TheAlgorithms/Python: Report

Team Members: Rohini Senthil
Uzair Mukadam
Shridhar Vilas Shinde

1. Introduction

1.1 Project Overview

TheAlgorithms/Python is an open-source library designed for learning, practicing, and understanding algorithms in Python. It offers a curated collection of algorithm implementations that serve as a reference, educational resource, and practical toolkit for both students and developers. The project has gained significant traction in the open-source community, with over 210,000 stars and 48,500 forks on GitHub, demonstrating its value to the programming community.

The library covers an extensive range of algorithmic domains, including:

- **Core Computer Science**: Data Structures, Algorithms, Sorting, Searching, Dynamic Programming
- **Mathematics**: Linear Algebra, Matrix Operations, Number Theory, Statistics
- **Cryptography & Security**: Ciphers, Blockchain, Boolean Algebra
- **Data Processing**: Data Compression, Digital Image Processing, Computer Vision
- **Advanced Topics**: Machine Learning, Neural Networks, Quantum Computing, Fuzzy Logic
- **Specialized Domains**: Physics, Genetics, Geodesy, Geometry, Graphics
- **Practical Applications**: Web Programming, File Transfer, Financial Calculations

This comprehensive testing effort focused primarily on the data structures directory, which serves as a foundational component of the entire library.

2. Initial State Assessment

2.1 Initial Coverage Metrics

Before the testing improvements were implemented, the project exhibited the following coverage characteristics:

- **Doctest Coverage**: 58.2%
- **Pytest Coverage**: 19%
- **Total Tests Passed**: 1,265

It is important to note that both doctest and pytest suites contained failing tests, primarily due to improper imports and syntax issues. This initial assessment revealed significant opportunities for improvement in test coverage and test quality.

doctest	rotate_to_the_right.py	21	0	76.2%
doctest	from_sequence.py	0	1	0.0%
doctest	circular_linked_list.py	1	0	92.5%
doctest	swap_nodes.py	38	0	77.3%
doctest	reverse_k_group.py	0	1	0.0%
doctest	floyds_cycle_detection.py	0	1	0.0%
doctest	doubly_linked_list.py	35	0	84.3%
doctest	print_reverse.py	31	0	87.2%
doctest	deque_doubly.py	0	1	0.0%
doctest	has_loop.py	0	1	0.0%
doctest	singly_linked_list.py	118	0	79.3%
doctest	bloom_filter.py	19	0	100.0%
doctest	hash_table.py	0	1	0.0%
doctest	double_hash.py	0	1	0.0%
doctest	quadratic_probing.py	0	1	0.0%
doctest	prime_numbers.py	9	0	38.9%
doctest	radix_tree.py	9	0	88.4%
doctest	trie.py	1	0	86.6%
doctest	min_heap.py	0	1	0.0%
doctest	heap.py	0	1	0.0%
doctest	heap_generic.py	0	1	0.0%
doctest	binomial_heap.py	0	1	0.0%
doctest	max_heap.py	0	1	0.0%
doctest	median_two_array.py	0	1	0.0%
doctest	kth_largest_element.py	15	0	86.7%
doctest	sparse_table.py	9	0	88.5%
doctest	find_triplets_with_0_sum.py	8	0	90.0%
doctest	permutations.py	2	0	85.7%
doctest	equilibrium_index_in_array.py	0	1	0.0%
doctest	index_2d_array_in_1d.py	17	0	89.5%
doctest	product_sum.py	0	1	0.0%
doctest	monotonic_array.py	0	1	0.0%
doctest	pairs_with_given_sum.py	3	0	71.4%
doctest	prefix_sum.py	13	0	92.3%
doctest	linked_queue.py	43	0	93.3%
doctest	double_ended_queue.py	134	0	89.7%
doctest	circular_queue_linked_list.py	30	0	93.2%
doctest	circular_queue.py	23	0	96.4%
doctest	priority_queue_using_list.py	55	0	45.9%
pytest	suffix_tree.cpython-39.pyc	-	-	19.0%
TOTAL		1265	25	58.2%

2.2 Requirements and Test Oracles

To guide the testing effort, we established clear functional and non-functional requirements along with corresponding test oracles.

Functional Requirements

The data structures were expected to support the following capabilities:

1. **FR-1: Traversal** - Enable traversal methods (e.g., in-order, pre-order, post-order for trees; forward/backward for linked lists)
2. **FR-2: Sorting** - Provide built-in or integrable sorting mechanisms where applicable
3. **FR-3: Search & Access** - Support efficient search and retrieval of elements

4. **FR-4: Error Handling** - Gracefully handle invalid operations (e.g., removing from an empty queue)
5. **FR-5: Insertion & Deletion** - Allow insertion and deletion operations at appropriate positions
6. **FR-6: Algorithm Integration** - Support direct usage with sorting, searching, or graph algorithms

Non-Functional Requirements

The implementation was also expected to meet these quality attributes:

1. **NFR-1: Testability** - Design should allow easy integration with unit testing frameworks
2. **NFR-2: Scalability** - Data structures should handle large datasets without significant performance degradation
3. **NFR-3: Performance/Efficiency** - Operations should be optimized for time and space complexity
4. **NFR-4: Reliability** - Functions should consistently return correct results under normal usage
5. **NFR-5: Maintainability** - Code should be modular, well-documented, and easy to update or extend

3. Testing Methodology

3.1 Tools and Frameworks

The following tools and frameworks were employed:

- **Unit Testing**: Python's unittest framework, pytest
- **Code Coverage**: coverage.py
- **Mutation Testing**: Mutatest
- **Static Analysis**: Ruff v0.14.3
- **Security Testing**: Bandit v1.9.1
- **Performance Testing**: Locust v2.42.6, Python threading

4. Detailed Testing Results

4.1 Unit Testing (Coverage Expansion)

What We Tested: Individual components across Wavelet Tree, Heap, Stack, Suffix Tree, Trie, Disjoint Sets, and Queues modules.

Result:

- Coverage increased from 58.2%/19% to **70.4%**
- Total passing tests: **1,859** (up from 1,265)
- Added **30+ new test cases** across modules

What We Fixed/Documented:

- Fixed failing tests caused by improper imports and syntax errors
- Added comprehensive edge case coverage for all major data structures
- New tests: Wavelet Tree (3), Heap (3), Stack (6), Suffix Tree (2), Trie (2), Disjoint Sets (12), Queues (2)

Key Insight: Many existing tests were failing not due to implementation issues but due to import problems, revealing the importance of proper test environment setup.

4.2 Unit Testing (Mocking & Stubbing)

What We Tested: Isolated components using mocks for Hash Table, Stack, and Disjoint Set to control dependencies and create deterministic test conditions.

Mocking Strategy:

- **Hash Table**: Mocked hash_function, balanced_factor, and values to simulate collisions and wrap-around
- **Stack**: Mocked deque, builtins.input, and builtins.print for interactive testing
- **Disjoint Set**: Mocked Node, find_set, and get_parent to isolate union-find logic

Result: Successfully validated **18 complex edge cases** that are difficult to test without mocking, including collision handling, interactive operations, and path simplification scenarios.

What We Fixed/Documented: Created comprehensive mocking strategy documentation with rationale for each mocking decision and test case purposes.

Key Insight: Mocking enables testing of edge cases and error conditions that are difficult to reproduce in real scenarios, significantly improving test reliability and determinism.

4.3 Mutation Testing

What We Tested: Evaluated test suite effectiveness using Mutatest on Disjoint Set, Stack, and Heap implementations.

Results:

Module	Targets	Sample	Killed	Survived	Score
Disjoint Set	16	3	0	3	0%
Stack	23	5	0	4	0%
Heap	60	10	20	6	76.9%

What We Fixed/Documented:

- Added 12 mutation-targeted tests for Disjoint Set
- Created comprehensive test_stack_mutation.py with 12 tests using strict identity checks
- Documented why mutations survive (type annotations, **main** blocks, language limitations)

Key Insight: Code coverage alone is insufficient to measure test quality. Mutation testing revealed that most surviving mutants were non-behavioral (type annotations, **main** blocks), indicating strong behavioral coverage despite low mutation scores for some modules.

4.4 Static Analysis & Code Smell Detection

What We Tested: Scanned 6 directories using Ruff v0.14.3 to identify code quality issues.

Result: Identified and fixed **200+ code quality issues**:

- **170+ assertion style** conversions (unittest to native assert)
- **Unused imports** removed (pytest, Mock, MagicMock)
- **Naming convention** fixes (uppercase to lowercase variables)
- **Whitespace issues** cleaned (trailing spaces, missing EOF newlines)
- **Trie Module**: 103 issues fixed
- **Suffix Tree Module**: 75 issues fixed

What We Fixed/Documented:

- Converted all unittest-style assertions to native Python assert statements
- Removed all unused imports and reorganized import blocks
- Renamed uppercase variables to lowercase (SRC→src, DST→dst)
- Cleaned all whitespace issues and added missing EOF newlines
- Renamed unused variables with underscore prefix

Key Insight: Static analysis revealed issues invisible to functional testing but significantly impacting code maintainability and readability. Consistency improvements make the codebase easier to understand and modify.

4.5 Integration Testing

What We Tested: Validated component interactions across three scenarios: Quadratic Probing (Hashing), Trie/RadixTree, and Sudoku Solver (Arrays).

Test Coverage:

- **Quadratic Probing**: 5 tests (collision resolution, rehashing, type safety)

- **Trie/RadixTree**: 8 tests (operation consistency, prefix sharing, compression)
- **Sudoku Solver**: 5 tests (full solve cycle, constraint propagation, failure handling)

Result: Discovered **critical bug in RadixTree**: find("") raises IndexError when attempting to access word[0] on empty string, even though insert("") works correctly.

What We Fixed/Documented:

- Documented RadixTree bug with clear reproduction steps and test case
- Validated all other integration points work correctly
- Confirmed Sudoku test failures were due to test suite issues, not implementation

Key Insight: Integration testing is essential for finding bugs that only manifest at module boundaries. Individual components may work perfectly in isolation but fail when combined due to unhandled edge cases.

4.6 System Testing

What We Tested: End-to-end workflows for Quadratic Probing, Trie, and Priority Queue implementations.

Test Coverage:

- **Quadratic Probing**: 4 tests (single insert, bulk insert, collision resolution, rehashing)
- **Trie**: 1 comprehensive test (word management, insertion, lookup, deletion with prefix preservation)
- **Priority Queue**: 7 tests (FPQ priority validation, EPQ duplicate handling)

Result: All system tests passed successfully, validating:

- Quadratic probing handles collisions and rehashing correctly
- Trie maintains prefix relationships (deleting 'apple' preserves 'application')
- FPQ enforces Priority-then-FIFO rule across all priority levels
- EPQ handles duplicate values correctly in FIFO order

What We Fixed/Documented: Created comprehensive system test suite covering realistic usage scenarios with documented expected behaviors for each workflow.

Key Insight: System testing validates that individual components work together correctly in realistic scenarios, ensuring end-to-end correctness beyond what unit and integration testing verify.

4.7 Security Testing

What We Tested: Ran Bandit v1.9.1 across 8 directories to identify security vulnerabilities and insecure coding practices.

Most Meaningful Result: Identified **160+ security issues**, primarily:

- **Assert misuse** (160+ instances in production code, CWE-703) - stripped in optimized mode
- **Weak randomness** (random.choice in Sudoku instead of secrets.choice, CWE-330)
- **Missing input validation** in hashing modules
- **No dependency audit** performed

Severity Breakdown:

- Low: 160+ assert statements that disappear in python -O
- Medium: Weak randomness, missing input validation
- High: None identified

What We Fixed/Documented: Documented all 160+ instances with recommended fixes:

- Replace assert with explicit exception handling: if not condition: raise ValueError(...)
- Use secrets.choice for cryptographic randomness
- Add type checks and input sanitization
- Run pip-audit for dependency vulnerabilities

Key Insight: Security testing revealed that coding practices matter as much as algorithmic correctness. Issues like assert misuse may not affect functionality in testing but create serious problems in production environments.

4.8 Performance Testing

What We Tested: Three testing types: Spike (Quadratic Probing), Load (Trie), and Stress (Doubly Linked List).

Test Results Summary:

Test Type	Target	Configuration	Key Result
Spike	Quadratic Probing	0→200 users, 2 min	80k insertions (64s), memory stable at 29.45 MB
Load	Trie	100 threads, 40k ops	**658,000 req/s**, 0.06s duration, 16.62 MB memory
Stress	Linked List	25k sequential ops	**O(N²) degradation** - 5k deletions took 10.58s

Result: Discovered **critical performance issue in Doubly Linked List**: **len** method traverses entire list (O(N)) instead of maintaining O(1) counter. Methods like delete_at_nth call len() inside loops, creating **O(N²) complexity**.

Performance Impact:

- Insertion: Expected O(1) → Actual O(N) → 5.51s for 10k operations
- Deletion: Expected O(N) → Actual O(N²) → 10.58s for 5k operations

What We Fixed/Documented:

- Documented spike test recommendation: increase spike size for better CPU metrics
- Confirmed Trie's excellent concurrent performance
- Identified and documented linked list bottleneck with recommended fix (maintain size attribute)

Key Insight: Design decisions have cascading effects. A simple choice (computing length vs. maintaining counter) can degrade complexity from O(N) to O(N²), creating severe performance problems under stress conditions.

5. Most Significant Issues Found

The comprehensive testing effort revealed several critical issues that require attention:

1. **RadixTree find("") Bug (IndexError)**: The find() method in RadixTree does not handle empty strings, causing an IndexError when attempting to access word[0] on an empty string. This represents a critical bug that could cause program crashes.
2. **Misuse of Assert Across Modules**: Widespread use of assert statements in production code creates a reliability issue, as these assertions are removed when Python runs in optimized mode (python -O), potentially masking validation failures in production environments.
3. **Weak Randomness in Sudoku Solver**: Use of random.choice instead of cryptographically secure random number generation (secrets.choice) represents a security concern if the solver is used in security-sensitive contexts.
4. **Lack of Input Validation in Hashing**: The hashing modules do not perform adequate type checking and input sanitization, which could lead to unexpected behavior or security vulnerabilities.
5. **Linked List Performance Issue**: The **len** method's O(N) implementation creates cascading performance problems, particularly in deletion operations that exhibit O(N²) complexity instead of the expected O(N).

6. Improvements Made

The comprehensive testing effort resulted in numerous improvements to code quality, test coverage, and documentation:

6.1 Test Coverage Improvements

- Increased doctest and pytest coverage from initial combined 58.2% and 19% to 70.4%
- Increased total passing tests from 1,265 to 1,859
- Added 30+ new test cases across multiple data structure modules
- Created comprehensive edge case tests for all major data structures

6.2 Test Quality Enhancements

- Strengthened test suites with identity checks (using 'is' instead of '==')
- Added comprehensive mocking tests for hash tables, stacks, and disjoint sets
- Created mutation-targeted tests to address specific logical gaps
- Standardized test assertion style by converting unittest-style assertions to native Python assert statements

6.3 Code Quality Improvements

- Removed 200+ code smells identified by static analysis
- Eliminated unused imports across multiple modules (pytest, Mock, MagicMock, call)
- Standardized variable naming conventions (renamed uppercase variables to lowercase)
- Cleaned whitespace issues (trailing whitespace, blank lines with spaces, missing EOF newlines)
- Reorganized import statements for better code organization

6.4 Maintainability Improvements

- Unified assertion style across all test files for consistency
- Improved code readability through consistent formatting
- Added comprehensive documentation for test strategies and methodologies
- Created detailed reports for each testing phase with clear recommendations

6.5 Documentation Improvements

- Documented all testing methodologies and strategies
- Created comprehensive reports for unit, integration, system, security, and performance testing
- Included clear recommendations for addressing identified issues
- Provided examples of test data preparation and execution strategies

7. Overall Quality Assessment

7.1 Reliability

The project demonstrates strong reliability characteristics:

- Comprehensive test coverage across unit, integration, and system levels
- High mutation detection rate (76.9% for heap, with surviving mutants primarily in non-behavioral code)
- Successful validation of end-to-end workflows
- Effective error handling in most data structures

However, reliability concerns remain:

- Widespread use of assert statements that are removed in optimized mode
- RadixTree bug that can cause program crashes
- Linked list performance issues under stress conditions

7.2 Maintainability

Maintainability has been significantly improved through:

- Static analysis fixes that enhance code readability
- Consistent code formatting and style adherence to PEP8
- Removal of unused code and imports
- Standardized test assertion style
- Comprehensive documentation

Areas for continued improvement:

- Further modularization of large functions
- Additional inline documentation for complex algorithms
- Consistent error handling patterns across modules

7.3 Performance

Performance characteristics are generally strong:

- Hash tables handle spike loads efficiently
- Trie demonstrates excellent concurrent operation performance (658,000 req/s)
- Most data structures exhibit expected time complexity for operations

Performance concerns:

- Linked list **len** implementation creates $O(N^2)$ complexity in deletion operations
- Opportunities for optimization in heavily used paths

7.4 Security

Security posture is acceptable for an educational library:

- No critical vulnerabilities identified
- Low-to-medium severity issues documented with clear remediation paths
- No use of dangerous functions or libraries
- Established baseline for ongoing security monitoring

Security improvements needed:

- Replace assert statements with proper exception handling
- Use cryptographically secure random number generation where appropriate
- Implement input validation in hashing modules
- Conduct regular dependency audits

8. Lessons Learned

8.1 Understanding Testing Levels is Critical

Different testing types reveal different issues. The RadixTree bug was only found during integration testing, not unit testing, proving that each testing level serves a distinct purpose that cannot be substituted.

8.2 Coverage Metrics Don't Equal Test Quality

Our testing revealed a stark reality:

- Disjoint Set: 100% coverage but 0% mutation score
- Heap: 70%+ coverage with 76.9% mutation score Coverage tells you what code runs; mutation testing tells you if tests actually verify behavior.

8.3 Integration Testing Finds the Critical Bugs

The most significant bugs (RadixTree empty string, linked list $O(N^2)$) were discovered through integration and performance testing, not unit tests. Real-world scenarios expose issues that isolated component testing misses.

8.4 Static Analysis Reveals Invisible Issues

200+ code quality issues (unused imports, naming violations, formatting) were invisible to functional testing but significantly impacted maintainability. Tools like Ruff automate enforcement of quality standards.

8.5 Assert Statements Are Production Time Bombs

160+ assert statements that work perfectly in testing completely disappear in production (python -O), creating a dangerous gap between test and production behavior. Use explicit exception handling instead.

8.6 Design Decisions Have Cascading Performance Effects

A simple choice (computing **len** vs. maintaining a counter) degraded linked list deletion from O(N) to $O(N^2)$, making it unusable at scale. Performance considerations must be part of initial design, not afterthoughts.

8.7 Python's Dynamic Nature Creates Testing Challenges

Type annotations aren't enforced at runtime, truthiness evaluation is permissive, and duck typing allows unexpected behavior. Tests must account for Python-specific characteristics that don't exist in statically typed languages.

8.8 Security Is About Practices, Not Just Vulnerabilities

Weak randomness, missing input validation, and assert misuse don't appear as "vulnerabilities" but create real production risks. Security testing validates coding practices, not just protection against attacks.

9. Team members and roles

The testing effort demonstrated effective collaboration:

- Clear division of tasks across team members
- Documented contributions for each testing phase
- Consistent methodology application across different modules
- Comprehensive reporting and knowledge sharing

10. Conclusion

This comprehensive testing initiative successfully evaluated TheAlgorithms/Python's data structures module across eight testing methodologies: unit testing, mocking and stubbing, mutation testing, static analysis, integration testing, system testing, security testing, and performance testing. We improved test coverage from 58.2%/19% to 70.4%, increased passing tests from 1,265 to 1,859, and fixed over 200 code quality issues. The testing revealed five critical issues: a RadixTree crash bug, 160+ assert statements that fail in production, linked list $O(N^2)$ performance degradation, weak randomness in Sudoku solver, and missing input validation in hashing modules. Each testing methodology contributed unique insights integration testing found the RadixTree bug, performance testing exposed complexity issues, security testing identified assert misuse, and static analysis improved code maintainability.