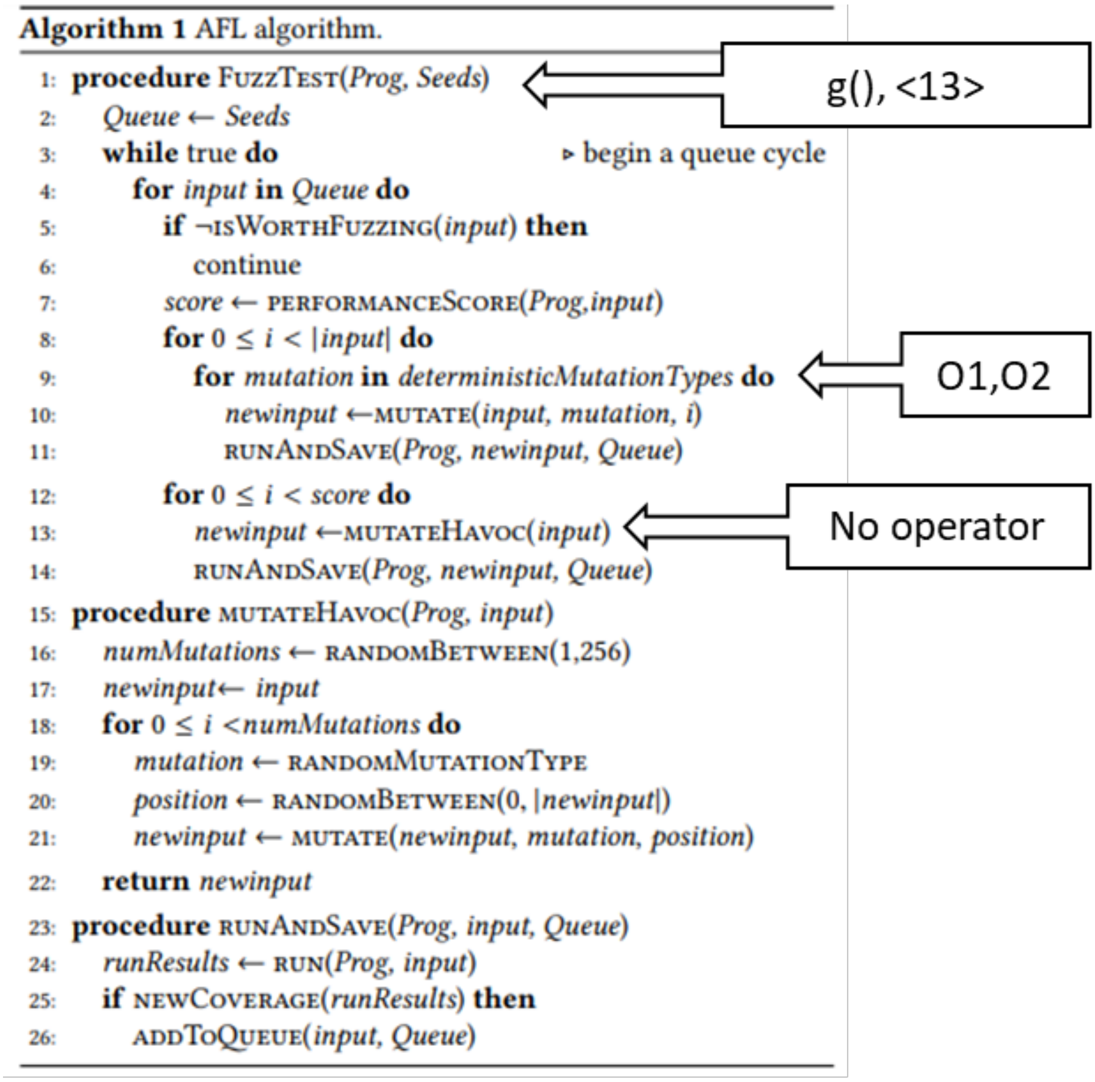
CS5351 Software Engineering
2024/2025 Semester B
Week 10: Exercises on Fuzzing by AFL and Metamorphic Testing

Ex 1. Fuzzing: Fuzzing aims to generate test cases automatically and quickly to expose the regions within the code base. It detects generic errors such as program crashes. The key component of a fuzzer is to find out whether a test case covers an interesting element (e.g., a new branch) in the program, and if this is the case, the test case is kept for future mutation (to generate new test cases).

Study the content on Page 1 and Page 2 of this exercise. **What is the sequence of test cases generated by AFL until it crashes the function g()?**

Algorithm 1 shows the AFL fuzzing algorithm. AFL is a coverage-guided fuzzer to be applied to test a function ***g*(int *x*)** using the following setting:

- The algorithm will **terminate if** running g() with a test case at line 24 causes **g()** to **crash.**
- The given seed sequence set is ⟨13⟩, i.e., *Seeds* = ⟨13⟩ at line 1 of Alg 1.
- There are two deterministic mutation operators. The algorithm applies O1 before O2 on the same seed.
    - (O1) decrement the input by 1. (e.g., computing 13 − 1 produces 12)
    - (O2) divide the input by the integer 2. Note that the division is an integer arithmetic operator. (e.g., Computing 13 / 2 produces 6)
- There are **no** nondeterministic mutation operators.
- Each value for *x* for fuzzing should be limited to the **range of 0 to 12**.
- For this exercise, in the algorithm, |*input*|, *mutate*(), *addToQueue*(), *newCoverage*(), *isWorthFuzzing*(), and *Run*(*g, input*) are defined as follows:
    - **|input|** is defined as 1 for all inputs.
    - **addToQueue(x, Y)** appends x to the current sequence Y.
    - **newCoverage(result)** returns *true* if the test case can execute any branches not yet executed by the test cases existing in Queue; otherwise, it returns *false*.
    - **isWorthFuzzing(y)** always returns *true*.
    - ***PerformanceScore*(*g, input*)** always returns −1 (no non-deterministic mutation operator)
    - ***Run*(*g, input*)** will execute *g*(*input*).
- The coverage achieved by executing *g*(*x*) on each input is shown in the following table. If a test case executes the branch statement indicates by the column title, there is a tick (✓) in a cell. For instance, when *x* = 5, the test case will execute the branches B5, B7 and B8. Note that not all branches in *g*() is shown in the coverage table shown on Page 2, but you can ignore the other branches in this exercise.



Algorithm 1 AFL algorithm.

```
procedure FuzzTest(Prog, Seeds)
  Queue ← Seeds
  while true do                                   ▹ begin a queue cycle
    for input in Queue do
      if ¬isWorthFuzzing(input) then
        continue
      score ← performanceScore(Prog,input)
      for 0 ≤ i < |input| do
        for mutation in deterministicMutationTypes do
          newinput ←mutate(input, mutation, i)
          runAndSave(Prog, newinput, Queue)
      for 0 ≤ i < score do
        newinput ←mutateHavoc(input)
        runAndSave(Prog, newinput, Queue)
procedure mutateHavoc(Prog, input)
  numMutations ← randomBetween(1,256)
  newinput← input
  for 0 ≤ i <numMutations do
    mutation ← randomMutationType
    position ← randomBetween(0, |newinput|)
    newinput ← mutate(newinput, mutation, position)
  return newinput
procedure runAndSave(Prog, input, Queue)
  runResults ← run(Prog, input)
  if newCoverage(runResults) then
    addToQueue(input, Queue)
```

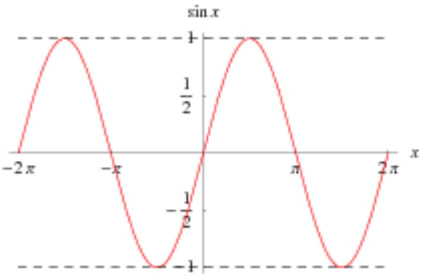
x's value	The branches in *g*() executed by each test case									Will *g*() crash?
	B1	B2	B3	B4	B5	B6	B7	B8	B9	
0	✓			✓	✓					No
1				✓	✓					No
2		✓		✓	✓					No
3					✓		✓	✓	✓	**Yes**
4					✓		✓		✓	No
5					✓		✓	✓		No
6		✓		✓	✓	✓				No
7					✓		✓	✓	✓	**Yes**
8					✓		✓	✓	✓	No
9		✓		✓	✓					No
10					✓					No
11	✓		✓		✓					No
12		✓		✓	✓	✓				No

Working Template
Name, Student ID:

Current test case	Mutated test case	Coverage achieved by the mutated test case	New Coverage found?	Seed queue after the execution of the mutated test case	crash g()?
		NIL		<13>	No
13	O1(13) = 12	B2, B4, B5, B6	Yes	<13, 12>	No
	O2(13) = 6				

Ex 2. Metamorphic Testing. Metamorphic testing aims to use the relationships among the outputs of multiple test cases to check whether a program may exhibit an anomaly in handling these test cases. The following exercise is taken from the Program Testing, Part

Consider a program P that aims to implement the mathematical sine function sin() double P(double i) where i is the degree.



- Traditional testing: E.g., P(30) = 0.5, P(90) = 1, P(180) = 0, P(32) = difficult to know
- Any metamorphic relation of P you can think of?