CS 121
Software Engineering

Debugging

Introduction

- Sometimes, rarely, every once in a while, you might make a mistake when you write code
  - Then, a test case will fail
    - (Because whenever you notice a bug, you'll write a test case for it!)
  - What do you do next? You *debug* the problem and fix it

- Apocryphal story: The term *bug* was coined when a moth stuck in a relay in the Harvard Mark II was found to be the cause of a glitch
  - Actually, the term dates back to at least the late 1800s

Terminology

- Bug, fault, defect, error, failure
  - All ways of saying code doesn't do what it is supposed to
  - Sometimes people make fine-grained distinctions
    - E.g., was it a mistake in writing the code? A misunderstanding by the software engineer? An incorrect specification? Etc.
  - Probably not worth obsessing over the differences

- Useful terminology: *Root cause* of a problem
  - Sometimes we observe a failure at one place in the code, but the problem is actually someplace else
    - The *root cause* is the thing we actually need to correct to fix the bug
  - Like the difference in medicine between a *symptom* and the underlying *disease*

Root Cause Example

```
void cause() {
  String x = foo(null);
}
int foo(String s) {
  List<String> l  = new List<>();
  l.add(s);
  Map<Integer, List<String>> m = new Map<>();
  m.put(42, l);
  // a bunch more computation
  String p = m.get(42).get(0);
  return p.length();
}
```

- `NullPointerException` raised on last line of `foo`
  - But the bug was actually passing in `null`!
    - Symptom of the bug appears far away from the cause
    - "Far away" could mean in terms of data flow, control flow, or both

Debugging Tools

- If the symptom of bug is close to root cause, almost any debugging approach will do
  - E.g., think about code near where exception raised
  - In practice, most bugs fall into this category!
- For more complex bugs, often need more insight about what is happening during program execution
  - First approach: Add print statements!
    - Easy to do, works really well, can be used is many environments
  - Improvement: add a logging facility to direct debugging info someplace besides `stdout`, and incorporate a *debug level*
    - E.g., `fatal`, `error`, `warn`, `info`, `debug`, …
      - `Logger.log(Logger.debug, “Class#m, x = “ + x);`
    - When run at a given log level, prints log messages at that level and higher

Debuggers

- Sometimes printing debug info isn't enough
  - Might not know what info to print, and can't print everything
- Many languages have *debuggers*
  - Tools that let you launch a program and control its execution
  - Typical operations:
    - Set breakpoint: indicate where execution should pause
    - Run/continue: Execute program until it hits the next breakpoint
    - Step: Execute one program statement
      - Depending what the source looks like, this could do many things!
    - Step into: At a method call, do the call and break at method's first statement
    - Step over: At a method call, execute method call as if it were a single step
    - Step out: In a method call, execute the first of the call and break after return
    - Print: Print the value of a variable (possibly in a different stack frame)
    - Watch: Break when a given value changes (usually expensive!)

Java Debugging Interface (JDI)

- Java has a generic facility for supporting debuggers
- Debuggers can connect to a VM or launch a VM under debugger control
  - Once connected, debugger can issue usual debugging commands
  - Supports some Java-specific stuff
    - Watch when threads started/stopped
    - Watch method entry/exit
    - Watch when `synchronize` acquires/releases lock
    - and more!

Time Travel Debuggers

- Common debugging scenario:
  - Step through the program, reach some execution time *t*, and figure something out
  - To diagnose further, must know what happened at time *t'* < *t*
    - Can't go back in time to check! Need to launch program from the start and step through again until we reach *t'*
- Radical solution: time travel debugging
  - Debugger with new command: jump to a given execution time
- These exist and really work!
  - Often work by *checkpointing* program state at various times and then *replaying* from there to reach desired time
  - However, slow and require a lot of resources
    - E.g., have to record a lot of info, e.g., syscalls

Debugging as Experimentation

- When bugs are more complex, helps to think of debugging as applying the scientific method
  - *Hypothesize* something about the program
    - E.g., a bug's cause, or something that might partially explain the bug
  - *Predict* consequences of the hypothesis
    - E.g., if the hypothesis holds, then `x` would be `42` on line 207 in `bar`
  - *Test* the prediction with an experiment
    - E.g., run the program and examine value of `x` on line 207
  - *Refine* hypothesis or create new hypotheses as needed
    - E.g., `x` is actually 39 on line 207, what hypothesis would explain that?
  - *Draw conclusion* of the root cause of a bug based on results of experiments
    - E.g., method `foo` passed `39` to method `bar` instead of `42`

Good Experimental Procedure

- When experiments get complicated, be systematic
  - Keep a lab notebook!
  - Write down each hypothesis, what experiment you did, and what happened
  - Use your notes to
    - Avoid repeating experiments you already did
    - Ensure you are making progress toward finding the root cause
      - Think of each experiment as pruning the set of possible causes until only one is left
    - Find gaps in the experiments you've done so far
    - Review experiments that seem to have contradictory or surprising results
- Be cautious about modify program during debugging
  - Might invalidate your prior experiments!
  - Best done once you've found the root cause…

Simplifying Tests

- Suppose we have a test case that exhibits a bug
- What part of the input is actually important?
  - Suppose test comes from a bug report from Q&A or the field
  - Test case could have many irrelevant details
    - "The bug happens when I launch the app, click Next, then Send, then Next, but then I hit the back button and click Submit, and then Back, and thn Next, then I enter the dog's name as "Fluffy," I spin my phone around, and it's 7:33pm on a Tuesday under the full moon, and the app crashes."
- Common debugging activity is *simplifying tests*
  - Try to find the shortest, simplest test case that exhibits bug
  - Helps restrict the hypothesis space for root causes
  - Makes tests faster to run, because they're shorter
  - Makes tests more orthogonal to each other

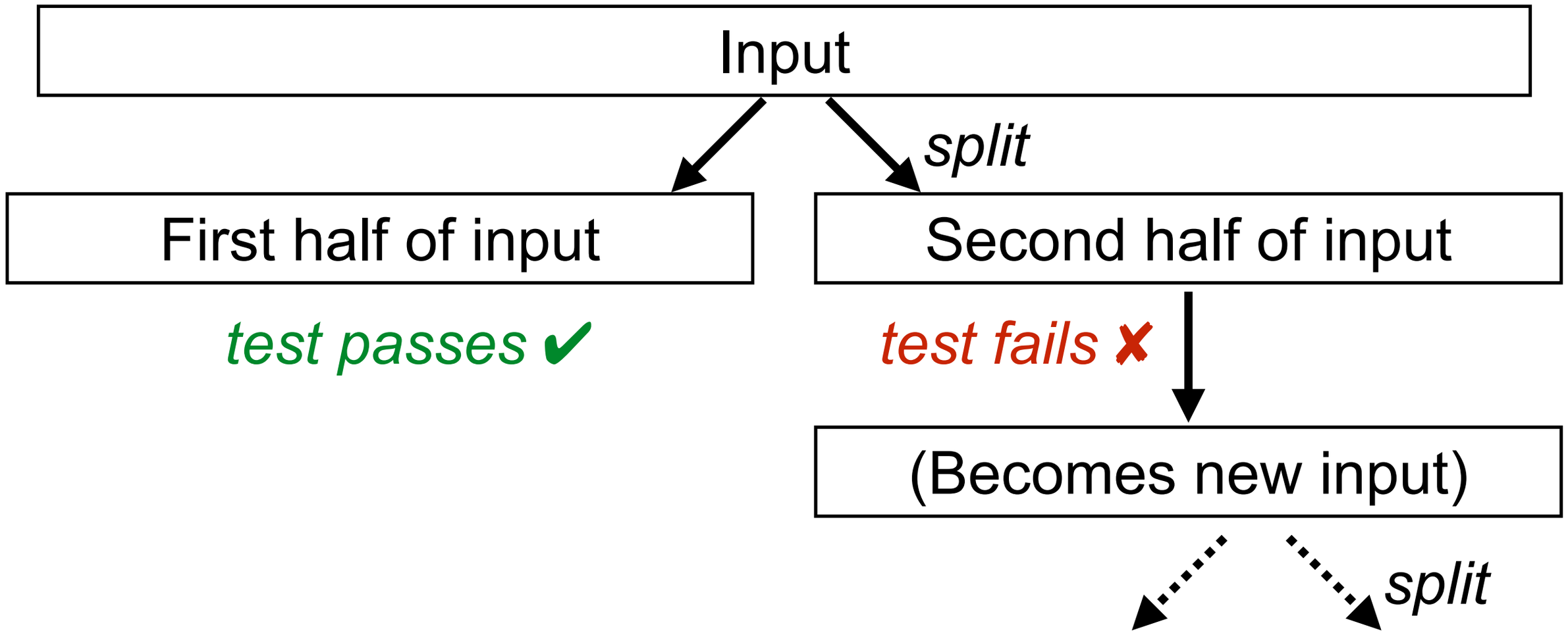
Simplifying Tests (cont'd)

- Simplify tests by removing *circumstances*
  - Cut some part of the failing test case
  - Rerun the test and see if it fails again
- But be careful!
  - Want test failure due to the same root cause
    - But since we don't know what that is, may be hard to preserve
  - E.g., might change the test case so that it triggers a different bug in the code
    - This is probably not so bad; double-check by running original, unsimplified test after fix
  - E.g., might change the test case so that it does something invalid, like violate a method's precondition
    - Then you'll waste a lot of debugging time figuring this out

Delta Debugging

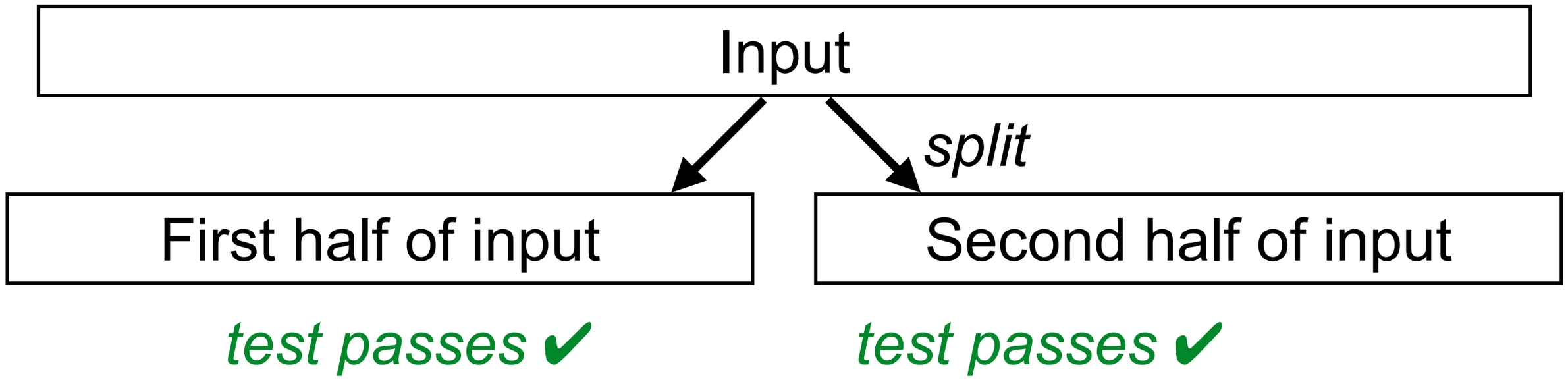
- If part of a test is a really big input, simplify it using a binary search-like process called *delta debugging*
  - Cut input in half, testing each of the halves
  - If one half has the same bug, throw away the other half and repeat
  - If neither half has the bug, increase granularity by removing quarters instead of halves, and repeat
- Illustrations on next slides
  - For full details, see Zeller, *Yesterday, My Program Worked. Today, It Does Not. Why?* FSE 99

Splitting the Input



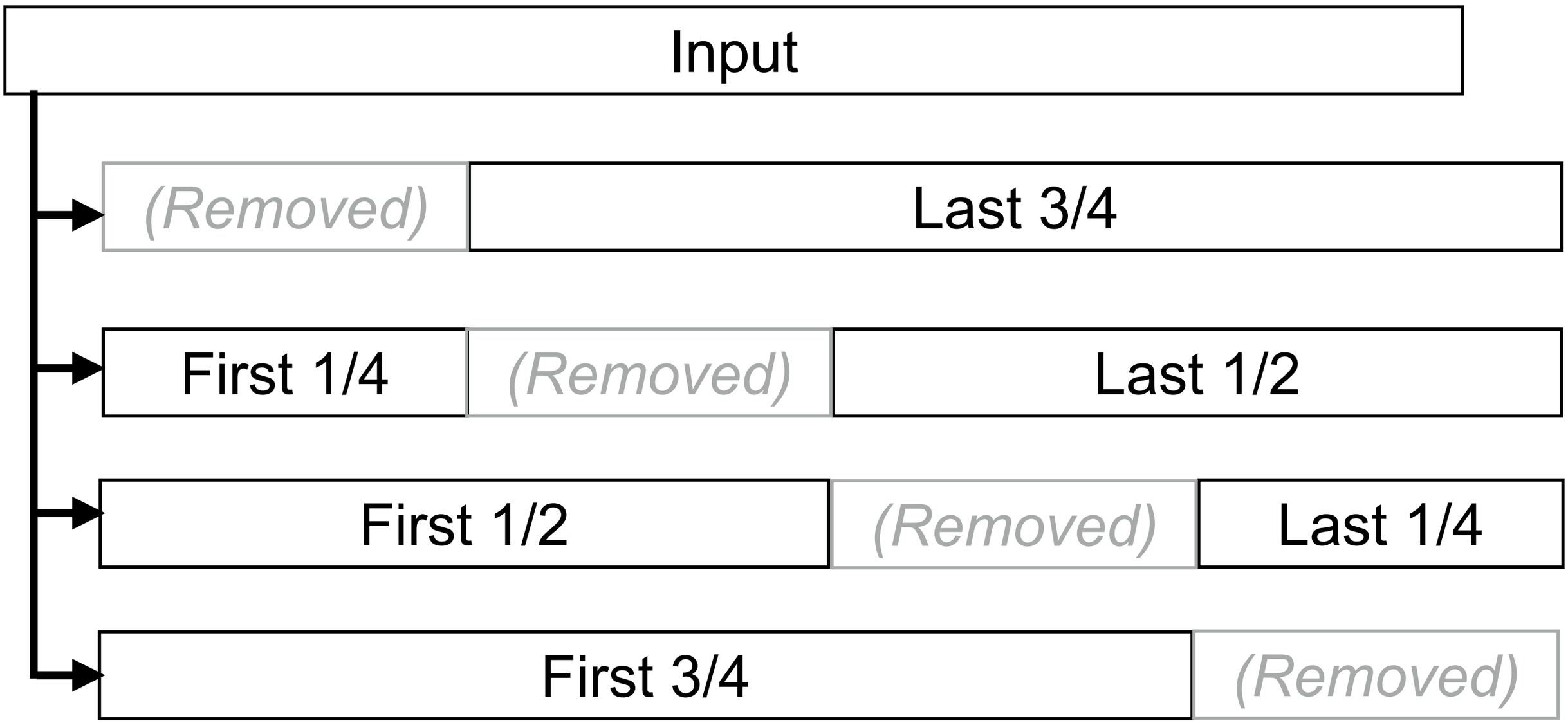
- Small input that fails becomes input for next round of delta debugging input simplification

Increasing Granularity



- Since all splits pass test, try removing smaller chunks
  - Instead of removing 1/2, removing 1/4
  - If all tests pass with removing 1/4, remove 1/8 instead, etc.
  - Stop increasing granularity when input can’t be divided more

Example: Removing 1/4 of Input



- If all splits pass test, try splitting into smaller chunks
  - Go from 1/2 to 1/4 to 1/8 etc.
  - Stop increasing granularity when input can't be divided more

Fixing the Bug

- Before you put in a fix for a bug, remember:
  - *Add a test case that exhibits the bug*
  - Consider adding test cases related to some of the other hypotheses you tested
- What if you put in a fix, but the test case still fails?
  - Maybe you didn't actually fix the bug?
  - Maybe there was a second bug in the code that's still there?
- What if you put in a fix, and another test case fails?
  - Perhaps you uncovered another bug?
- Might your code have multiple copies of the bug?
- What if the bug is deep, and fixing it would require a major design change?