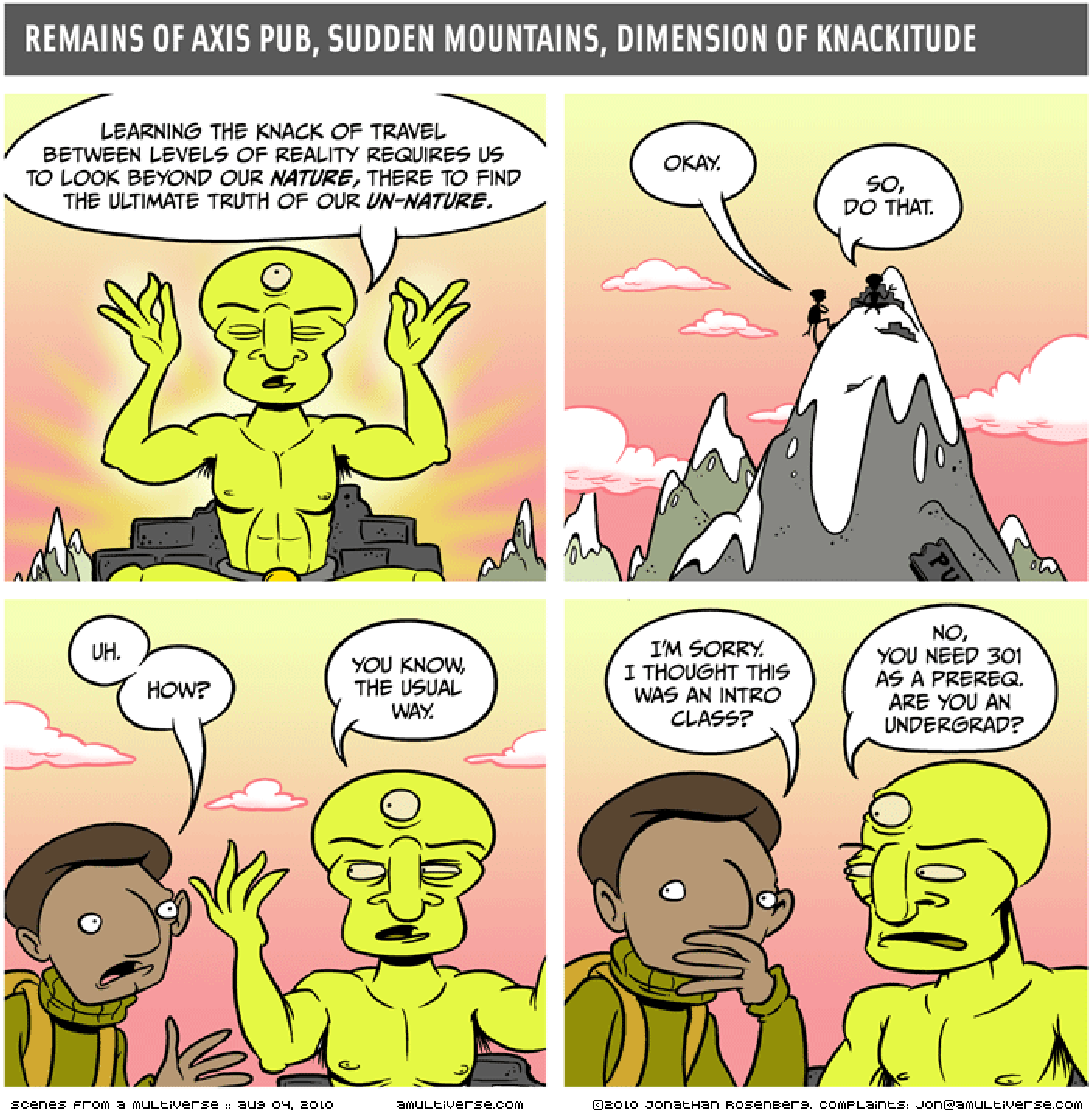
# Test Inputs, Oracles and Generation



# One-Slide Summary

- Formally, a **test case** consists of an **input** (**data**), an **oracle** (output), and a **comparator**.
- Test inputs determine the behavior of the program. High-coverage inputs can be **generated automatically** through path enumeration, path predicates and mathematical constraint solving.
- Test oracles correspond to what the program should do. Generating them is an expensive **problem**; it can be done automatically through invariants and mutation.
- **Test suite minimization** finds the smallest subset of tests that meet a coverage goal.

# The Story So Far …

- Testing is very expensive (e.g., 35% of total IT spending).
- Test suite **quality metrics** support informed comparisons between tests.
- But where do we get one test, much less many to compare?

# Outline

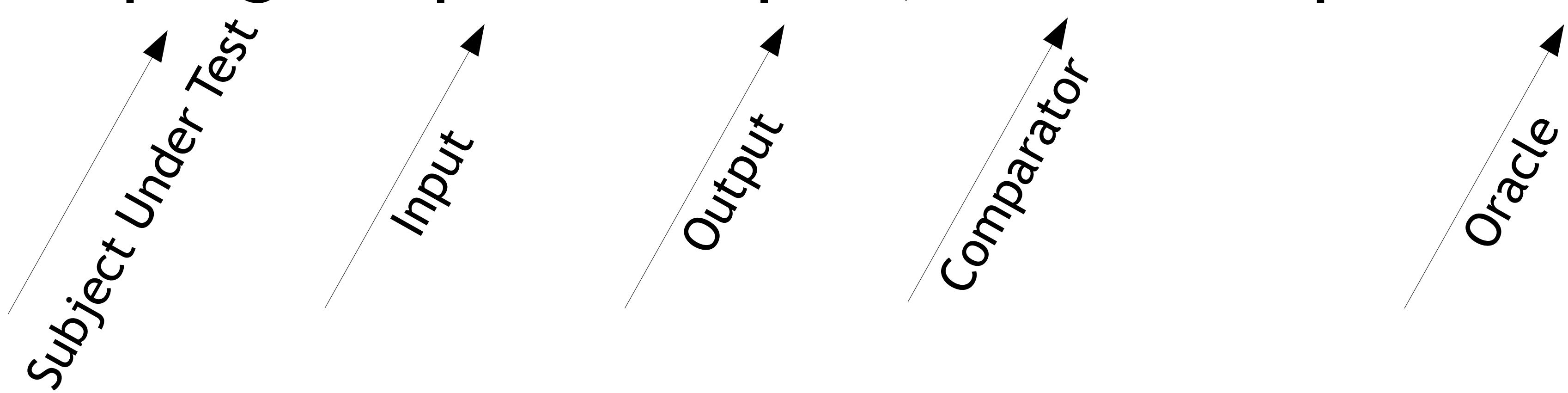
- Test inputs
- Test input generation
- Test oracles
- Test oracle generation
- Test minimization

- “Kill it with Math” vs. “Humans Are Central”

# What is a test?

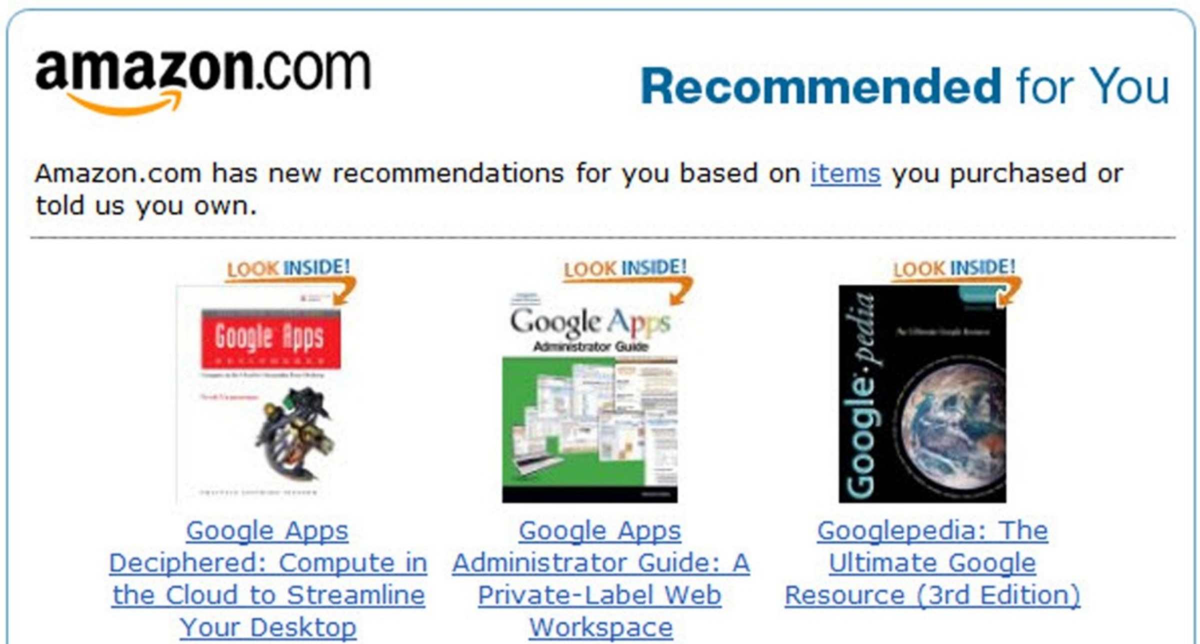
- Formally, a **test case** has three components: the **test input** (or **data**), the **test oracle** (or expected output), and the **comparator**.
  - Sometimes called the Oracle-Comparator model.

```
./prog < input > output ; diff -b output oracle
```



# Comparator

- Many test cases use "must match exactly" as the comparator
- But officially it could be more general
  - Known random output, precision limits, embedded dates, etc.



# Non-Trivial Comparator Example

- jsoup/internal/ConstrainableInputStreamTest.java

  (from Homework 2)

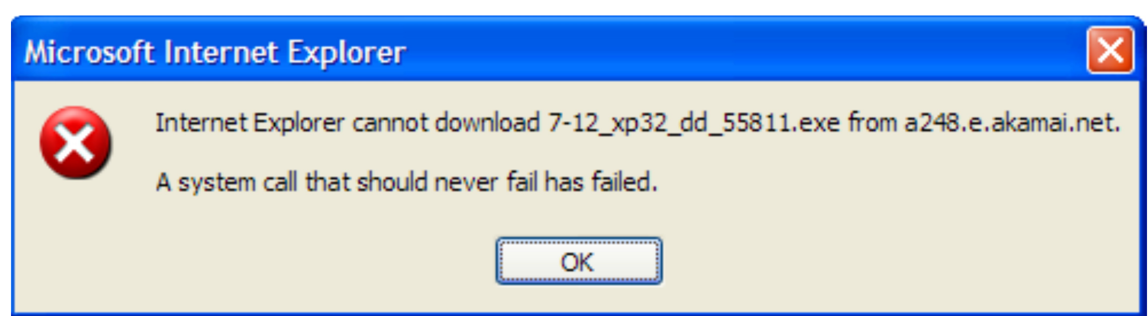
```
@Test
public void noLimitAfterFirstRead() throws IOException {
    int bufferSize = 5 * 1024;

    String url = "http://direct.infohound.net/tools/large.html"; // 280 K
    BufferedInputStream inputStream = Jsoup.connect(url).execute().bodyStream();

    assertTrue(inputStream instanceof ConstrainableInputStream);
    ConstrainableInputStream stream = (ConstrainableInputStream) inputStream;

    // simulates parse which does a limited read first
    stream.mark(bufferSize);
    ByteBuffer firstBytes = stream.readToByteBuffer(bufferSize);
    byte[] array = firstBytes.array();
    String firstText = new String(array, "UTF-8");
    assertTrue(firstText.startsWith("<html><head><title>Large"));
    assertEquals(bufferSize, array.length);
```

# Test Data

- What are *all* the inputs to a test?
  - Many programs (especially student programs) read from a file or stdin …
  - But what else is “read in” by a program and may influence its behavior?

# Test Inputs

- User Input (e.g., GUI)
- Environment Variables, Command-Line Args
- Scheduler Interleavings
- Data from the Filesystem
  - User configuration, data files
- Data from the Network
  - Server and service responses



# Operating System Philosophy

- "Everything is a file."
- After a few libraries and levels of indirection, reading from the user's keyboard boils down to opening a special device file (e.g., /dev/ttyS0) and reading from it
  - Similarly with mouse clicks, GUI commands, etc.
- Ultimately programs can only interact with the outside world through **system calls**
  - open, read, write, socket, fork, gettimeofday
  - Those (plus OS scheduling, etc.) are the full inputs

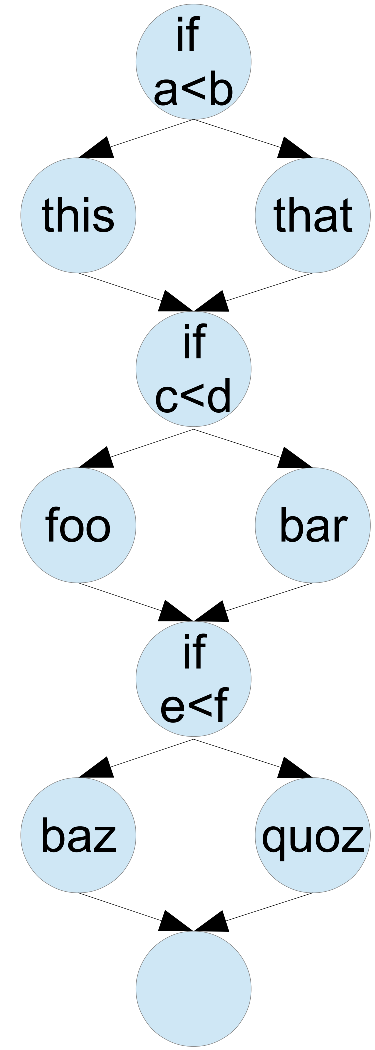
# Test Input Generation

- We want to generate high quality tests
  - **Automatically**!
- Using test suite metrics to prefer some tests

- Statement Coverage: visit every line
- Branch Coverage: visit every →true, →false
- Path Coverage: visit every path

# Path Coverage

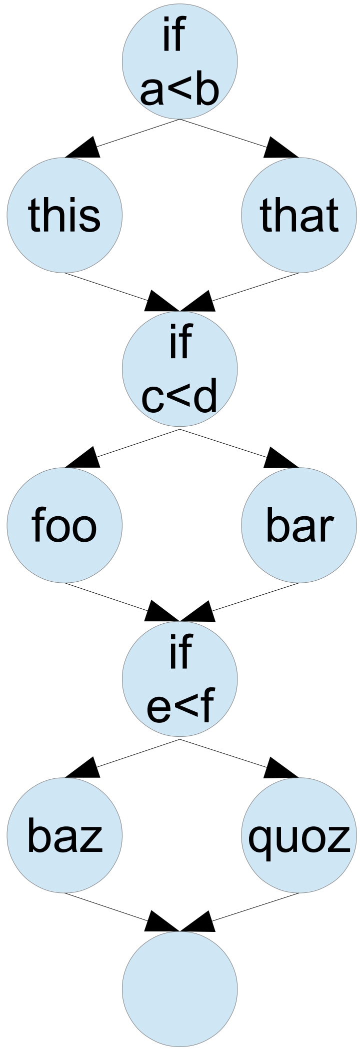
```
foo(a,b,c,d,e,f):
  if a < b: this
  else: that
  if c < d: foo
  else: bar
  if e < f: baz
  else: quoz
```




- How many *paths*?

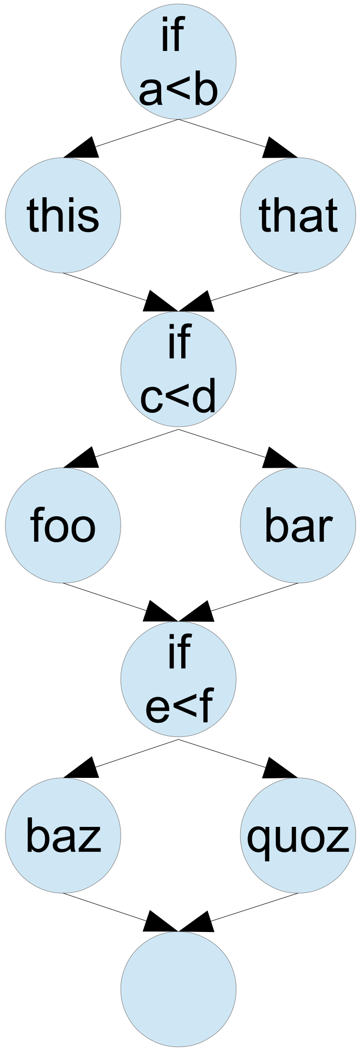
# Path Coverage

```
foo(a,b,c,d,e,f):
  if a < b: this
  else: that
  if c < d: foo
  else: bar
  if e < f: baz
  else: quoz
```




- There are 8 paths, but only 6 branch coverage edges

# Branch vs. Path



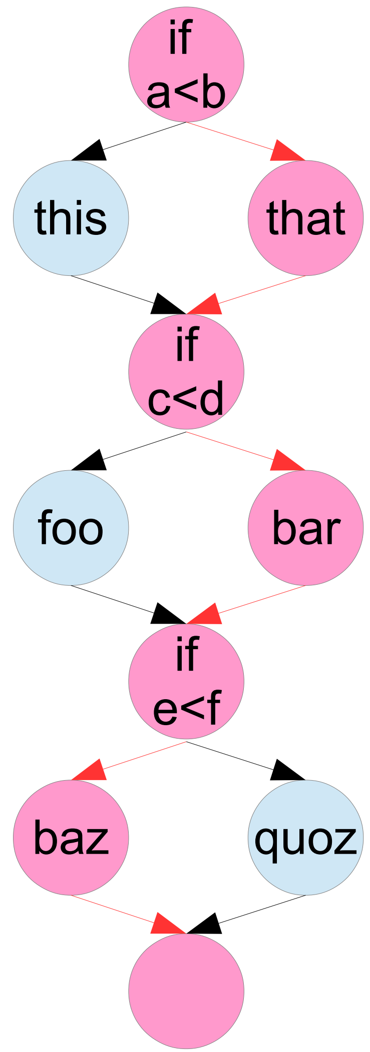

- If you have **N sequential (or serial) if-statements** …
- There are **2N branch edges**
  - Which you could cover in 2 tests!
    - One always goes left, one always right
- But there are **$2^N$ paths**
  - You need $2^N$ tests to cover them
- Path coverage subsumes branch coverage

# Path Test Input Generation

- Consider generating test inputs to cover a path
  - If we could do that, branch, stmt, etc., would be easy!
- Solve this problem with math
- A **path predicate** (or **path condition**, or **path constraint**) is a boolean formula over program variables that is true when the program executes the given path

# Path Predicate Example

- Consider the highlighted path
  - a.k.a. “False, False, True”
- Its path predicate is
  - `a >= b && c >= d && e < f`
- When the path predicate is true, control flow follows the given path
- So what should we do to make a test input that covers this path?



# Solving Systems of Equations

- A **satisfying assignment** is a mapping from variables to values that makes a predicate true.
- One satisfying assignment for

```
a >= b && c >= d && e < f
```

- Is

```
a=5, b=4, c=3, d=2, e=1, f=2
```

- Another Is

```
a=0, b=0, c=0, d=0, e=0, f=1
```

# Producing Satisfying Assignments

- Ask Humans (HW1?)
  - Labor-intensive, expensive, etc.
- Repeatedly guess randomly
  - Works surprisingly well (when answers are not sparse)
- Use an **automated theorem prover**
  - cf. Wolfram, MatLab, Mathematica, etc.
  - Works very well on restricted types of equations (e.g., linear but not arbitrary polynomial, etc.)

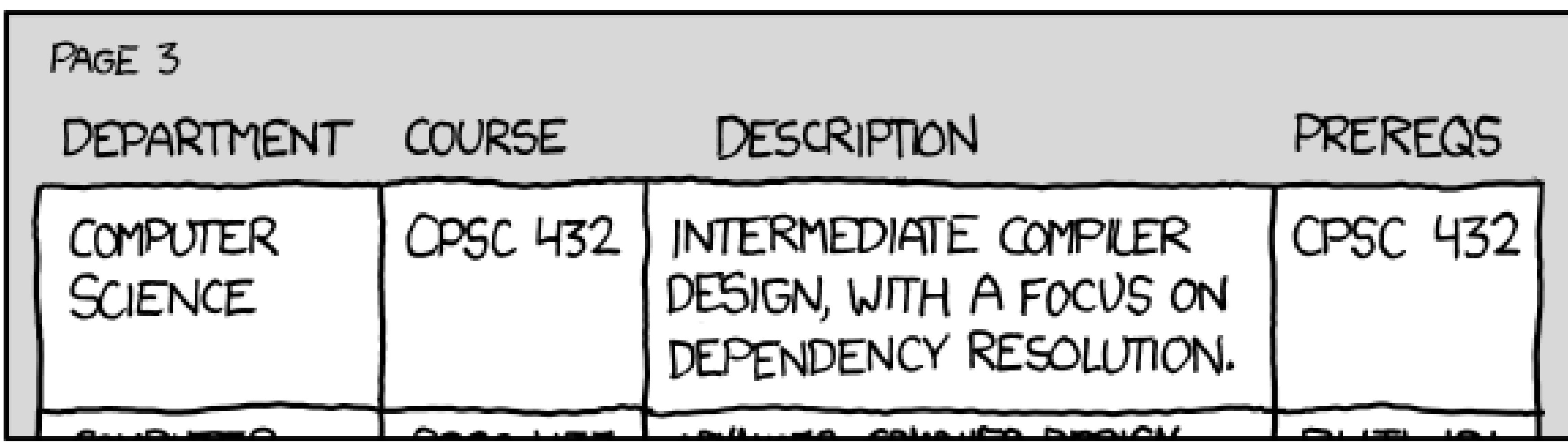
# Test Input Generation Plan

- Consider generating high-branch-coverage tests for a method …

- Enumerate "all" paths in the method
- For each path, collect the path predicate
- For each path predicate, solve it
  - A solution is a satisfying assignment of values to input variables → those are your test input
  - None found? Dead code, tough predicate, etc.

# Enumerating Paths

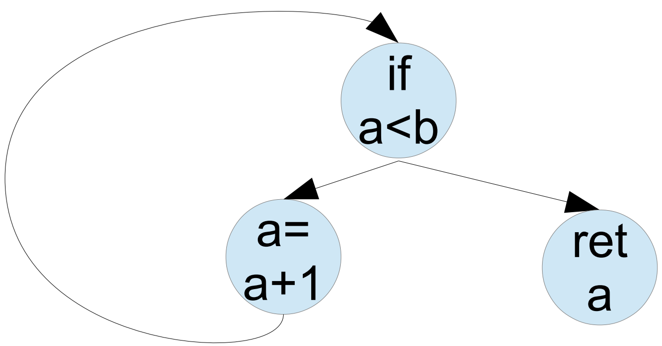
- What could go wrong with enumerating paths in a method?



# Enumerating Paths

- What could go wrong with enumerating paths in a method?
- There could be infinitely many!

```
while a<b:
  a = a + 1
return a
```




- One path corresponds to executing the loop once, another to twice, another to three times, etc.

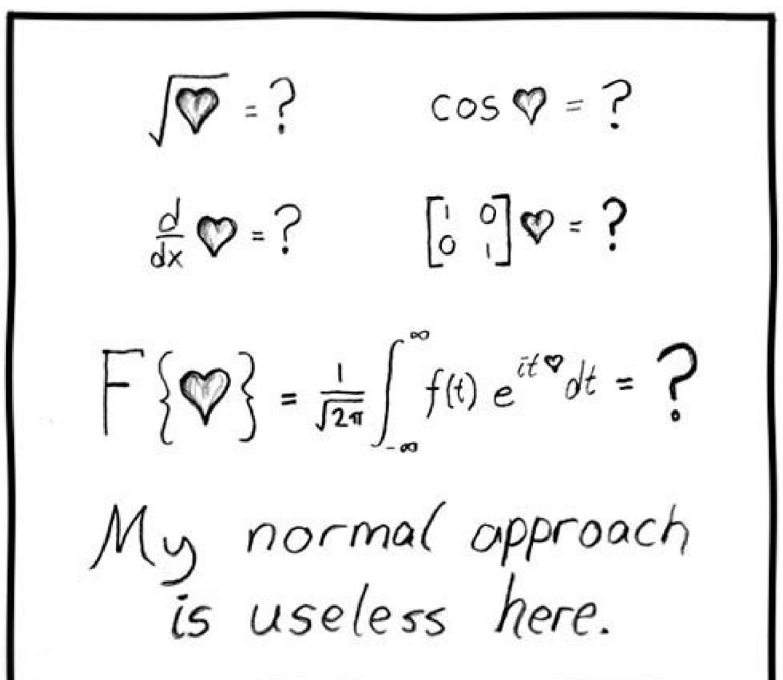
# Path Enumeration Approximations

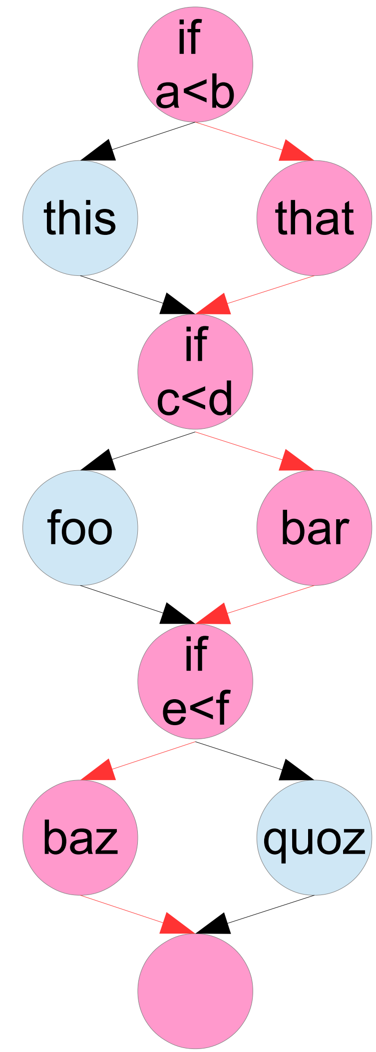
- Typical Approximations
  - Consider only **acylic** paths (corresponds to taking each loop zero times or one time)
  - Consider only taking each loop at most *k* times
  - Enumerate paths breadth-first or depth-first and stop after *k* paths have been enumerated

- (For more information, take a *Programming Languages*, *Compilers* or *Theory* class)

# Collecting Path Predicates

- Now we have a path through the program
- What could go wrong with collecting the path predicate?



# Path Predicate

- The path predicate may not be expressible in terms of the inputs we control

```
foo(a,b):
  str1 = read_from_url("abc.com")
  str2 = read_from_url("xyz.com")
  if (str1 == str2):
    bar()
```

- Suppose we want to exercise the path that calls `bar`. One predicate is `str1==str2`. What do you assign to `a` and `b`?

# Path Predicate Woes

- Typical solutions:
  - “We don't care.”
  - Collect up the path predicate as best you can
  - Ask the solver to solve it in terms of the input variables
  - If it can't

    … either because the math is too hard

    … or because the variables are out of our control
  - Then we don't generate a test input exercising that path. Best effort.

# Trivia: Worldwide Box Office

- Identify the top-six grossing worldwide cinematic franchise associated with:
  - The most versatile substance on the planet, and they used it to make a Frisbee. ($13.5B)
  - Do. Or do not. There is no try. ($8.9B)
  - You'll be next Mudbloods! ($8.5B)
  - A martini. Shaken, not stirred. ($7.1B)
  - Even the smallest person can change the course of the world. ($5.9B)
  - I live my life a quarter mile at a time. ($5.1B)

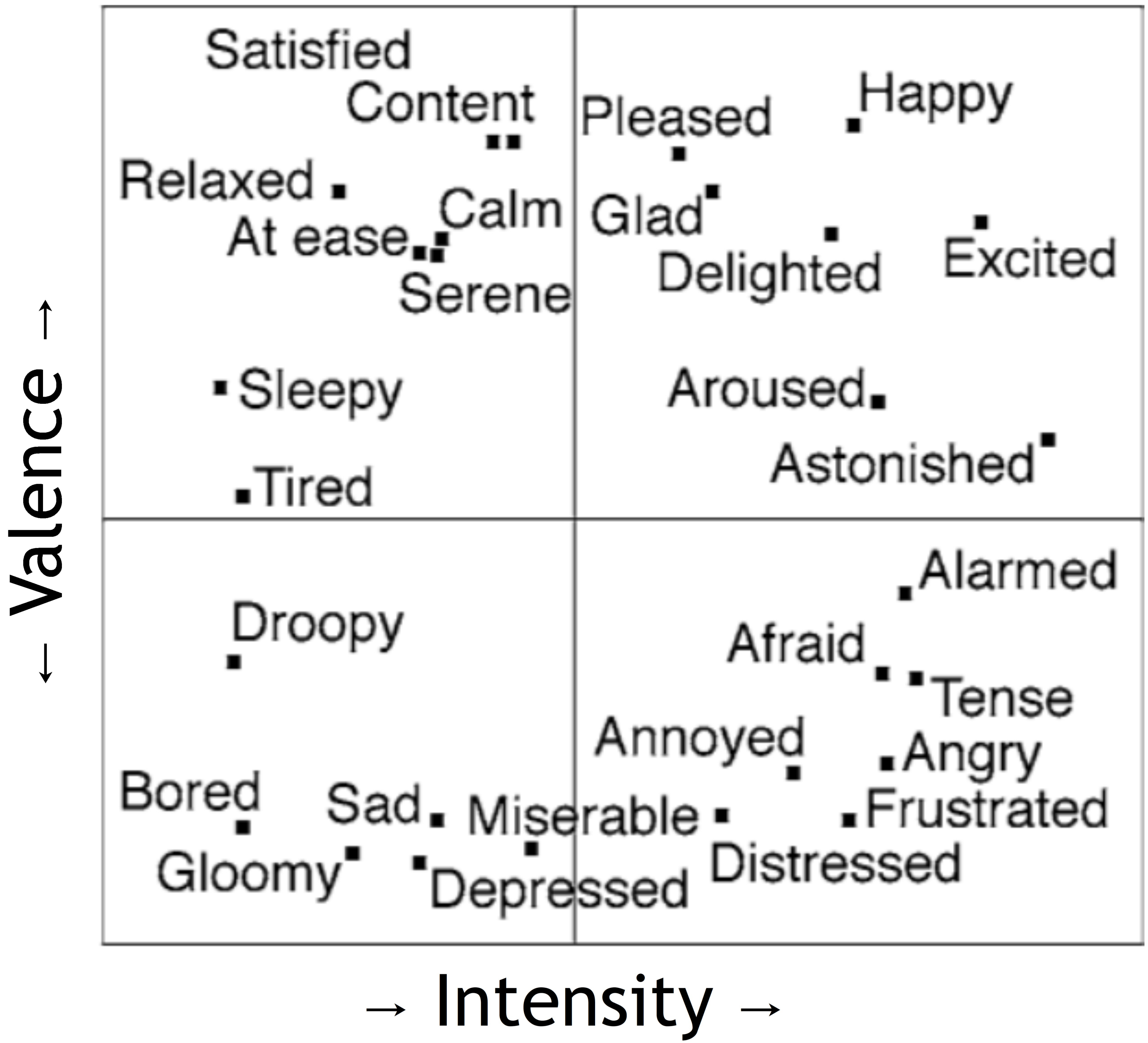
# Psychology: Memory

- Which factors make it more likely that you will remember something that happened to you:
  - The memory was happy
  - The memory was calm
  - The memory was sad
  - The memory was from long ago
  - The memory was recent

# Psychology: Memory

- In three experiments involving hundreds of participants, researchers found that "**intensity** affects the properties of autobiographical memories more so than does valence"
  - Valence = positive or negative emotion
  - Intensity = strong or weak emotion

# Psychology: Memory



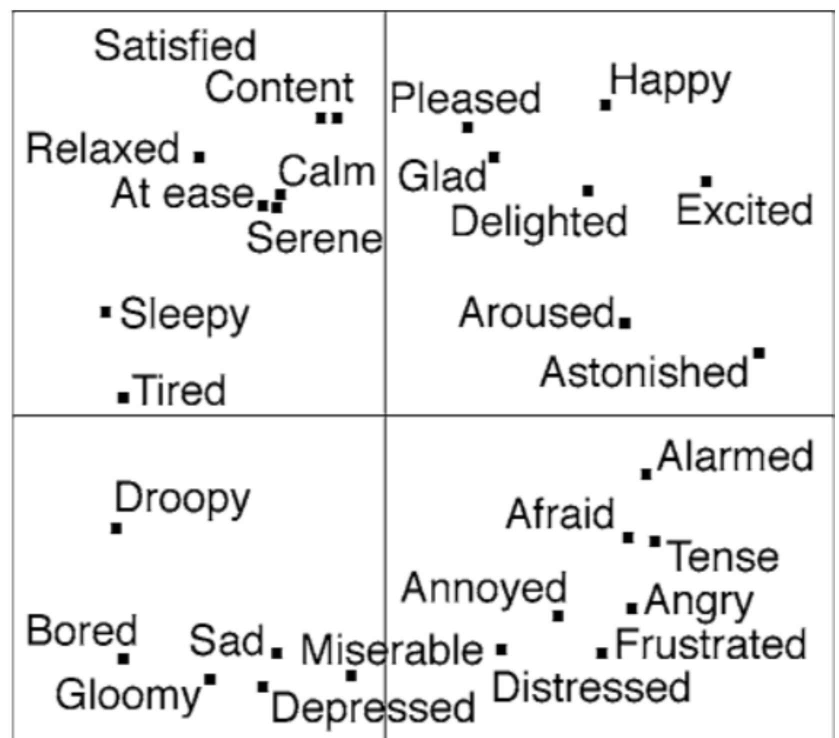
# Emotional Intensity Predicts Autobiographical Memory Experience

- "**intensity** affects the properties of autobiographical memories more so than does valence … these intensity differences are not the result of a simple retention difference, because the age of the memory was also included in the analyses and it was less influential than intensity or valence … not only will highly intense events tend to be remembered longer, but they will also tend to be remembered with greater vividness, a greater sense of recollection"

[ JENNIFER M. TALARICO, KEVIN S. LABAR, and DAVID C. RUBIN. Memory & Cognition, 2004, 32 (7) 1118-1132. ]

# Emotional Intensity Predicts Autobiographical Memory Experience

- Implications for SE: When asked to evaluate code are less likely to remember the times we were "merely" satisfied (or bored). Instead we will remember the times we were excited or alarmed by bugs.



# Test Data Generation

- One of the earliest approaches was DART (Directed Automatic Random Testing)
- Their example program has three paths:
  - False, True-False, True-True

```
int f(int x, int y) {
  int z;
  z = y;
  if (x == z)
    if (y == x + 10)
      abort();
  return 0;
}
```

- Predicates:
  - **`z=y && x!=z`**
  - **`z=y && x=z && y!=x+10`**
  - **`z=y && x=z && y=x+10`**
- *Participation*: For each of those 3 predicates, give 1 solution in terms of **`x`** and **`y`**.

# Microsoft's Pex Tool

- Pex is a test input generation tool integrated into Visual Studio
  - It has special handling for pointers, is language-independent, etc., but otherwise works just like what we covered here
  - Other tools (e.g., jCUTE for Java) exist

# Does it Work?

| Class | Blocks | Block Coverage | Arcs | Arc Coverage |
|---|---|---|---|---|
| A (mostly stateless methods) | >300 | 95% | >400 | 90% |
| B (mostly stateless methods) | >100 | 97% | >200 | 94% |
| C (stateful) | >200 | 76% | >300 | 65% |
| D (parsing code) | >500 | 81% | >800 | 73% |
| E (numerical algorithms) | >400 | 71% | >600 | 67% |
| F (numerical algorithms) | >100 | 82% | >200 | 79% |
| G (numerical algorithms) | >100 | 98% | >100 | 97% |
| H (numerical algorithms) | >200 | 71% | >200 | 61% |
| I (numerical algorithms) | >200 | 97% | >300 | 96% |

- Why are these MS Dot.Net classes anonymous?
- What are block and arc coverage?

# So, did we win?

- We want to automatically generate test cases
- We have an approach that works well in practice:
  - Enumerate some paths
  - Extract their path constraints
  - Solve those path constraints
- What are we missing?

# We Forgot Oracles!

- We know to generate test inputs
  - e.g., "for high coverage, run f(1,0) and f(-5,-7)"
- But we don't know what the answer is supposed to be when you do that!
- So we cannot tell if a program is passing or failing.

- Well … maybe we can still salvage something. Thoughts?

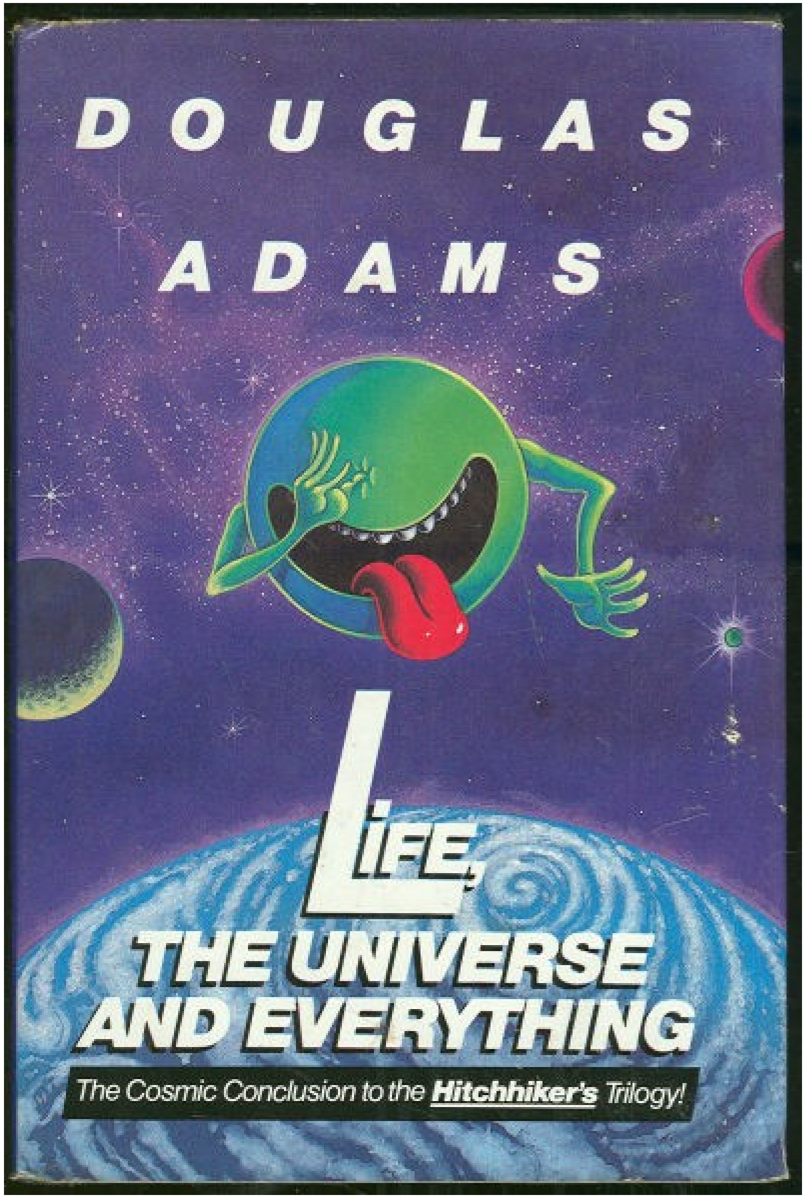
# Test Generation → Bug Finding

- If your program crashes on that input → bad
- "This paper presents EXE, an effective bug-finding tool that automatically generates inputs that crash real code … EXE works well on real code, finding bugs along with inputs that trigger them in: the BSD and Linux packet filter implementations, the udhcpd DHCP server, the pcre regular expression library, and three Linux file systems."

[Cadar et al. EXE: Automatically Generating Inputs of Death. CCS 2006.]

# Big Problem

- In general, though, we're going to need **both** the question **and** the answer!




- But don't panic yet ...
- No need to throw in the towel ...

# Oracles

- "If Croesus goes to war he will destroy a great empire."
  - ~~Barbara Gordon~~ The Oracle at Delphi, on whether Croesus should go to war against the Persians

- Oracles are tricky.
- Many believe that formally writing down what a program should do is as hard as coding it.
  - (We return to this topic later.)

# The Oracle Problem

- The **Oracle Problem** is the difficulty and cost of determining the correct test oracle (i.e., output) for a given input.
  - “What *should* the program do?”
- It is expensive both for humans and for machines.
- An **implicit oracle** is one associated with the language or architecture, rather than program-specific semantics (e.g., “don't segfault”, “don't loop forever”).

# Aside: Philosophy

- The difficulty here should not be surprising.
- Recall from Ethics that it is often easier to make negative moral edicts (“Do not steal”) than it is to elaborate positive ones (“Here is what it means to be a generous person ...”)
- Similarly, it is much easier to make negative program edicts (“Do not crash”) than it is to elaborate positive ones (“Here is what it means to be a good webserver ...”)

# Idea: Use The Program

- In this setting we do *have* the program
  - We're trying to generate tests for it …
- Perhaps the program itself can somehow tell us what its correct behavior should be
  - But how?

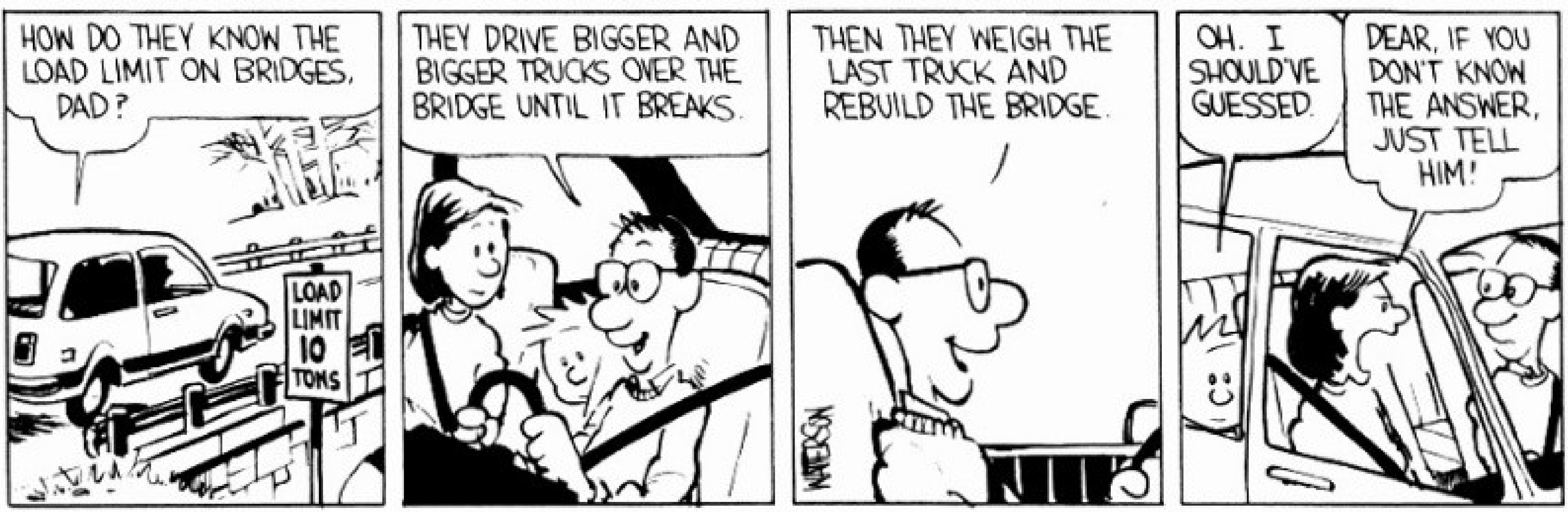
# Insight: Competent Programmers

- We return to the assumption that the program is mostly correct (where was this from?)
- If I run the program ten different times and every time we have index == array_len - 1
  - … perhaps that is the test oracle we want:

    ```
    assertEquals(index, array_len-1);
    ```
  - That is, “it should be true every time”
- An invariant is a predicate over program expressions that is true on every execution.
  - High-quality invariants can serve as test oracles

# Learning Invariants

- We can **learn** (or **infer**) program invariants by running the program many times and noting what is always true of the output
  - e.g., if we run sqrt() many times, we may learn retval>=0



# Learning Invariants

- We can **learn** (or **infer**) program invariants by running the program many times and noting what is always true of the output
  - e.g., if we run sqrt() many times, we may learn retval>=0
- Simple implementation: start with a big list of possible invariants (e.g., retval=0, retval=5, retval>=0, etc.) and, on every run, cross off those that are falsified
  - Recall: by definition an invariant is true on all runs

# Common vs. Correct

- In some sense, we are assuming that common behavior (or behavior we can observe) is correct behavior
- This is like learning the rules of English by reading high school essays. What could go wrong?

# Bad Invariants

- Consider the following situations

- We test sqrt once, on sqrt(9), and learn the invariant: retval==3

- We test findNode thousands of times, and learn the invariant: pointer%4==0

# Fixing This Mess

- The “sqrt == 3” issue can be partially addressed with more random inputs
- The “ptr % 4 == 0” issue is more troubling
  - It is only coincidentally correct here
  - (Why do we care? Hint: cost!)
- Competent Programmers: in general, every line of code matters to correctness

# The Chain of Reasoning

- Competent Programmers: in general, every line of human-written code matters to human-intended correctness
- So if an invariant or oracle captures human-intended correctness, there must be at least one line of code that ensures it
- So if I poke and mutate your programs, I should be able to falsify the invariant!
  - If I can't, that candidate invariant was coincidental and not a product of the code you actually wrote!

# Example

- Suppose we have tested this on 1, 9, 16, 30
- Candidate Invariants:
  - retval < retval+1
  - retval <= 6
  - x >= retval*retval
- What do we do?

```
int floorSqrt (int x) {
  int i = 1, result = 1;

  // Base cases
  if (x == 0 || x == 1)
    return x;

  // Staring from 1, try all numbers until
  // i*i is greater than or equal to x.
  while (result <= x) {
      i++;
      result = i * i;
  }
  return i - 1;
}
```

# Example

Never ruled out by any mutation, dropped!

- Suppose we have tested this on 1, 9, 16, 30
- Candidate Invariants:
  - ~~retval < retval+1~~
  - ~~retval <= 6~~
  - x >= retval*retval
- What do we do?

```
int floorSqrt (int x) {
  int i = 1, result = 1;

  // Base cases
  if (x == 0 || x == 1)
    return x;

  // Staring from 1, try all numbers until
  // i*i is greater than or equal to x.
  while (result <= x) {
      i++;
      result = i * i;
  }
  return i - 1;
}
```

Ruled out by trying more inputs (e.g., 81), dropped!

Falsified by some mutations (which?), retained!

# EvoSuite

- This oracle-generation approach is implemented in the EvoSuite tool
  - It generates high-coverage unit tests for Java
  - It is award-winning, takes first place in competitions as recently as 2017, etc.
  - You will get a chance to try it in Homework 2!
- EvoSuite is an instance of search-based software engineering, a topic we'll return to at the end of this course

# An Embarrassment of Riches

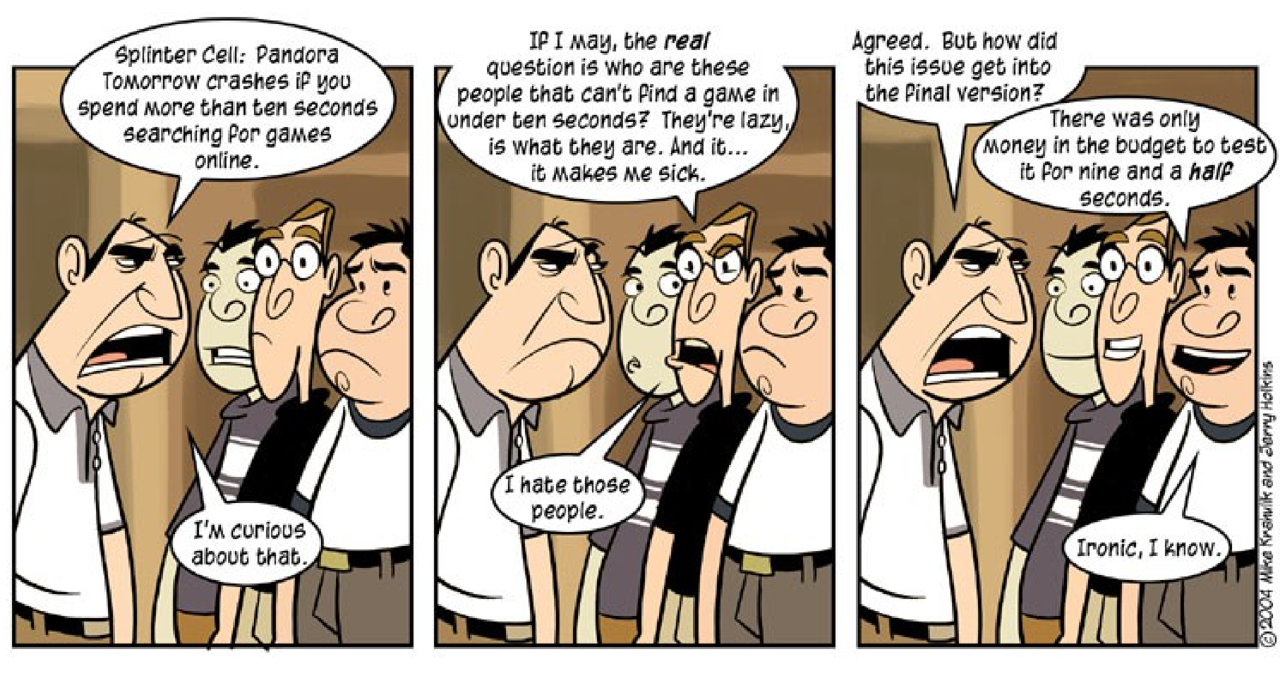
- At this point, we may actually have *too many* test cases
  - Surprisingly, this is normal in industry: you almost always have far too few or far too many!
  - Recall Google optional reading from last week
- This is especially true when using automated test generation tools
  - Which many produce many tests but lower-quality ones than humans would produce
  - A big cost problem!

# Test Suite Minimization

- Given a set of test cases and coverage information for each one, the **test suite minimization** problem is to find the minimal number of test cases that still have the maximum coverage.
- Example
  - T1 covers lines 1,2,3
  - T2 covers lines 2,3,4,5
  - T3 covers lines 1,2
  - T4 covers lines 1, 6

# Revenge of CS Theory

- You can add in details like the tests have different costs to run, but ignore that for now
- How *hard is it* to solve the test suite minimization problem?
- What is a correct algorithm for it? Can we do better?



# Questions?

- Homework 1b, 1c, 1d all due!
  - They are viewed as *much* harder than 1a