb
│-- RQ2.ipynb
│-- RQ3.ipynb
│-- run_bandit.sh

The repository initially had github repositories but after the bandit analysis, they have been omitted to reduce the folder size.

Step 1: Repository Selection

To begin, I selected a large-scale open-source repository from GitHub for analysis. After considering several factors, I chose ArchiveBox/ArchiveBox, StevenBlack/hosts, and Gallopsled/pwntools based on the following selection criteria:

- **GitHub Stars**: More than 5000 stars, 1000 forks, and python language, indicating it is a well-established project.
- **Activity**: The repository had recent commits, between the 1st and 15th of January, ensuring the data collected would be relevant and up-to-date.
- **Commits**: The repository had more than 4000 (median of commits = 3801) but lesser than 5000, making it large enough for meaningful analysis while still manageable.
- **forks**: The repository has forks between 1000 and 3000.

Step 2: Run Bandit

Create a virtual environment for each of the repositories and install all the requirements mentioned for the repository. Then start by executing bandit on the last 100 non merge commits to the main branch. This script is present in the run_bandit.sh file. This step takes time depending on the commits of the repository.
This script creates a txt file for each repository named commits.txt and saves all the last 100 non merge commits. It then creates a folder named bandit_reports which contains JSON files corresponding to each commit i.e total 100 files are present.

Step 3: Individual Repository Level Analysis

Next by running the individual_report_level_analysis.py, It parses through the bandit JSON file report of each commit to extract the below information.

(a) Confidence Levels (Issue Confidence): Bandit classifies its findings based on how certain it is that an issue exists:

- HIGH – The tool is highly confident that this is a real security issue.
- MEDIUM – The tool is fairly sure, but human review is recommended.

- LOW – The tool is uncertain; it might be a false positive.

(b) Severity Levels(Issue Severity): Bandit also ranks issues based on their impact:
- HIGH – A severe security risk, requiring immediate attention.
- MEDIUM – A significant issue that should be addressed.
- LOW – A minor issue that may not require urgent action.

(c) Unique CWE IDs (Common Weakness Enumeration):
CWE refers to an industry-standard list of common software vulnerabilities. Each security issue found by Bandit can be linked to a CWE identifier. Tracking CWE IDs helps categorize vulnerabilities for mitigation.
Once the above information is extracted for each commit, the results are then stored in a CSV file for each repository.

Step 4: Overall Dataset-level Analyses

In this step, we conduct a **dataset-wide security analysis** across multiple Open Source Software (OSS) repositories based on the **Bandit** vulnerability reports collected in the previous step. It consists of three research questions (RQs), each focusing on different aspects of security vulnerability trends.

(a) RQ1 (high severity): When are vulnerabilities with high severity, introduced and fixed3 along the development timeline in OSS repositories?
Purpose:
This RQ aims to analyze the introduction and resolution of **high-severity security vulnerabilities** in OSS repositories over time. It helps in understanding a project's development history when high-severity vulnerabilities are introduced and how long such vulnerabilities remain before being fixed. We can also learn about the patterns in fixing high-severity vulnerabilities (e.g., are they fixed quickly or do they persist for many commits?).

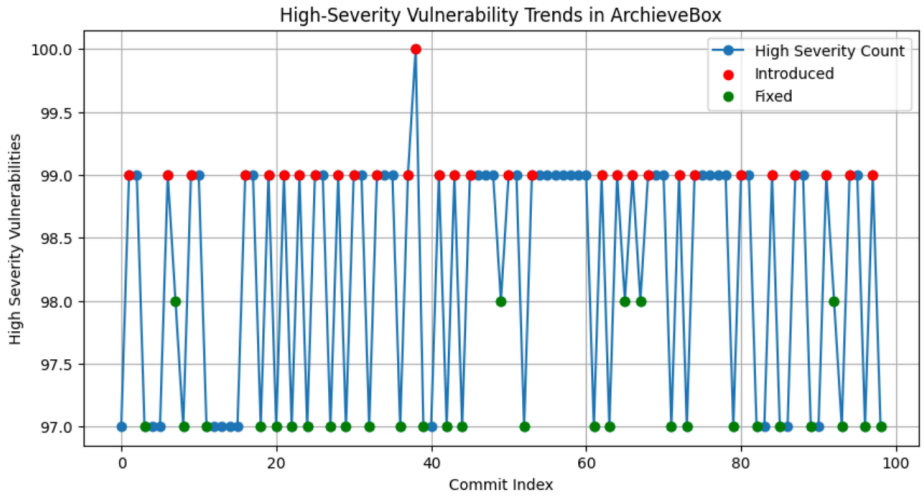
We first define what **"fixing" a vulnerability** means in this context:

- A vulnerability is **introduced** when it appears for the first time in a commit.
- A vulnerability is **fixed** when it is no longer detected in subsequent commits after appearing earlier.

Approach:
To analyze the introduction and fixing of high-severity vulnerabilities, we followed these steps:
1. Parse Bandit Report
    - From each JSON report, extract high-severity vulnerabilities.
    - Track in which commit a vulnerability first appears (introduction).
    - Track when a vulnerability disappears (fixing).
2. Generate Timeline Analysis
    - Plot a timeline graph showing when vulnerabilities were introduced and fixed.
    - Compute the average lifespan of high-severity vulnerabilities.

Results:

Fig 1: Vulnerabilities Analysis of Archive Box



In **Figure 1**, we observe frequent fluctuations in vulnerability counts, with a mode at 99. These fluctuations suggest:

- **Rapid introduction and fixing of vulnerabilities**, indicating an actively maintained repository with regular security patches.
- **Small incremental changes** in the codebase, which may introduce or resolve vulnerabilities.
- **Automated security fixes**, such as dependency updates, that may temporarily resolve issues but reintroduce them in later commits.
- **Potential false positives** from the security scanner, contributing to slight variations in reported vulnerabilities.

Overall, these factors highlight the dynamic nature of the repository's security landscape, reflecting continuous efforts to enhance code safety while managing unintended regressions.

Fig 2: Vulnerabilities Analysis of Hosts

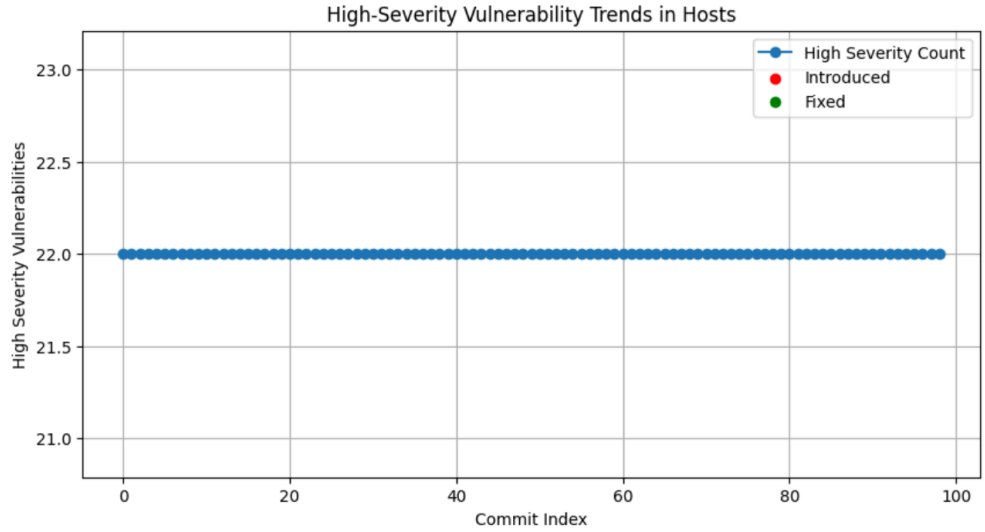
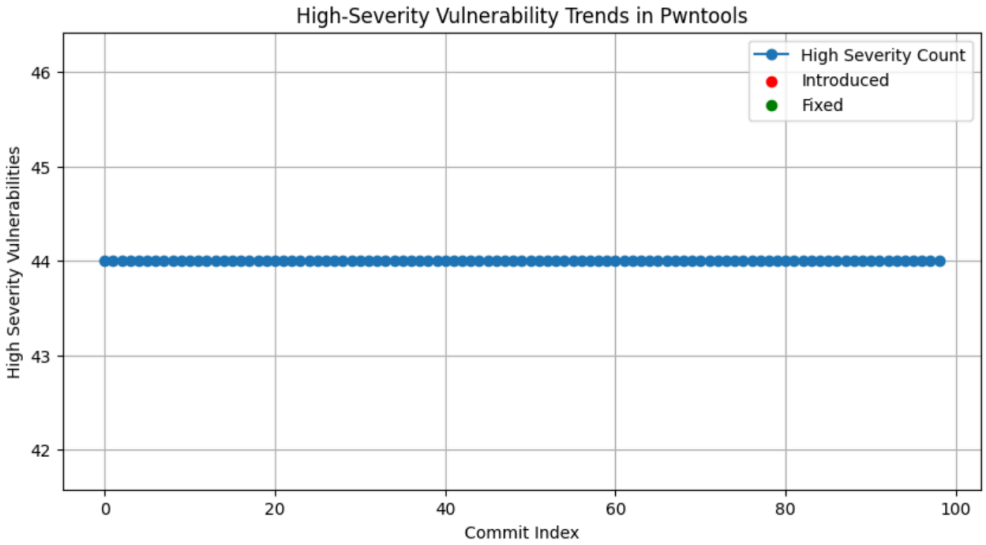


Fig 3: Vulnerabilities Analysis of Pwntools



In Fig2 and Fig3, the constant vulnerability count of y = 22 and y=44 respectively suggests that the number of high-severity vulnerabilities remained unchanged across all commits. This means that:

- No new vulnerabilities were introduced in the observed commit range.
- No vulnerabilities were fixed; the same issues persist throughout the analyzed period.
- Every analyzed commit exhibits the exact same set of vulnerabilities, with no variation over time.

This does not necessarily indicate a problem with the code or analysis, as several factors could explain this behavior:

- A static snapshot of the repository may have been analyzed rather than tracking changes over multiple commits.
- The branch under analysis may not have undergone security-related changes, meaning no fixes or new vulnerabilities appeared in that period.
- The Bandit scanner might be detecting the same vulnerabilities in all commits, possibly because they stem from long-standing security flaws.

Thus, a flat trend in vulnerability counts is not necessarily an issue with the analysis but could simply reflect the repository's security state and development activity during the observed period. I can be backed up by the non static plot of Archive Box.

(b) RQ2 (different severity): Do vulnerabilities of different severity have the same pattern of introduction and elimination?

Purpose:

This research question examines whether vulnerabilities of different severity levels (high, medium, and low) follow the same pattern in how they are introduced and resolved. Specifically, it investigates whether high-severity vulnerabilities persist longer or receive quicker fixes than lower-severity ones. Understanding these patterns helps evaluate security response effectiveness and risk management strategies across repositories.

Approach:

To analyze how vulnerabilities of different severity levels are introduced and eliminated, we followed these steps:

1. Tracking Introduction and Elimination Trends:
    - Counted the total number of vulnerabilities introduced over time.
    - Tracked how many vulnerabilities were eliminated across commits.
2. Measuring Resolution Patterns:
    - Calculated the number of fixed vulnerabilities for each severity level.
    - Measured the average resolution time (in commits) by tracking when vulnerabilities were introduced and eliminated.
3. Visualizing Trends:
    - Plotted introduction and elimination trends side by side for comparison.
    - Created summary tables showing introduction vs. elimination and resolution times across repositories.

Results:
The figures below provide insights into how vulnerabilities of different severities are introduced and resolved over time:

Vulnerability Trends Over Time (Left Plot): This figure shows the total number of vulnerabilities (High, Medium, and Low severity) present in the system over time (commit index). The lines remain mostly stable, indicating that new vulnerabilities are introduced at a steady rate rather than in sudden spikes.

Vulnerability Resolution Trends (Right Plot): This figure tracks the number of vulnerabilities eliminated (fixed) per commit. Negative values indicate cases where vulnerabilities were reintroduced, either due to rollbacks or new issues being introduced after previous fixes.

Fig 4: Vulnerabilities Analysis of Archive Box

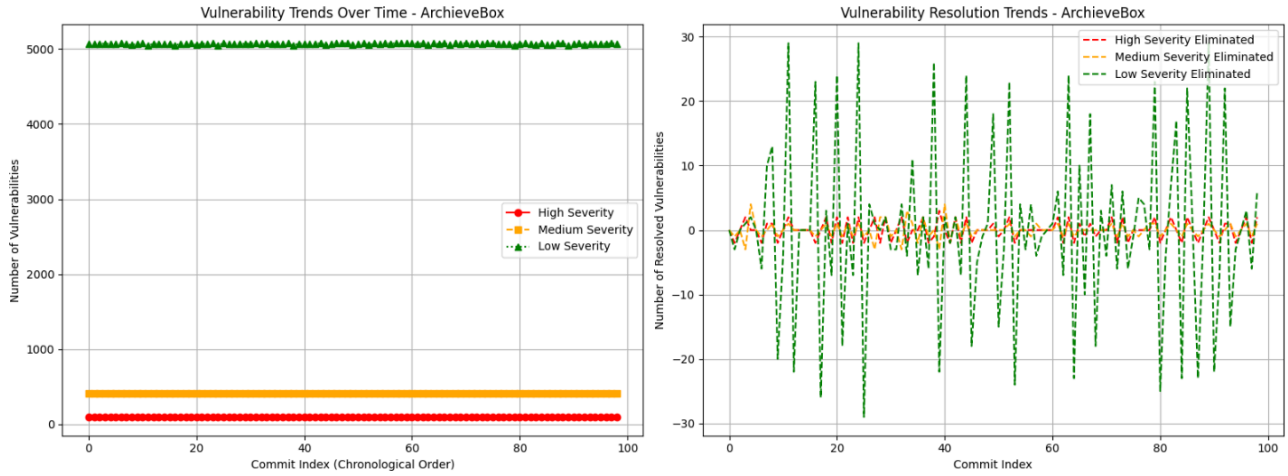


Fig 5: Vulnerabilities Analysis of Hosts

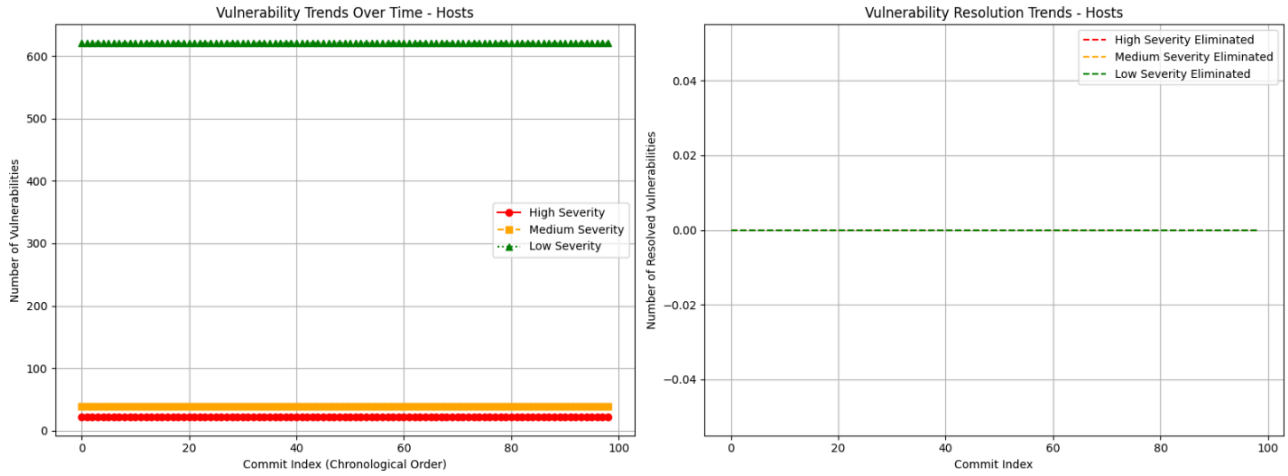
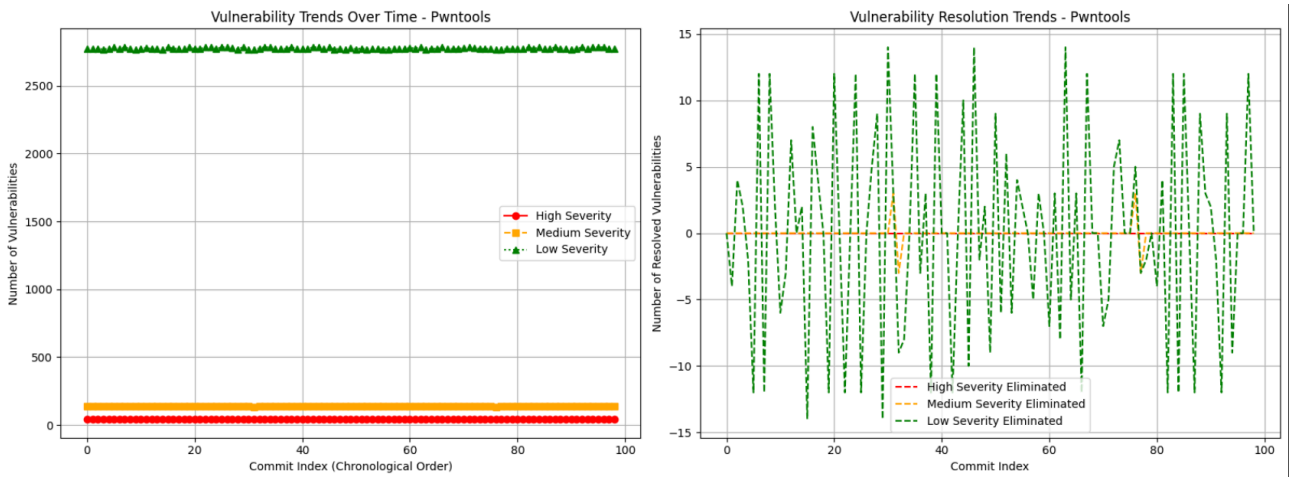


Fig 6: Vulnerabilities Analysis of Pwntools



Key Differences in Patterns Across Severity Levels

1. Low-Severity Vulnerabilities: Frequent Fluctuations & Bulk Fixes

- In ArchiveBox (Fig 4) and Pwntools (Fig 6), resolution trends fluctuate significantly, suggesting that low-severity vulnerabilities are often addressed in bulk rather than one by one.
- Continuous fluctuations imply an active management cycle where new low-severity issues are frequently introduced as development progresses.
- In Hosts (Fig 5), low-severity vulnerabilities remain constant, with no recorded eliminations—indicating a lack of security patches.

2. Medium-Severity Vulnerabilities: Rarely Eliminated, Addressed Individually

- In ArchiveBox (Fig 4), medium-severity vulnerabilities follow a steady resolution trend, suggesting they are handled individually rather than in bulk.
- In Pwntools (Fig 6), elimination of medium-severity vulnerabilities is infrequent, indicating slow security responses.
- In Hosts (Fig 5), there is no change in medium-severity vulnerabilities, implying no fixes or security updates in the commit history.

3. High-Severity Vulnerabilities: Persistent & Neglected

- In ArchiveBox (Fig 4), high-severity vulnerabilities are gradually eliminated, suggesting that fixes, when made, are intentional and deliberate.
- In Pwntools (Fig 6), high-severity vulnerabilities are almost never resolved, raising concerns about long-term security risks.
- In Hosts (Fig 5), no high-severity vulnerabilities were fixed, indicating a lack of critical security measures.

Table 1: Introduction vs. Elimination Summary (Across Repositories)

Repository	High Introduced	High Eliminated	Medium Introduced	Medium Eliminated	Low Introduced	Low Eliminated
ArchiveBox	9,723	56	40,673	41	501,880	477
Hosts	2,178	0	3,861	0	61,479	0
Pwntools	4,356	0	13,656	6	274,682	297

Key insights from Table 1:

- Vulnerabilities are introduced in large numbers but rarely eliminated.
- Hosts had zero fixes across all severity levels, indicating a complete lack of security maintenance.
- High- and medium-severity vulnerabilities persist longer, while low-severity vulnerabilities see more frequent fixes.

Table 2: Fix Time Analysis(Across Repositories)

Repository	Avg Fix Time (High)	Avg Fix Time (Medium)	Avg Fix Time (Low)
ArchiveBox	3.16 commits	3.35 commits	2.21 commits
Hosts	N/A (No fixes)	N/A (No fixes)	N/A (No fixes)
Pwntools	N/A (No fixes)	45.0 commits	2.37 commits

Key insights from Table 2:

- ArchiveBox resolves high- and medium-severity vulnerabilities at similar rates (~3 commits), while low-severity issues are resolved faster.
- Pwntools has delayed resolution for medium-severity vulnerabilities (45 commits), while low-severity issues are fixed relatively quickly (~2 commits).
- High-severity vulnerabilities remain unresolved in Pwntools and Hosts, posing critical security risks.
- The tables reinforce that vulnerabilities of different severities do not follow the same pattern of introduction and elimination.

Final Observations:

- Low-severity vulnerabilities are introduced and eliminated frequently, often in bulk.
- Medium-severity vulnerabilities persist longer and are eliminated far less frequently.
- High-severity vulnerabilities are rarely fixed, making them the most significant security risk.
- Hosts repository lacks any security fixes, confirming no vulnerability management.

(c) RQ3 (CWE coverage): Which CWEs are the most frequent across different OSS repositories?

Purpose:

This RQ aims to identify the most common CWEs (Common Weakness Enumerations) across different open-source repositories. Understanding CWE distribution helps prioritize security improvements and focus on recurring vulnerabilities that pose significant risks.

Approach:

To analyze the most frequent CWEs across repositories, we followed these steps:

1. Extract CWE Data:
    - Retrieved unique_cwes from each repository dataset (df1, df2, and df3).
    - Flattened the lists to collect all CWEs across commits.
2. Count CWE Occurrences:
    - Used the Counter function to count the occurrences of each CWE across all repositories.
    - Identified the top 10 most frequent CWEs in each repository separately.

Results:

The output to the code in RQ3 revealed the following top 10 CWEs for each repository:
ArchiveBox: CWE-[259, 327, 377, 330, 78, 79, 400, 80, 20, 22]
Hosts: CWE-[259, 327, 330, 78, 400, 20, 22, 502, 377, 605]
Pwntools: CWE-[259, 327, 295, 330, 78, 80, 20, 22, 502, 377]

Table 3: Top CWE Vulnerabilities in Open-Source Software: Prevalence and Security Risks

CWE ID	Name	Description
CWE-259	Hard-coded Password	Software has embedded credentials, risking unauthorized access.
CWE-327	Weak Cryptography	Uses outdated or broken encryption algorithms.
CWE-377	Insecure Temp File	Temporary files are created without security measures.
CWE-330	Weak Randomness	Predictable random values weaken security.
CWE-78	OS Command Injection	User input allows arbitrary system command execution.
CWE-79	Cross-Site Scripting (XSS)	Injected scripts can hijack sessions or steal data.
CWE-400	Resource Exhaustion	Uncontrolled resource use can lead to DoS attacks.
CWE-80	Basic XSS	Web input is not sanitized, enabling script injection.
CWE-20	Improper Input Validation	Poor input checks lead to security flaws.
CWE-22	Path Traversal	Malicious input accesses restricted directories.
CWE-502	Unsafe Deserialization	Untrusted data can execute malicious code.
CWE-89	SQL Injection	Malicious queries can expose or modify databases.
CWE-732	Excessive Permissions	Grants unnecessary access to sensitive data.
CWE-605	Port Rebinding	Multiple bindings to a port cause security issues.
CWE-703	Poor Error Handling	Improper error checks can crash or expose vulnerabilities.

These results indicate that certain CWEs (e.g., CWE-259, CWE-327, CWE-330) are prevalent across multiple repositories, suggesting that these weaknesses are widespread and require prioritized security attention.

---

3. Results and Analysis

Vulnerability Trends and Severity-Based Fixing Patterns

Our analysis of OSS repositories reveals distinct patterns in the introduction and resolution of vulnerabilities across different severity levels.

Introduction and Fixing of High-Severity Vulnerabilities (RQ1)

Across repositories, we observed varied security maintenance trends. ArchiveBox displayed frequent fluctuations in high-severity vulnerabilities, indicating an active development cycle where vulnerabilities are both introduced and fixed at a steady rate. Conversely, Hosts and Pwntools exhibited a static high-severity vulnerability count, suggesting an absence of security patches during the observed period. The persistence of vulnerabilities in these repositories underscores the need for more proactive security measures. On average, ArchiveBox fixed high-severity vulnerabilities within ~3 commits, while Pwntools and Hosts had no recorded fixes, raising concerns about long-term security risks.

Severity-Based Resolution Patterns (RQ2)

Vulnerabilities of different severities follow distinct elimination patterns:

- Low-severity vulnerabilities experience frequent introduction and bulk fixes, reflecting active but less urgent security management.
- Medium-severity vulnerabilities show a slower and more deliberate fixing process, often addressed individually rather than in bulk.
- High-severity vulnerabilities persist the longest and are rarely eliminated, posing critical risks, especially in Hosts and Pwntools.

Table 1 summarizes the number of introduced and eliminated vulnerabilities across repositories, while Table 2 highlights the resolution time. ArchiveBox effectively manages vulnerabilities, but the lack of fixes in Pwntools and Hosts suggests inadequate security measures.

Most Prevalent CWEs in OSS (RQ3)

Our findings identify the top CWEs across repositories, with CWE-259 (Hard-Coded Passwords), CWE-327 (Weak Cryptography), and CWE-330 (Weak Randomness) appearing most frequently. The widespread presence of these vulnerabilities suggests systemic issues in security practices, emphasizing the need for stricter cryptographic standards and secure credential handling.

Key Takeaways

- Active repositories like ArchiveBox frequently introduce and fix vulnerabilities, while others (Hosts, Pwntools) show stagnant security maintenance.
- High-severity vulnerabilities persist longer than medium and low-severity issues, indicating a need for prioritization in fixes.
- Critical CWEs, such as weak cryptographic practices and hard-coded passwords, remain prevalent, highlighting recurring security weaknesses in OSS project.

---

4. Discussion and Conclusion

Summary

In this lab,I analyzed security vulnerabilities across OSS repositories, examining their **introduction, resolution patterns, severity distribution, and CWE coverage**. Key findings include:

- **Frequent introduction and resolution of vulnerabilities in actively maintained projects**, while others exhibit stagnant security risks.
- **Low-severity vulnerabilities are frequently addressed**, whereas **high-severity vulnerabilities persist**, posing critical security concerns.
- **Common CWEs such as hardcoded passwords and weak cryptography appear across multiple repositories**, indicating systemic security challenges.

To enhance OSS security, **proactive vulnerability management, standardized security practices, and developer training** are necessary. Future work can explore automated solutions for prioritizing and mitigating high-severity vulnerabilities more effectively.

---

5. References

1. Common Weakness Enumeration (CWE) - Wikipedia – Overview of CWE, a standardized list of software and hardware weaknesses.
2. MITRE CWE Overview – Official documentation on CWE classification and tracking.
3. CWE Top 25 (2024) – List of the most critical software vulnerabilities in 2024.
4. Bandit - Python Security Linter (GitHub) – A tool for detecting security issues in Python code.
5. Bandit Documentation – Guide on using Bandit for static code analysis.
6. Chatgpt for debugging, refining the grammar and formatting of this report