Exercise 2: Automated Testing & Coverage

This is an individual assignment. The final submission must be completed and submitted independently by each student.

Deadline: Nov 10, 11:59PM EST*. The deadline is sharp -- late submissions will not be accepted. Assignments must be submitted through GradeScope. We understand that circumstances may arise where you may need to submit the assignment late.* ***Seek approval by contacting TA Abhishek Varghese (avarghese@umass.edu) at least 24 hours in advance (unless it's a last-minute emergency and you cannot).*** *Medical conditions, religious or funerary events, university-related events (conference visit, athletic event, field trip, or performance), or extenuating non-academic reasons (military obligation, family illness, jury duty, automobile collision) that need extension will be accommodated with written documentation. Assignments or exams from other courses, interviews, or paper deadlines are not legitimate reasons for an extension.*

Objective

Using the programs you produced in Exercise 1, design and run automated tests and collect code coverage with automated tools. You'll (1) measure baseline coverage, (2) use LLMs to improve tests, and (3) analyze coverage vs. fault detection.

Scope & Prereqs

• Use the same problems you completed in Exercise 1.
• Keep your original prompts and solutions accessible (you'll reference them as needed).
• Languages: Java or Python (or both).
• Java → JaCoCo
• Python → pytest-cov

Part 1 — Baseline Coverage (30% – 6 points)

1. Set up automated coverage collection for your A1 solutions (Use the tests provided with your benchmark):
2. For each problem, report at least:
   - Number of tests passed
   - Line coverage and branch coverage (if the tool supports it for your language).
   - A one-line interpretation (e.g., "low branch coverage due to untested error path").
3. Include a single summary table across all problems (problem → line %, branch %, notes).

For the following parts you will only work with 2 set of (problem statement, benchmark test suite, LLM generated solution). Choose ones with room for improvement. Here's an example: Chose highest */%test - %branch-coverage| x %test* (You can use your own metric to choose the two problems)

Tip: If your Exercise 1 code is in multiple packages/modules, be explicit about what you measure (module/package name).

Part 2 — LLM-Assisted Test Generation & Coverage Improvement (50% – 10 points)

Use an LLM to generate or improve tests with the goal of increasing meaningful coverage.
For each of the 2 problems selected before:
1. Prompt the LLM to produce new or improved unit tests. Example: "Produce tests that increase branch coverage for conditions X/Y." (You can choose your own prompt)
2. Run coverage again with the new tests.
3. For two representative problems provide:
   - The prompts used to generate tests (paste in the PDF).
   - Before/after coverage numbers (line & branch) and a brief explanation of what changed.
   - A note on redundancy (did the LLM produce duplicate/near-duplicate tests? how did you de-dupe?).
4. Repeat 1-3 accumulating the LLM generated tests until convergence. Convergence criteria: 3 consecutive iterations should have increase in %coverage less than 3%. Aka Coverage(i) - Coverage(i-2) <= 3%. Measure convergence for branch coverage if your tool supports it, otherwise use line-coverage. (Try and find on your own why branch coverage is *usually* a better metric than line coverage, you can refer to slides and external sources – it won't be graded).

Include each iteration in your report. Keep the tests that you generate after each iteration so that the addition of tests from LLM is cumulative and the coverage only goes up.

Part 3 — Fault Detection Check (20% – 4 points)

Coverage alone isn't enough. Evaluate whether your test suite catches bugs.
For each of the two problems do one of the following:
- Seeded bug check: Introduce a small, realistic bug (off-by-one, wrong boundary, wrong exception handling, etc.) into your A1 solution and re-run tests. Did they fail?
- Buggy baseline comparison: If you have a known buggy version (e.g., an earlier failed Exercise 1 attempt), run your improved tests against it. Do they catch the bug?

Report:
- What bug you injected/compared against and why it's realistic.
- Whether your tests failed as expected and which test(s) caught it.
- A short conclusion linking coverage ↔ fault detection (e.g., "branch ↑ uncovered the else

path that exposed the bug").

Tooling Requirements (use automated tools)

- Python: pytest, pytest-cov, coverage (HTML + XML reports).
- Java: JaCoCo (Gradle jacocoTestReport or Maven jacoco:report).

Deliverables (same format as Exercise 1)

• Submit one PDF document on the course platform. The PDF must include:
• Baseline coverage table (per problem: line %, branch %, notes). (Part 1)
• Prompts used to generate/improve tests (paste for the two representative problems in Part 2).
• Before/after coverage comparisons and commentary (Part 2).
• Fault detection write-up for two problems (what bug, which tests caught it) (Part 3).
• A link to your GitHub repository, which must contain:
• Source code from Exercise 1, test files (baseline + improved), and any scripts/config for coverage.
• Generated coverage reports (HTML/XML) or instructions to reproduce locally.

Resources to help:

(Reach out to add more bullet points to this list)

1. Automated Coverage Collection using Python
   - Requirements : pip install pytest-cov
   - Documentation : pytest documentation, pytest-cov 7.0.0 documentation
   - Videos : https://youtu.be/6toeRpugWjI?si=KiCUo2Selb9CAyzs (Pytest-Cov Usage), https://youtu.be/EgpLj86ZHFQ?si=cAVztZeUn8FIKUbE (How to create tests in python),
   - Articles : What is Pytest Coverage & Generate Pytest Coverage Report | LambdaTest, How To Generate Beautiful & Comprehensive Pytest Code Coverage Reports (With Example) | Pytest with Eric (Article uses Pytest + Coverage.py, but pytest-cov simplifies the process)