Software Testing, Quality Assurance & Maintenance—Lecture 11

Patrick Lam
University of Waterloo

February 9, 2026

Part I

Reducing Inputs

LaTeX errors

```
) (/usr/share/texlive/texmf-dist/tex/latex/epstopdf-pkg/epstopdf-base.sty
(/usr/share/texlive/texmf-dist/tex/latex/latexconfig/epstopdf-sys.cfg))
(/usr/share/texlive/texmf-dist/tex/latex/upquote/upquote.sty)
(/usr/share/texlive/texmf-dist/tex/latex/inconsolata/ot1zi4.fd)
Overfull \hbox (3.22516pt too wide) in paragraph at lines 84--84
[][]\OT1/cmr/m/n/9 Discussion: $\OT1/zi4/m/n/9 https : / / tex . meta . stackex
change . com / questions / 6255 / why-[]does-[]tex-[]require-[]such-[]elaborate
-[]mwes$|

! Package tikz Error: Giving up on this path. Did you forget a semicolon?.

See the tikz package documentation for explanation.
Type  H <return>  for immediate help.
 ...

l.97 \end{tikzpicture}

? █
```

To ask for help on StackExchange, need a Minimal Working Example.

Fixing a bug

1. need to reproduce the bug, so need a working example;
2. better yet: a *minimal* working example is easier to deal with.

MWEs and Fuzzing

Fuzzers produce large inputs.

When input contains extraneous context,
hard to understand what's happening.

Reducing an input

We'll show a way to reduce a failing input:

> "to identify those circumstances of a failure that are relevant for the failure to occur, and to *omit* (if possible) those parts that are not"

A Mystery

```
class MysteryRunner(Runner):
    def run(self, inp: str) -> Tuple[str, Outcome]:
        x = inp.find(chr(0o17 + 0o31))
        y = inp.find(chr(0o27 + 0o22))
        if x >= 0 and y >= 0 and x < y:
            return (inp, Runner.FAIL)
        else:
            return (inp, Runner.PASS)
```

Fails on some inputs. Can use `RandomFuzzer` to find a failure.

Fuzzing a Failure

```
def fuzz_mystery_runner():
    mystery = MysteryRunner()
    random_fuzzer = RandomFuzzer()
    while True:
        inp = random_fuzzer.fuzz()
        result, outcome = mystery.run(inp)
        if outcome == mystery.FAIL:
            break
    print (result)
```

This works and eventually finds a failing input.
(Manually, took me 6 tries.)

```
$ python3 mystery_runner.py
(%*50  1)-&7,;49:4?%:43*(-.
```

But why?

Part II

Manual Input Reduction

Divide and Conquer

Kernighan and Pike recommend:

> Proceed by binary search. Throw away half the input and see if the output is still wrong; if not, go back to the previous state and discard the other half of the input.

Does this work?

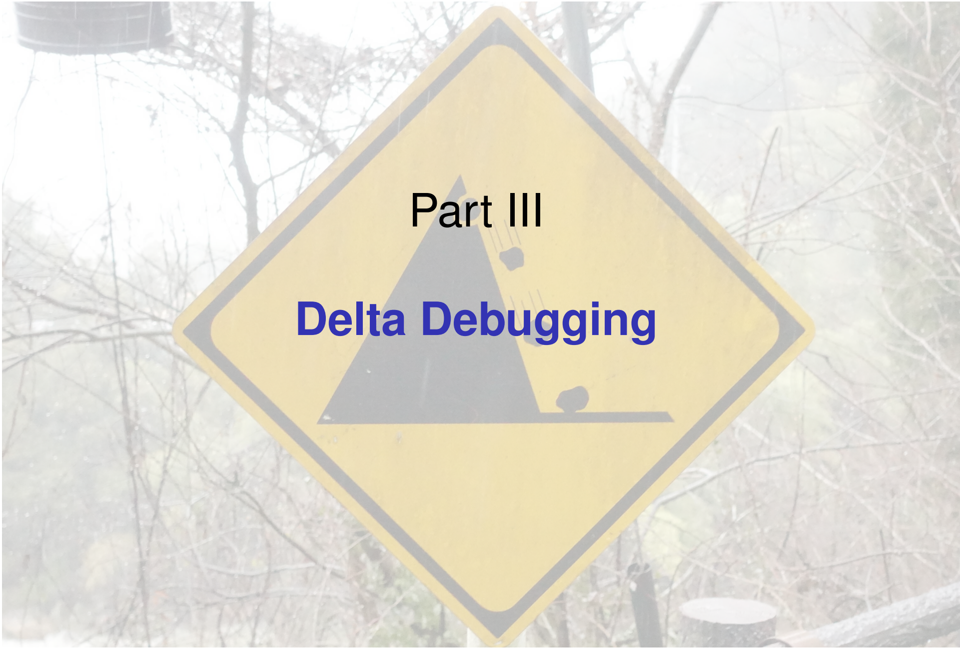
```
>>> from mystery_runner import *
>>> failing_input = "(%*50  1)-&7,;49:4?%:43*(-."
>>> mystery = MysteryRunner()
>>> mystery.run(failing_input)
('(%*50  1)-&7,;49:4?%:43*(-.', 'FAIL')
>>> half_length = len(failing_input) // 2 # integer division
>>> first_half = failing_input[:half_length]
>>> mystery.run(first_half)
('(%*50  1)-&7,', 'FAIL')
```

Progress! Halved the original input.

Failing at Failing

```
>>> quarter_length = len(first_half) // 2
>>> first_quarter = first_half[quarter_length:]
>>> mystery.run(first_quarter)
(' 1)-&7,', 'PASS')
>>> second_quarter = first_half[:quarter_length]
>>> mystery.run(second_quarter)
('(%*50 ', 'PASS')
```

Same trick doesn't work again.
The code says it's looking for two characters, but in our test case, the characters aren't in the same quarter.



Part III

Delta Debugging

A Change in Perspective

We tried direct binary search—didn't work.

Delta debugging is another way.

We instead *remove* smaller and smaller parts of the input, and see if it still fails.

Intuitively: more likely to keep the brokenness.

Example: Removing Quarters 1

Let's start with the first quarter.

```
>>> quarter_length=len(failing_input)//4
>>> input_without_first_quarter=failing_input[
                                    quarter_length:]
>>> mystery.run(input_without_first_quarter)
(' 1)-&7,;49:4?%:43*(-.', 'PASS')
```

PASS, so must keep 1st quarter.

Example: Removing Quarters 2

Now for the second quarter.

```
>>> input_without_second_quarter=failing_input[:
                                   quarter_length]+
                                   failing_input[
                                   quarter_length*2:]
>>> mystery.run(input_without_second_quarter)
('(%*50 ,;49:4?%:43*(-.', 'PASS')
```

We knew this already:

must keep the first half.

We also know we can discard the second half, but let's see.

Example: Removing Quarters 3, 4

```
>>> input_without_3rd_quarter=
     failing_input[:quarter_length*2]+failing_input[
                                           quarter_length*3:]
>>> mystery.run(input_without_3rd_quarter)
('(%*50  1)-&7?%:43*(-.', 'FAIL')
>>> input_without_4th_quarter=failing_input[:quarter_length*3]
>>> mystery.run(input_without_4th_quarter)
('(%*50  1)-&7,;49:4', 'FAIL')
```

This doesn’t tell us anything new,
but we’re sort of following the algorithm.

(The algorithm doesn’t actually quite work like this.)

Infrastructure for Reducing

An abstract base class that doesn't really do anything.

```python
class Reducer:
    def __init__(self, runner: Runner, log_test: bool = False)
                                                   -> None:
        # ...

    def test(self, inp: str) -> Outcome:
        # ...

    def reduce(self, inp: str) -> str:
        # here, non-real (abstract) impl
```

Caching

We can cache results:

```
class CachingReducer(Reducer):
    def test(self, inp):
        if inp in self.cache:
            return self.cache[inp]

        outcome = super().test(inp)
        self.cache[inp] = outcome
        return outcome
```

Actually reducing 1

Here is the outer loop for delta debugging.

```python
class DeltaDebuggingReducer(CachingReducer):
  def reduce(self, inp: str) -> str:
    self.reset()
    assert self.test(inp) != Runner.PASS

    n = 2     # Initial granularity
    while len(inp) >= 2:
      start = 0.0
      subset_length = len(inp) / n
      some_complement_is_failing = False
      # inner loop goes here
```

We initialize `n` to specify that we divide the input into halves at first.

Also, we set the subset length to the current input length, divided by `n`.

Actually reducing 2

Now the inner loop:

```
        # remove chunks of size len(inp)/n
        while start < len(inp):
            complement = inp[:int(start)] + \
                inp[int(start + subset_length):]
            if self.test(complement) == Runner.FAIL:
                # save the failing test, decrease n
                inp = complement
                n = max(n - 1, 2)
                some_complement_is_failing = True
                break
            start += subset_length
        if not some_complement_is_failing:
            # all subtests pass, get half-as-small chunks.
            if n == len(inp):
                break
            n = min(n * 2, len(inp))
    return inp
```

Running the delta debugger

```
dd_reducer = DeltaDebuggingReducer(mystery, log_test=True)
dd_reducer.reduce(failing_input)
```

and there is an example run in the *Fuzzing Book*, which I'll show excerpts from:

```
Test #1 ' 7:,>((/$$-/->.;.=;(.%!:50#7*8=$&&=$9!%6(4=&69\':\'<3+0-3.24#7=!&
Test #2 '\'<3+0-3.24#7=!&60)2/+";+<7+1<2!4$>92+$1<(3%&5\'\'>#' 49 PASS
Test #3 " 7:,>((/$$-/->.;.=;(.%!:50#7*8=$&&=$9!%6(4=&69':" 48 PASS
Test #4 '50#7*8=$&&=$9!%6(4=&69\':\'<3+0-3.24#7=!&60)2/+";+<7+1<2!4$>92+$1
Test #5 "50#7*8=$&&=$9!%6(4=&69':<7+1<2!4$>92+$1<(3%&5''>#" 49 PASS
Test #6 '50#7*8=$&&=$9!%6(4=&69\':\'<3+0-3.24#7=!&60)2/+";+' 48 FAIL
...
Test #23 '(460)' 5 FAIL
Test #24 '460)' 4 PASS
Test #25 '(0)' 3 FAIL
Test #26 '0)' 2 PASS
Test #27 '(' 1 PASS
Test #28 '()' 2 FAIL
Test #29 ')' 1 PASS
'()'
```

Solving the mystery

Answer: system fails on an input with a `(` and then a `)`.

Delta debugger commentary

Wouldn't want to do this manually on this input:
random input is harder to understand than a human-generated one.

Assuming that the system is deterministic, we can run the algorithm and get the answer.

Also assume: test cases can run quickly enough that we can afford dozens of iterations.

These are the same conditions as for fuzzing to work well.

Commentary continued

Implementation checks that the initial test case does fail.

Delta debugging is best-case $O(\log n)$ and worst-case $O(n^2)$.

Minimality

We get a 1-minimal test case:
removing any character is guaranteed to not fail.

In the example, we see that the single-paren cases pass.

This is a local minimum: might be some other smaller test case that one would reach with different choices.

Advantages of minimality

- reduces cognitive load for the programmer: no irrelevant details, easier to understand what’s happening.
- easier to communicate:
  “`MysteryRunner` fails on ”`()`””” vs
  “`MysteryRunner` fails on 4100-character input (attached)”
- helps identifying duplicates (to some extent— assuming failure has a single cause).