EN.601.422 / EN.601.622

Software Testing & Debugging

Testing vs. Debugging

- So far: Testing
  - Look for inputs that cause failures
    - Coverage criteria
    - Test generation
    - Test Oracle
- Program fails, now what? → **Debugging**
  - Failures are typically discovered by
    - Tests
    - Real user

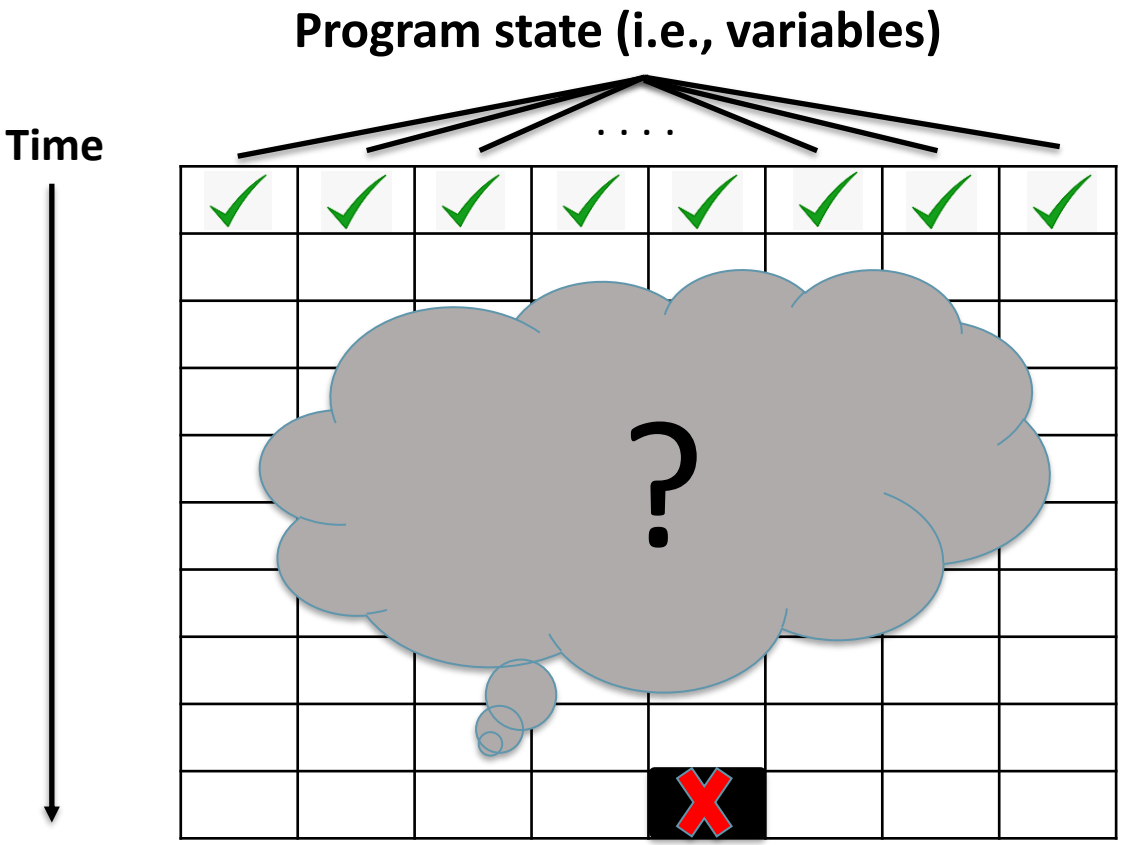
Six Stages of Debugging

1. That can’t happen.
2. That does not happen on my machine.
3. That should not happen.
4. Why does that happen?
5. Oh, I see.
6. How did that ever work?

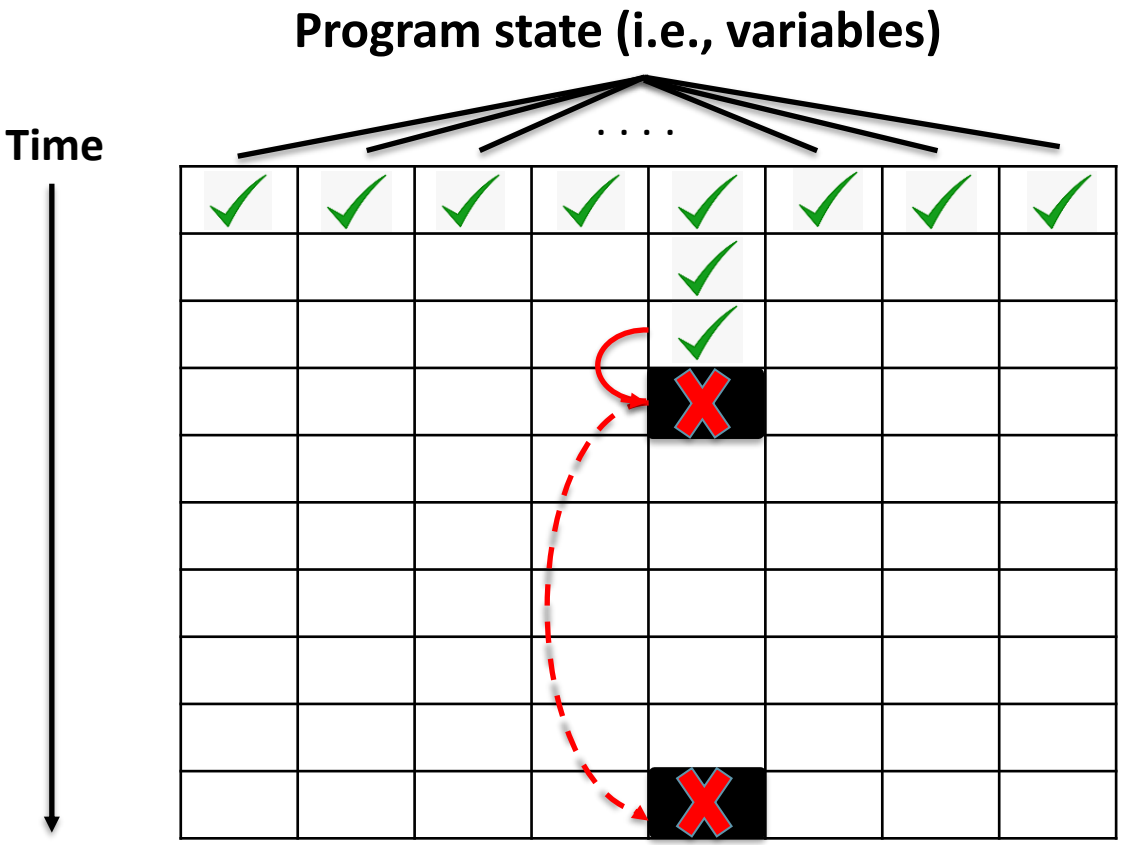
Debugging Steps

- Debugging Steps:
  1. Reproduce the error, understand
  2. Isolate and Minimize (shrink)– Simplification
  3. Eyeball the code, where/what could it be? Reason backwards
  4. Devise and run an experiment to test your hypothesis
  5. Repeat 3 and 4 until you understand what is wrong
  6. Fix the Bug and Verify the Fix
  7. Create a Regression Test

Search in Time and Space



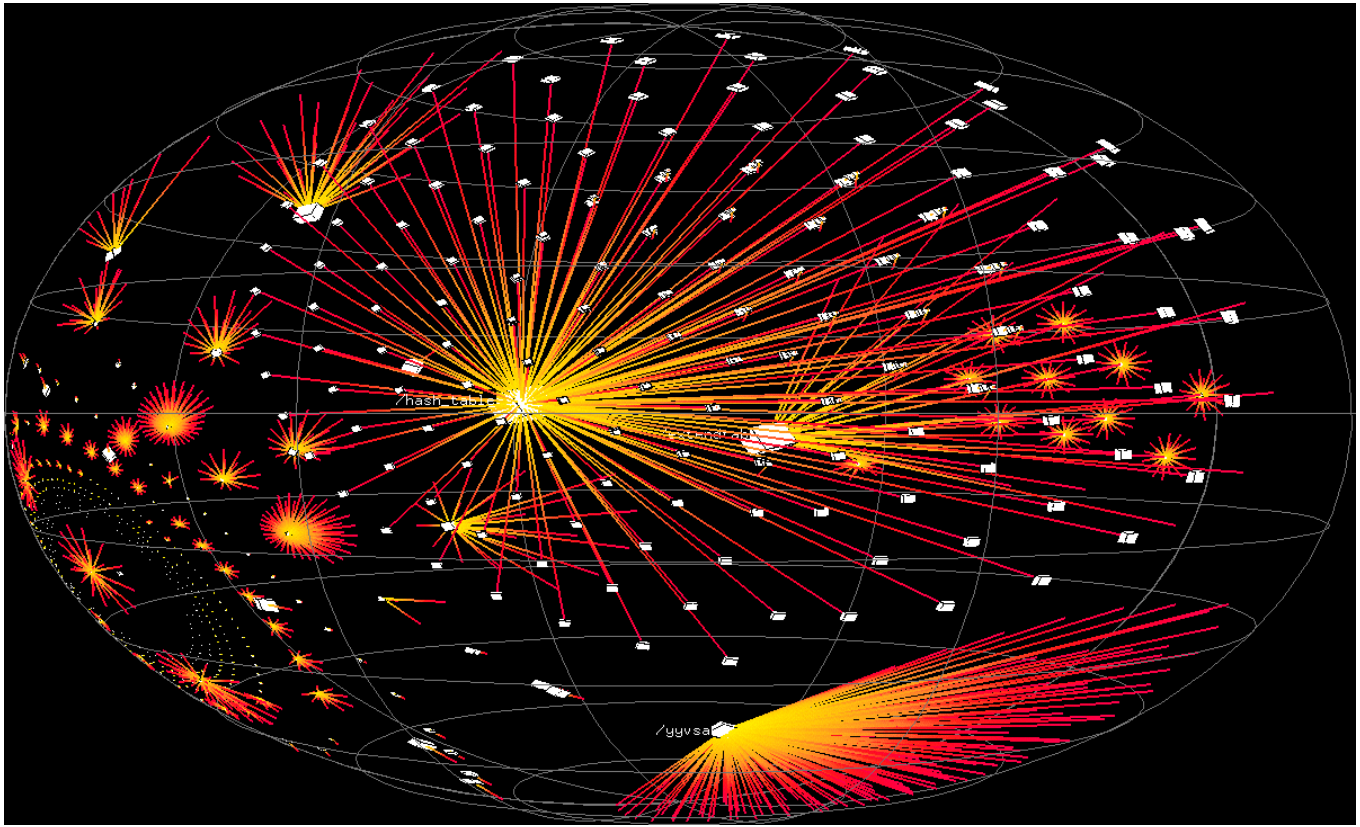
The Fault!



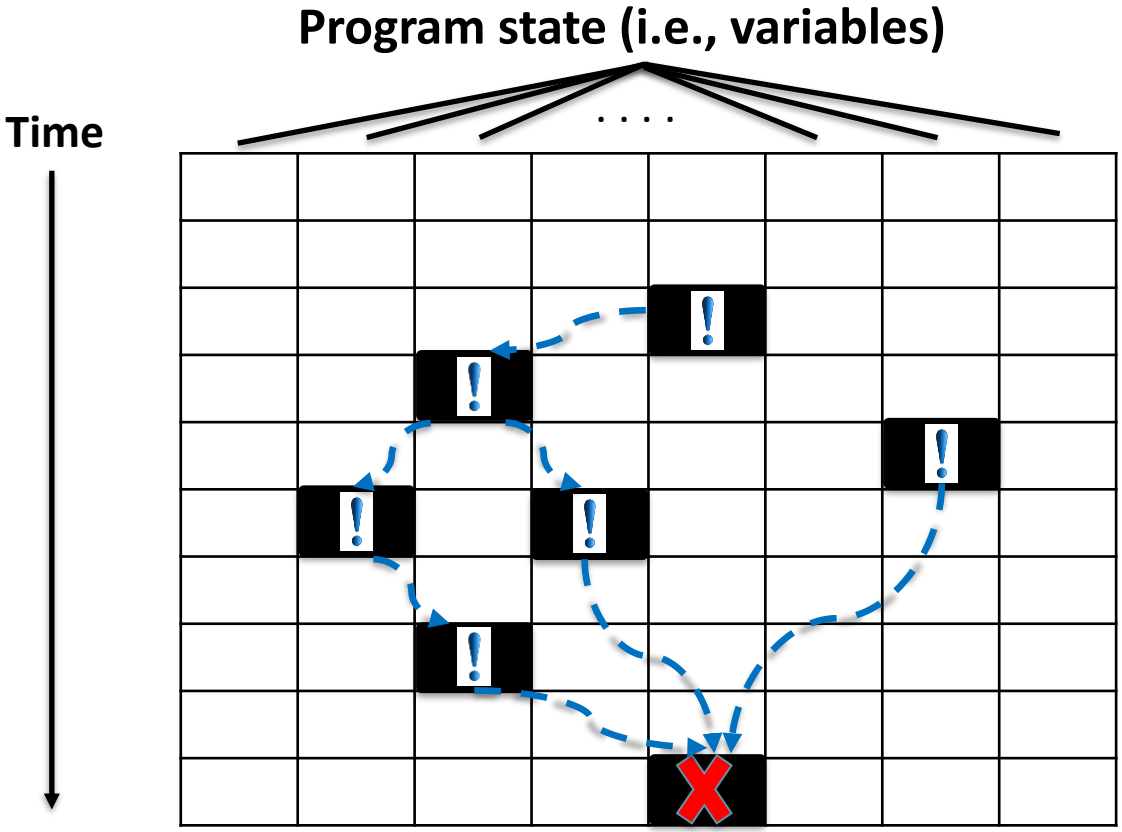
Debugging

- Debugging is a search in space (a given state of the program) and time (all the states that the program goes through) to find and resolve faults in a computer program
  - Each single program state may involve a large number of variables
  - A program may pass through millions of states before failure occurs
- This may seem like searching for a needle in endless rows of haystack

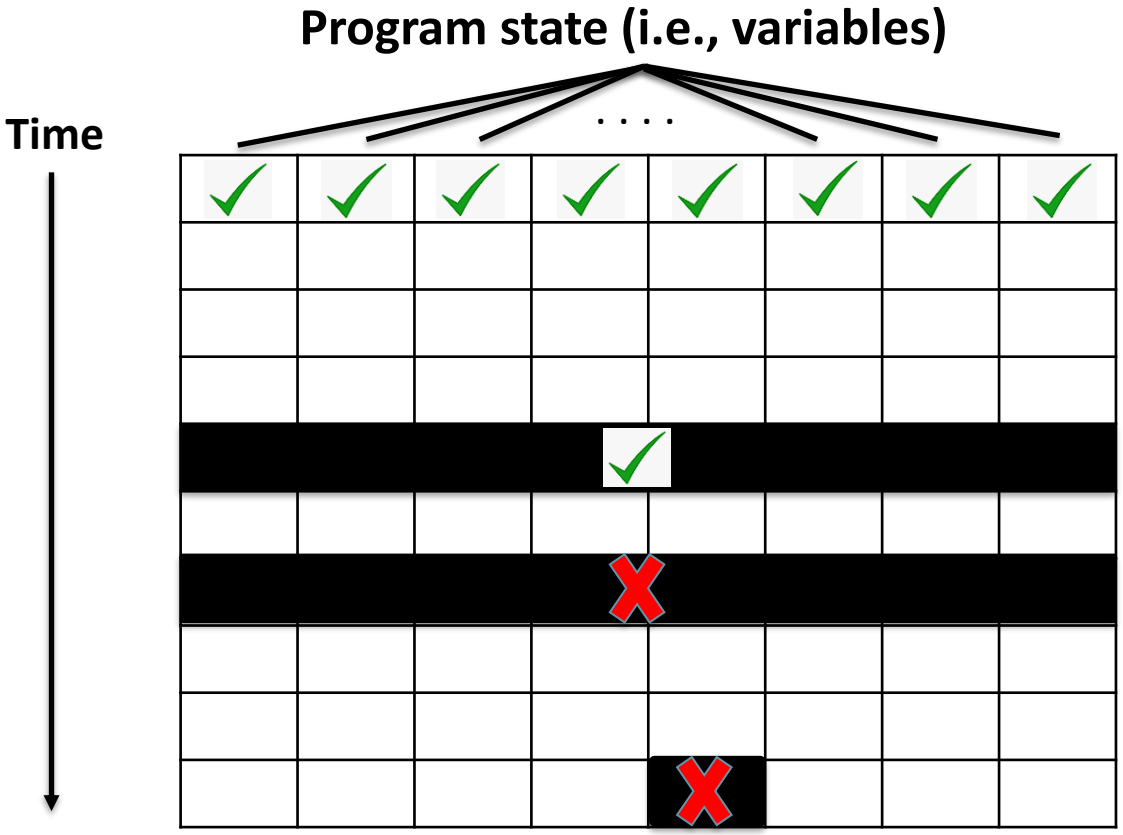
Program State Can Be Huge



Finding the Origins



Observing Transitions of States



How Failures Come To Be

- A failure comes to be in three stages:
  - The programmer makes a *fault* by creating a *defect* in the code
  - The defect causes an infection (i.e., incorrect state or error)
  - The infection causes a failure -- an externally visible error

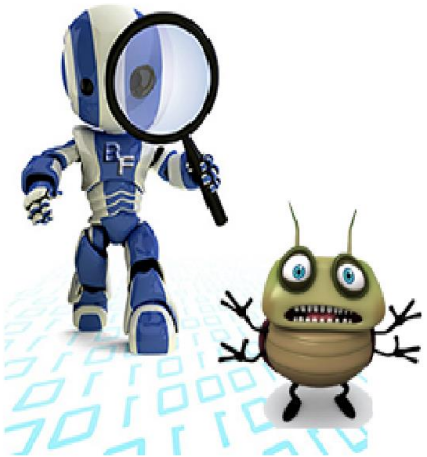
- Not every defect results in an infection, and not every infection/error results in a failure.

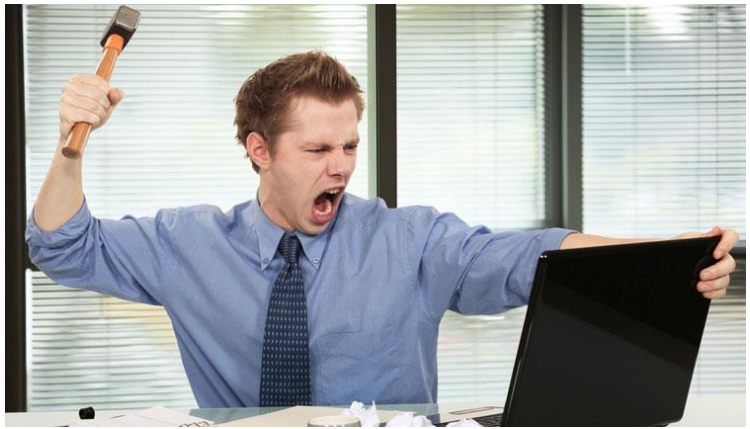
How To Debug Automatically

- A variety of tools and techniques are available to *automate debugging:*
  - Program Slicing
  - Observing & Watching State
    - logging
    - using debugger tools
  - Asserting Invariants
  - Detecting Anomalies
  - Isolating Cause-Effect Chains

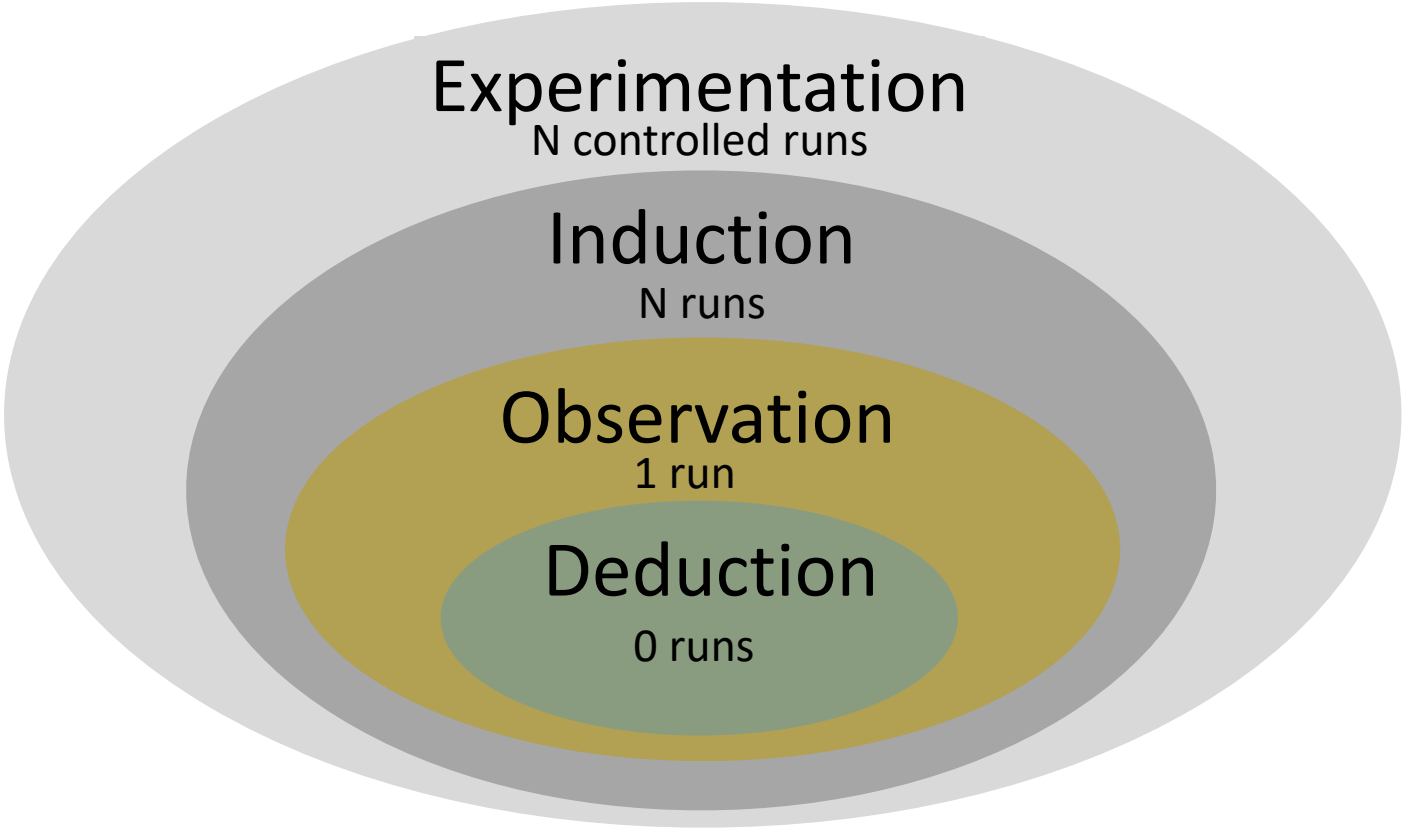
Debugging Initiation

- Typically, debugging process is initiated when:
  - An automated test case causes a failure
  - A user experiences a failure and submits a *“Bug Report”*





Debugging is Reasoning



How to Debug Automatically

- A variety of tools and techniques are available to *automate debugging:*
  - Program Slicing
  - **Observing & Watching State**
    - **logging**
    - **using debugger tools**
  - Asserting Invariants
  - Detecting Anomalies
  - Isolating Cause-Effect Chains

Debugging by Observation

- Determine facts based on what has happened in a concrete run
- Know what to observe and when to observe in a systematic way
- Debugging by Observation techniques:
  - Logging
  - Interactive debugging
  - Postmortem debugging

Observation Principles

- **Proceed systematically:** Rather than observing values at random, search scientifically → develop hypotheses
- **Know what to observe and when to observe:** program run is a long succession of huge program states (i.e., large number of variables), so it is impossible/impractical to observer everything all the time
- **Do not interfere:** Whatever you observe should be the effect of the original program run rather than an effect of your observation

Debugging by Observation

- How can we observe the software state:

Logging the execution

Logging the Execution

- General idea: Insert output statements at specific places in the program
- Also known as *println* debugging

```
public void quickSort(int arr[],
                      int low, int high) {
    if (low < high) {
        /* pi is partitioning index,
           arr[pi] is now at right place */
        int pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
```

```
public void quickSort(int arr[],
                      int low, int high) {
    if (low < high) {
        /* pi is partitioning index,
           arr[pi] is now at right place */
        int pi = partition(arr, low, high);
        System.out.println("pi is: " + pi);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
```

“println” Debugging Issues

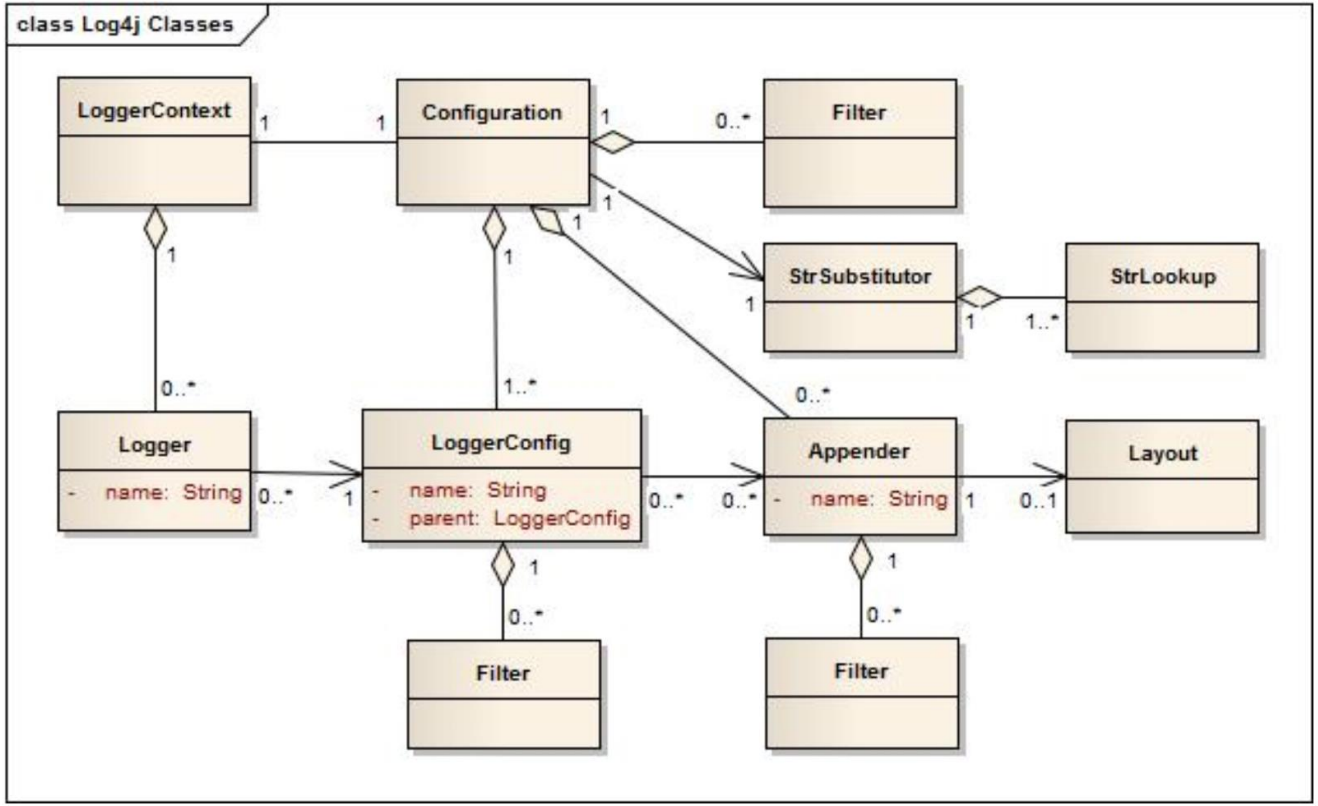
- Cluttered code: logging statements serve no purpose other than debugging
- Cluttered output: logging statements can produce a large amount of output which gets interleaved with ordinary output
  - designate a separate channel for logging (e.g., error channel, a separate logfile etc.)
- Slowdown: huge amount of logging statements can slow down the program
- Loss of Data: for performance reasons, outputs are buffered before being outputted
  - if the program crashes, output data will be lost
  - do not buffer or buffer less frequently → **Slowdown**

Apache Log4j 2

- A full-fledged logging framework
- Offers more functionality compared to `java.util.logging`
- Many open-source applications utilize log4j

https://logging.apache.org/log4j/2.x/

Log4j Architecture



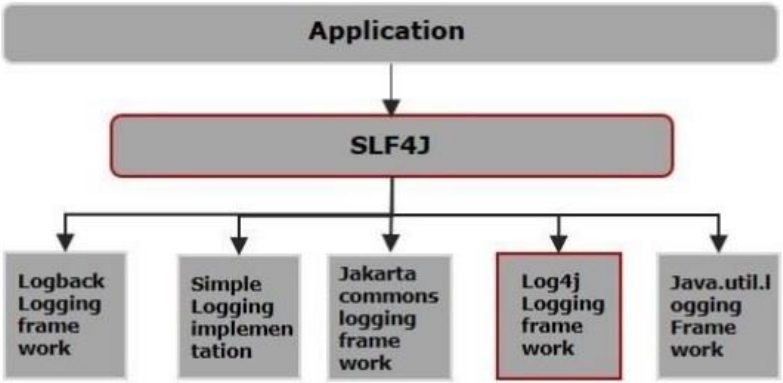
Log Levels

All < Trace < Debug < Info < Warn < Error < Fatal < Off

Event Level	LoggerConfig Level						
	TRACE	DEBUG	INFO	WARN	ERROR	FATAL	OFF
ALL	YES	YES	YES	YES	YES	YES	NO
TRACE	YES	NO	NO	NO	NO	NO	NO
DEBUG	YES	YES	NO	NO	NO	NO	NO
INFO	YES	YES	YES	NO	NO	NO	NO
WARN	YES	YES	YES	YES	NO	NO	NO
ERROR	YES	YES	YES	YES	YES	NO	NO
FATAL	YES	YES	YES	YES	YES	YES	NO
OFF	NO	NO	NO	NO	NO	NO	NO

SLF4J

- Simple Logging Facade for Java (abbreviated SLF4J):
  - acts as a facade for different logging frameworks e.g., java.util.logging, logback, Log4j 2).
  - The underlying logging framework can be plugged in at run-time



Simplifying

- Once one has reproduced a problem, one must find out what's relevant:
  - Does the problem really depend on 10,000 lines of input?
  - Does the failure really require this exact schedule?
  - Do we need this sequence of calls?

Why Simplify

- An airplane crashes:
  - Remove passenger seats, does it still crash?
  - Remove coffee machine, does it still crash?
  - Remove the engines, it does not move

engines are relevant!

Simplifying and Circumstances

- For every circumstance of the problem, check whether it is relevant for the problem to occur.
- If it is not, remove it from the problem report or the test case in question.
- Any aspect that may influence a problem is a circumstance:
  - Aspects of the problem environment
  - Individual steps of the problem history

Simplifying by Experimentation

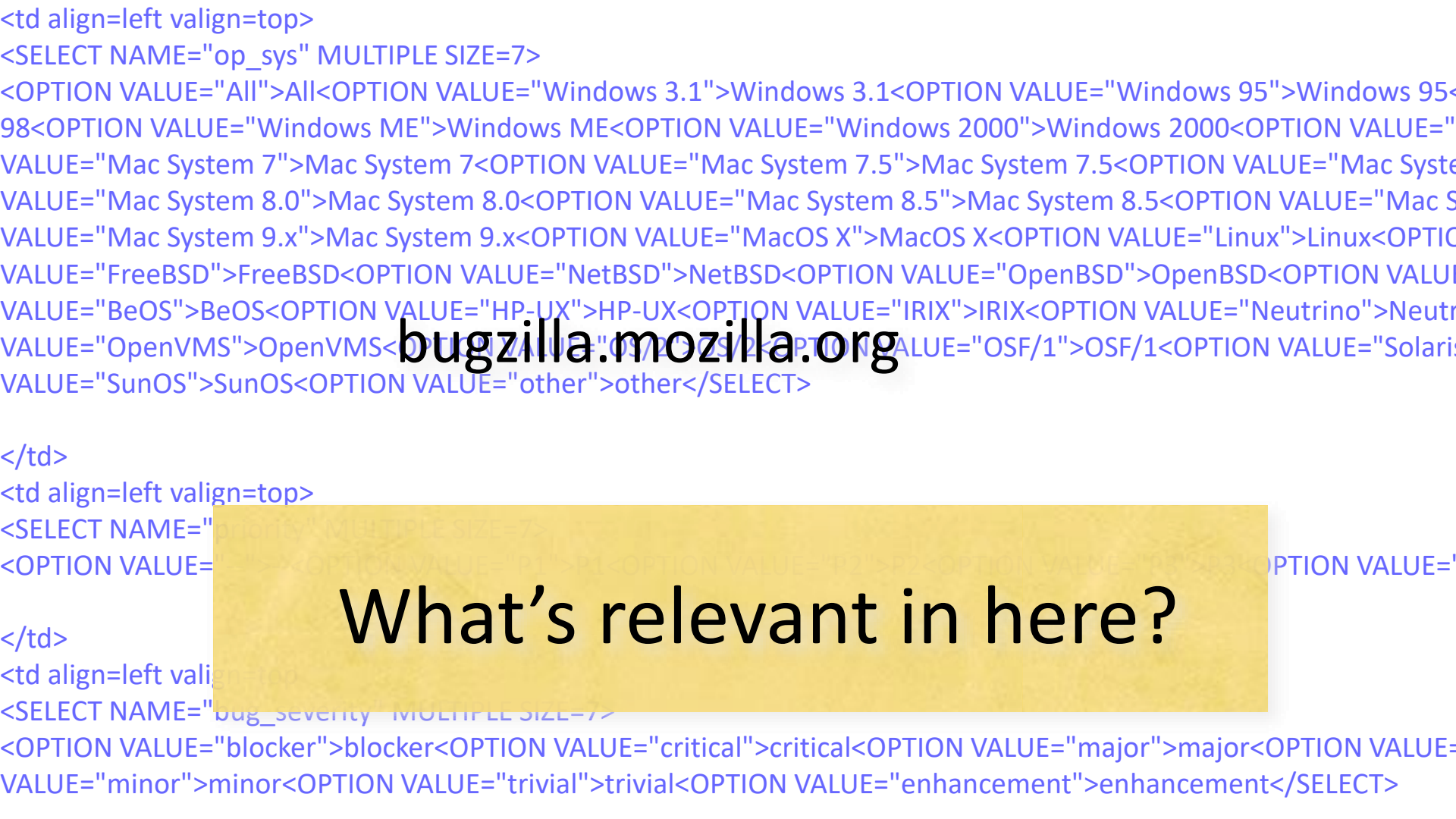
- By experimentation, one finds out whether a circumstance is relevant or not:
- Omit the circumstance and try to reproduce the problem.
- The circumstance is relevant iff the problem no longer occurs.

Mozilla Gecko and a Reported Bug

- Gecko: Mozilla HTML layout engine
- In 1999, there were 370 open problem reports
- Loading an 896-lines HTML crashed the browser
- Much better to work with the smallest possible HTML input file that contains the “failure cause”

Why Simplify

- Ease of communication:
  - A simplified test case is easier to communicate.
- Easier debugging:
  - Smaller test cases result in smaller states and shorter executions.
- Identify duplicates:
  - Simplified test cases subsume several duplicates.


bugzilla.mozilla.org
What’s relevant in here?

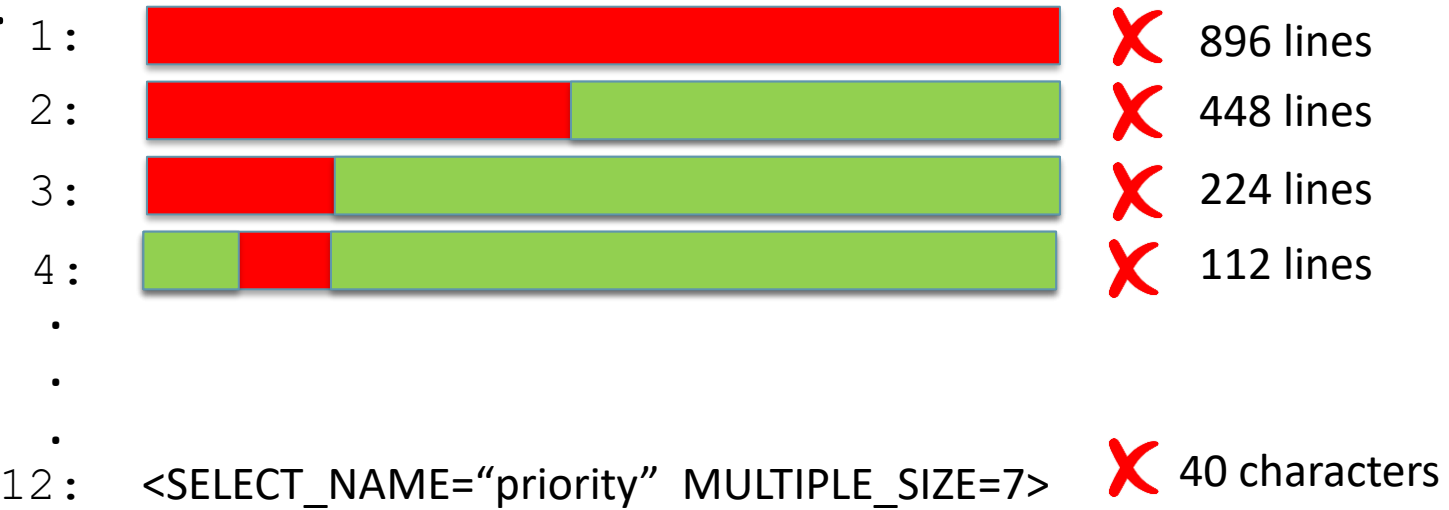
The Gecko BugAThon

- New problem reports came in way faster than the Mozilla developers could possibly simplify them or even look at them

- Eric Krock, the Mozilla product manager, came up with a brilliant idea
  - Download the Web page to your machine.
  - Using a text editor, start removing HTML from the page. Every few minutes, make sure it still reproduces the bug.
  - Code not required to reproduce the bug can be safely removed.
  - When you've cut away as much as you can, you're done.

Binary Search

- 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.



Simplified Input

<SELECT NAME="priority" MULTIPLE SIZE=7>

- Simplified from 896 lines to one single line
- Required 12 tests only

Benefits

- Ease of communication:
  - All one needs is “<SELECT> tag causes a crash”
- Easier debugging:
  - We can directly focus on the piece of code that renders <SELECT>
- Identify duplicates:
  - Check other test cases whether they’re <SELECT>-related, too.

Automated Simplification

- Manual simplification is slow & boring.
- We have machines for mechanical tasks.
- Basic idea:
  - We set up an automated test that checks whether the failure occurs or not e.g., Mozilla crashes or not
  - We implement a strategy that realizes the binary search

Automated Test

- Launch Mozilla
- Replay (previously recorded) steps from problem report
- Wait to see whether
  - Mozilla crashes (= the test fails)
  - Mozilla still runs (= the test passes)
- If neither happens, the test is *unresolved*

Binary Search

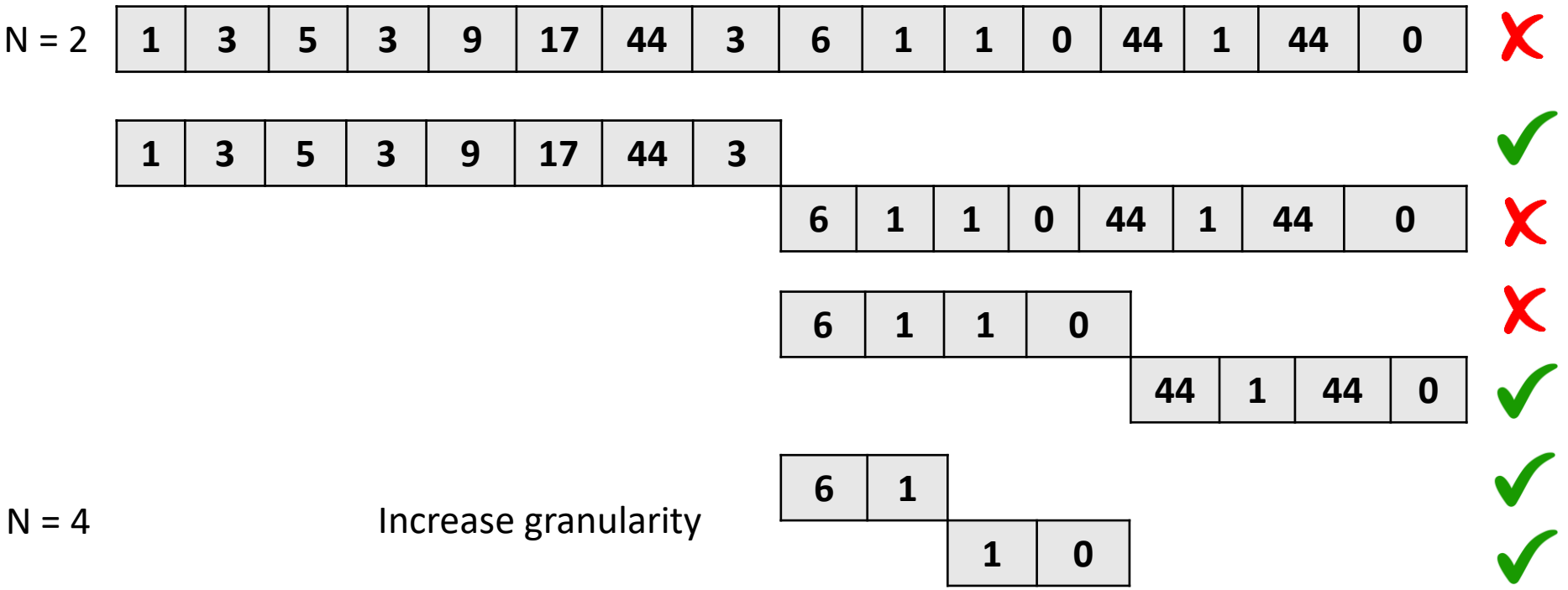
- What do we do if both halves pass?
  - Increase granularity, i.e., break the input into smaller pieces

Example

```
public static int checkSum(int[] a)
```

- is supposed to compute the checksum of an integer array
- gives wrong result, whenever “a” contains two identical consecutive numbers, **but we don’t know that yet**
- we have a failed test case, e.g., from protocol transmission:
  - {1, 3, 5, 3, 9, 17, 44, 3, 6, 1, 1, 0, 44, 1, 44, 0}

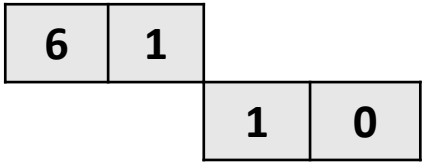
Another Example (N is number of chunks)

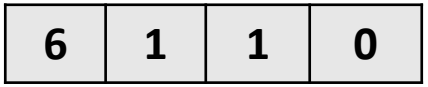


Another Example - Continued

N = 4

Increase granularity





N = 3

Adjust granularity to input size

ddmin Algorithm

- Let **`c`** be a failing input configuration (sequence of individual inputs)
- **`test(c)`** runs a test on **`c`** with possible outcome PASS or FAIL
- **`n`** is the number of chunks to split **`c`** into (initially **`n = 2`**). We will remove one chunk at a time and test the remaining input.

```
ddMin(c, n) :
1. If |c| = 1 return c
```

ddmin Algorithm

- Let **c** be a failing input configuration (sequence of individual inputs)
- **test(c)** runs a test on **c** with possible outcome PASS or FAIL
- **n** is the number of chunks to split **c** into (initially **n = 2**). We will remove one chunk at the time and test the remaining input.

ddMin(c, n) :

1. If |c| = 1 **return c**

Otherwise, systematically remove one chunk c_i at the time. Test the remaining input $c \setminus c_i$:

2. If there exist some c_i such that test(**c** \ c_i) = FAIL return **ddMin(c \ c_i , max(n-1, 2))**

ddmin Algorithm

- Let **c** be a failing input configuration (sequence of individual inputs)
- **test(c)** runs a test on **c** with possible outcome PASS or FAIL
- **n** is the number of chunks to split **c** into (initially **n = 2**). We will remove one chunk at the time and test the remaining input.

ddMin(c, n) :

1. If |c| = 1 **return c**

Otherwise, systematically remove one chunk c_i at the time. Test the remaining input $c \setminus c_i$:

2. If there exist some c_i such that test(**c** \ c_i) = FAIL return **ddMin(c \ c_i , max(n-1, 2))**

3. Else, if n < |c| return **ddMin(c, min(2n, |c|))**

ddmin Algorithm

- Let **c** be a failing input configuration (sequence of individual inputs)
- **test(c)** runs a test on **c** with possible outcome PASS or FAIL
- **n** is the number of chunks to split **c** into (initially **n = 2**). We will remove one chunk at the time and test the remaining input.

ddMin(c, n) :

1. If: |c| = 1 **return c**

// Otherwise, systematically remove one chunk c_i at the time. Test the remaining input **c** \ c_i :

2. If there exist some c_i such that test(**c** \ c_i) = FAIL
return **ddMin(c \ c_i , max(n-1, 2))**

3. Else if: n < |c|
return **ddMin(c, min(2n, |c|))**

4. Else: // (can't split into smaller chunks)
return **c**

Delta Debugging

- The technique is an instance of *delta debugging*:
  - An approach to isolate failure causes by narrowing down differences (deltas) between runs
- Delta Debugging can be applied to various types of inputs such as:
  - failure-inducing program input, e.g., HTML page
  - failure-inducing user interactions e.g., the key/mouse strokes that make a program crash
  - failure-inducing changes to the program code, e.g., after a failing regression test
  - etc.

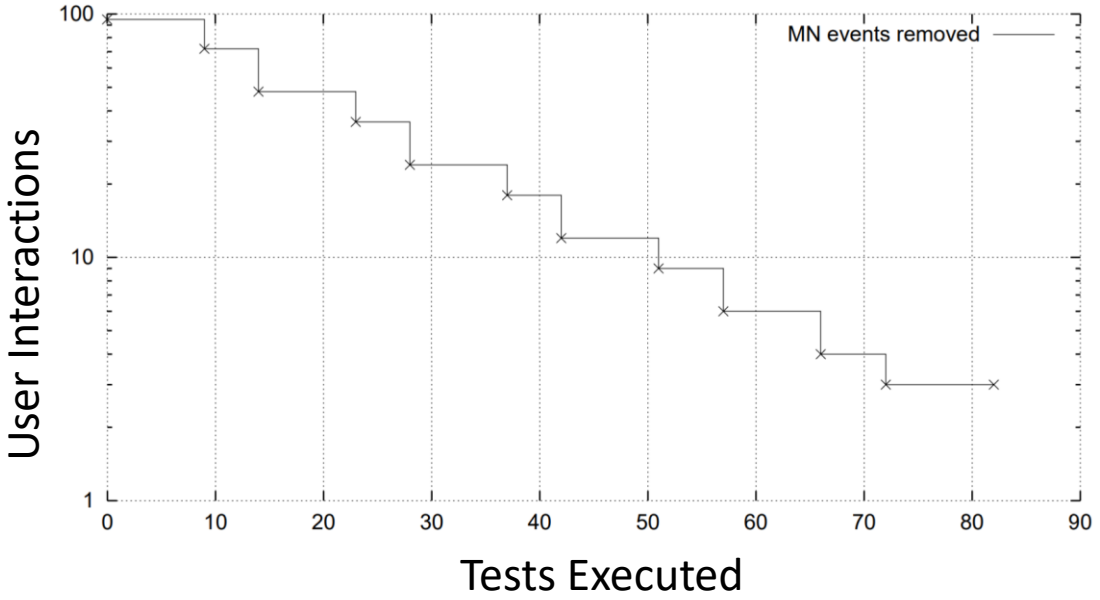
Delta Debugging

Delta Debugging gives an alternate mindset to the debugging problem. Normally, when something does not work as expected (i.e., you have a “failure”), you would naturally think: “hmmm, what’s wrong here?”. Delta Debugging takes an alternate approach: “what could NOT be wrong here?” In other words, “what is irrelevant here?” so that I can exclude those parts and put them aside to simplify things.

Delta Debugging

- After 82 tests, **`ddmin`** has simplified the user interactions to 3 events:

  1. Press `P` while holding `Alt`
  2. Press the `left mouse button` on the `Print` button
  3. Release the `left mouse button`



Relevant Reads and Resources

- Recommended Texts:
  - “Why Programs Fail”: ch1 and ch2
- https://logging.apache.org/log4j/
- https://www.slf4j.org/
- Recommended Texts
  - “Why Programs Fail”: ch5
- https://www-archive.mozilla.org/newlayout/bugathon.html
- TDA567/DIT082 Chalmers University of Technology http://www.cse.chalmers.se/edu/year/2018/course/TDA567/