Topics for this Lecture

- Test Case Minimization
- Delta Debugging
- What is fault localization?

Debugging

- Often when debugging, we find ourselves with the problem of having an input that crashes a program but not knowing what aspect of the input is causing the program's failure. For example, a random sequence of method/constructor calls crashes a Java class.
- Isolating the cause of the failure would be enormously helpful in finding what change needs to be made to the program's code.

- There are **two** common problems:
  1. Figuring out the error which causes this failure
  2. Reducing a test case to a minimal example

- How do people solve these problems?

Delta Debugging

- One automated technique for reducing down large failing inputs is **delta debugging**.
- Based on Zeller's delta-debugging tools
- **Delta debugging** is based on the scientific method: **hypothesize**, **experiment**, and **refine**.
- **Delta debugging** automatically removes irrelevant information from a failing test case in order to attain a "minimal" bug-inducing input.

- **REMEMBER DELTA DEBUGGING: IT CAN SAVE YOU ENDLESS HOURS OF EFFORT**

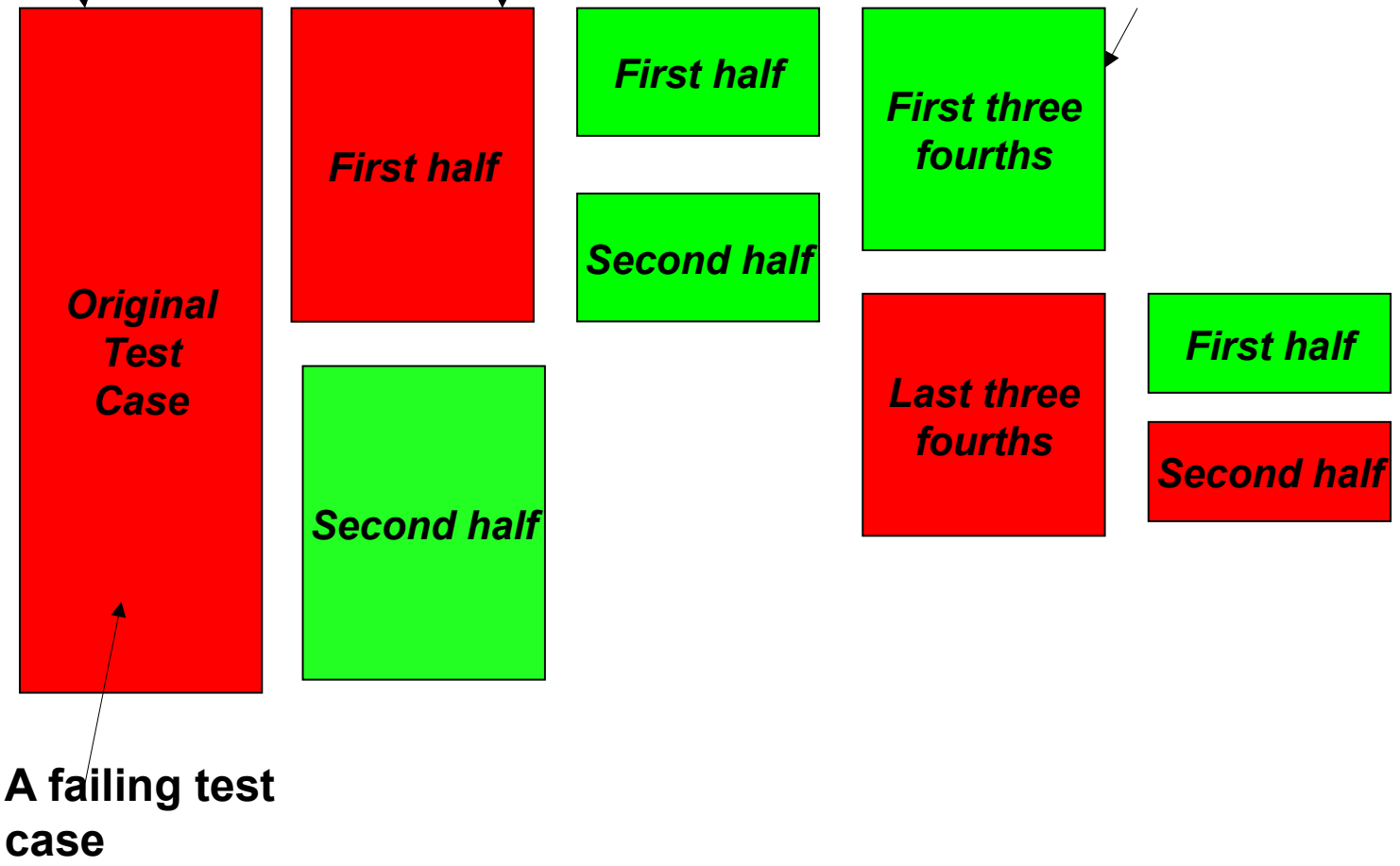
Delta Debugging

- Based on a clever modification of a "binary search" strategy

1) Split input into n subsets (initially n=2)

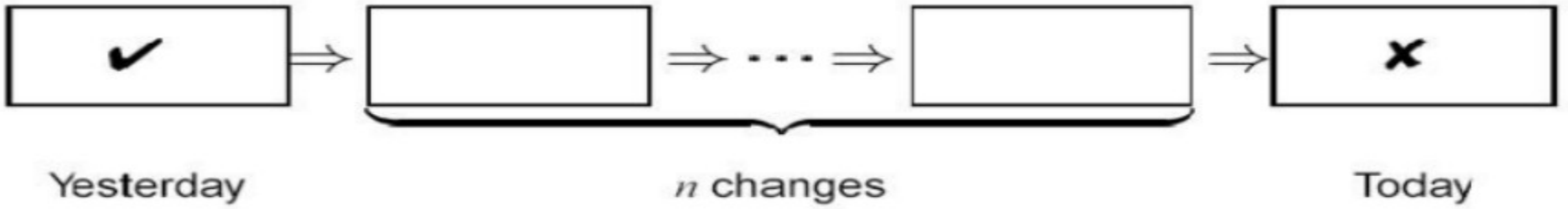
2) If removing any of these subsets fails, proceed with this subset.

3) Otherwise, increase granularity.



Oregon State University

Example - My Program Worked Yesterday, Not Today



Step	c_i	Configuration								*test*	
1	c_1	1	2	3	4	.	.	.	.	✔	
2	c_2	.	.	.	.	5	6	7	8	✘	
3	c_1	.	.	.	.	5	6	.	.	✔	
4	c_2	.	.	.	.	.	.	7	8	✘	
5	c_1	.	.	.	.	.	.	7	.	✘	7 is found
Result		.	.	.	.	.	.	7	.		

Searching a single failure (7)

Step	c_i	Configuration								*test*	
1	c_1	1	2	3	4	.	.	.	.	✔	
2	c_2	.	.	.	.	5	6	7	8	✔	
3	c_1	1	2	.	.	5	6	7	8	✔	
4	c_2	.	.	3	4	5	6	7	8	✘	
5	c_1	.	.	3	.	5	6	7	8	✘	3 is found
6	c_1	1	2	3	4	5	6	.	.	✘	
7	c_1	1	2	3	4	5	.	.	.	✔	6 is found
Result		.	.	3	.	.	6	.	.		

Searching two failure (3 and 6)

Oregon State University

Example - A Very Bad Test Case

- Printing the following file causes **Mozilla** to crash:

```
<td align=left valign=top>
<SELECT NAME="op sys" MULTIPLE SIZE=7>
<OPTION VALUE="All">All<OPTION VALUE="Windows 3.1">Windows
3.1<OPTION VALUE="Windows 95">Windows 95<OPTION VALUE="Windows
98">Windows 98<OPTION VALUE="Windows ME">Windows ME<OPTION
VALUE="Windows 2000">Windows 2000<OPTION VALUE="Windows
NT">Windows NT<OPTION VALUE="Mac System 7">Mac System 7<OPTION
VALUE="Mac System 7.5">Mac System 7.5<OPTION VALUE="Mac
System 7.6.1">Mac System 7.6.1<OPTION VALUE="Mac System 8.0">Mac
System 8.0<OPTION VALUE="Mac System 8.5">Mac System
8.5<OPTION VALUE="Mac System 8.6">Mac System 8.6<OPTION VALUE="Mac
System 9.x">Mac System 9.x<OPTION VALUE="MacOS X">MacOS
X<OPTION VALUE="Linux">Linux<OPTION VALUE="BSDI">BSDI<OPTION
VALUE="FreeBSD">FreeBSD<OPTION VALUE="NetBSD">NetBSD<OPTION
VALUE="OpenBSD">OpenBSD<OPTION VALUE="AIX">AIX<OPTION
```

Example - A Very Bad Test Case

```
VALUE="BeOS">BeOS<OPTION VALUE="HP-UX">HP-UX<OPTION
VALUE="IRIX">IRIX<OPTION VALUE="Neutrino">Neutrino<OPTION
VALUE="OpenVMS">OpenVMS<OPTION VALUE="OS/2">OS/2<OPTION
VALUE="OSF/1">OSF/1<OPTION VALUE="Solaris">Solaris<OPTION
VALUE="SunOS">SunOS<OPTION VALUE="other">other</SELECT></td>
<td align=left valign=top>
<SELECT NAME="priority" MULTIPLE SIZE=7>
<OPTION VALUE="--">--<OPTION VALUE="P1">P1<OPTION
VALUE="P2">P2<OPTION VALUE="P3">P3<OPTION VALUE="P4">P4<OPTION
VALUE="P5">P5</SELECT>
</td>
<td align=left valign=top>
<SELECT NAME="bug severity" MULTIPLE SIZE=7>
<OPTION VALUE="blocker">blocker<OPTION
VALUE="critical">critical<OPTION VALUE="major">major<OPTION
VALUE="normal">normal<OPTION VALUE="minor">minor<OPTION
VALUE="trivial">trivial<OPTION
VALUE="enhancement">enhancement</SELECT>
</tr>
</table>
```

Oregon State University

Example - A Very Bad Test Case

- Now looking at that file it is hard to figure out what the real cause of the failure is
- It would be very helpful in finding the error if we can simplify the input file and still generate the same failure
- Assume that we know that when Mozilla tries to print the following HTML input it crashes:
  <SELECT NAME="priority" MULTIPLE SIZE=7>
- How can we go about simplifying this input?
  Remove parts of the input and see if it still causes the program to crash

Example - A Very Bad Test Case

Bold parts remain in the input, the rest is removed

1	**<SELECT NAME="priority" MULTIPLE SIZE=7>**	F	N=40 characters
2	<SELECT NAME="priori**ty" MULTIPLE SIZE=7>**	P	N=20 characters
3	**<SELECT NAME="priori**ty" MULTIPLE SIZE=7>	P	N=20 characters
4	<SELECT NA**ME="priority" MULTIPLE SIZE=7>**	P	N=30 characters
5	**<SELECT NA**ME="priori**ty" MULTIPLE SIZE=7>**	F	N=30 characters
6	**<SELECT NA**ME="priority" MULTIP**LE SIZE=7>**	F	N=20 characters
7	**<SELECT NA**ME="priority" MULTIPLE SIZE=7>	P	N=10 characters
8	<SELE**CT NA**ME="priority" MULTIP**LE SIZE=7>**	P	N=10 characters
9	**<SELE**CT NAME="priority" MULTIP**LE SIZE=7>**	P	N=15 characters
10	**<SELECT NA**ME="priority" MULTIPLE SI**ZE=7>**	F	N=15 characters
11	**<SELECT NA**ME="priority" MULTIPLE SIZE=7>	P	N=10 characters
12	<S**ELECT NA**ME="priority" MULTIPLE SI**ZE=7>**	P	N=13 characters
13	**<S**ELECT **NA**ME="priority" MULTIPLE SI**ZE=7>**	P	N=13 characters

Example - A Very Bad Test Case

Bold parts remain in the input, the rest is removed

14	**<SELECT NAME="priority" MULTIPLE SIZE=7>**	P	N=13 characters
15	**<SELECT NAME="priority" MULTIPLE SIZE=7>**	P	N=13 characters
16	**<SELECT** NAME="priority" MULTIPLE **SIZE=7>**	F	N=12 characters
17	**<SELECT** NAME="priority" MULTIPLE **SIZE=7>**	F	N=11 characters
18	**<SELECT** NAME="priority" MULTIPLE SIZE=7>	F	N=10 characters
19	<**SELECT** NAME="priority" MULTIPLE SIZE=7>	P	N=9 characters
20	<S**ELECT** NAME="priority" MULTIPLE SIZE=7>	P	N=9 characters
21	<SE**LECT** NAME="priority" MULTIPLE SIZE=7>	P	N=9 characters
22	<SEL**ECT** NAME="priority" MULTIPLE SIZE=7>	P	N=9 characters
23	<SEL**ECT** NAME="priority" MULTIPLE SIZE=7>	P	N=9 characters
24	<SELE**CT** NAME="priority" MULTIPLE SIZE=7>	P	N=9 characters
25	<SELEC**T** NAME="priority" MULTIPLE SIZE=7>	P	N=9 characters
26	**<SELECT** NAME="priority" MULTIPLE SIZE=7>	F	N=10 characters

Example - A Very Bad Test Case

- After 26 tries we found that printing an HTML file which consists of:
  <SELECT> causes Mozilla to crash

- Delta debugging technique automates this approach of repeated trials for reducing the input

Causality

- When a test case fails we start debugging
- We assume that the fault (what we're really after) causes the failure
- What do we mean when we say that "A causes B"?
- We don't know
- Though it is central to everyday life – and to the aims of science
  - A real understanding of causality eludes us to this day
  - Still no non-controversial way to answer the question "does A cause B"?

What does this have to do with automated debugging??

- A **fault** is an incorrect part of a program
- In a failing test case, some fault is reached and executes
  - Causing the state of the program to be corrupted (**error**)
  - This incorrect state is propagated through the program (propagation is a series of "A causes B"s)
  - Finally, bad state is observable as a **failure** – caused by the **fault**

Fault Localization

- **Fault localization**, then, is:
  - An effort to automatically find (one of the) causes of an observable failure
  - It is inherently difficult because there are many causes of the failure that are not the fault
  - For example: if we have a program that fails/crashes but there is an option input to terminate the execution immediately, like we hit the input X.
  - One causes of the failure is we did immediately hit X. The fault localization should roll out such a cause, which is not very interested to us.

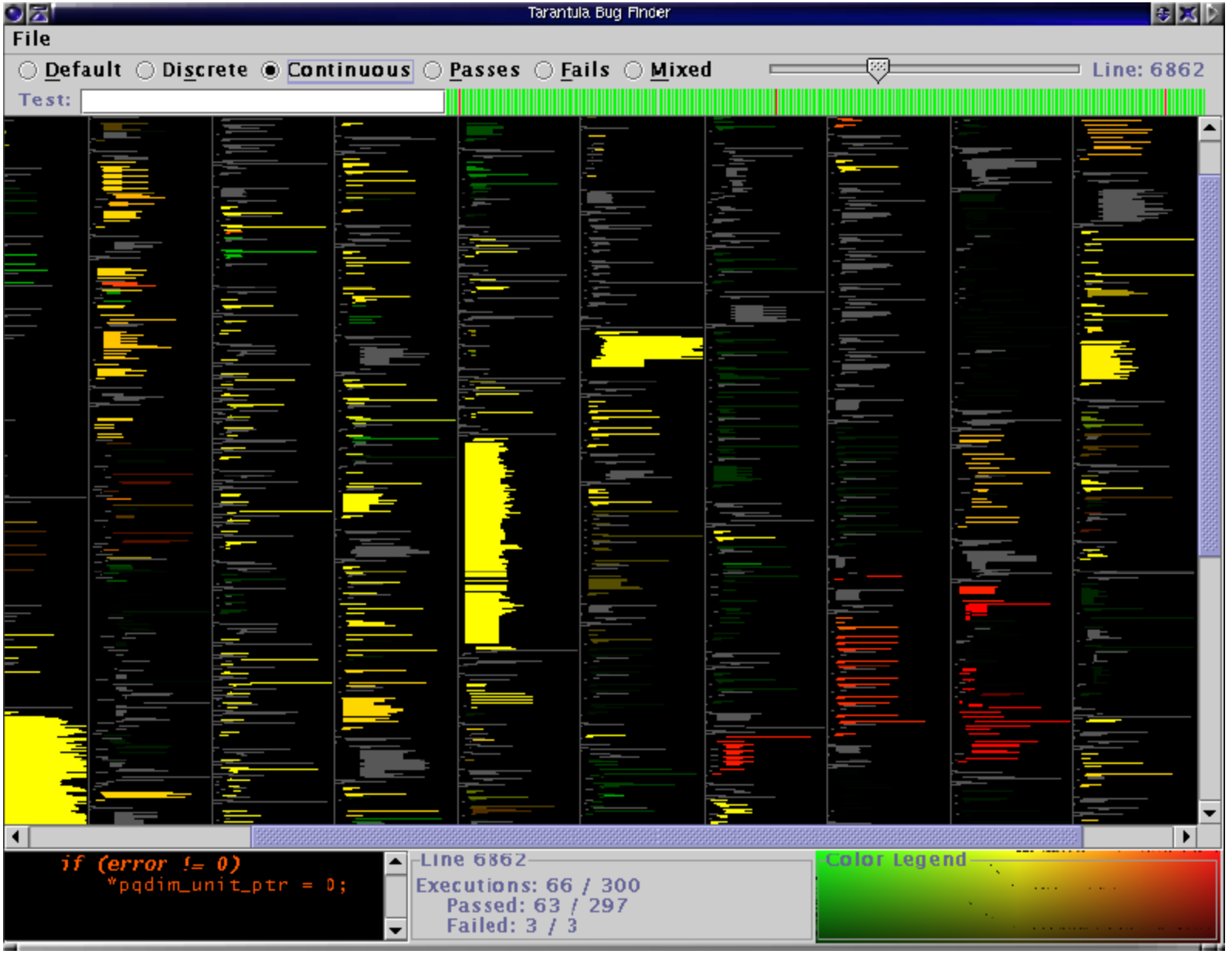
The Tarantula Approach

- Jones, Harrold (and Stasko): Tarantula

- Originally conceived of as a visualization approach: produces a picture of all source in program, colored according to how "suspicious" it is

  - Green: not likely to be faulty
  - Yellow: hrm, a little suspicious
  - Red: very suspicious, likely fault

The Tarantula Approach



Oregon State University

The Tarantula Approach

- How do we score a statement in this approach? (where do all those colors come from?)
- Again, assume we have a large set of tests, some passing, some failing
- "Coverage entity" e (e.g., statement)
  - failed(e) = # tests covering e that fail
  - passed(e) = # tests covering e that pass
  - totalfailed, totalpassed = total number of failed and passed test cases

The Tarantula Approach

- How do we score a statement in this approach? (where do all those colors come from?) . The suspiciousness of a statement (***e***) is computed by the following equation:

$$suspiciousness(e) = \frac{\frac{failed(e)}{totalfailed}}{\frac{passed(e)}{totalpassed} + \frac{failed(e)}{totalfailed}}$$

- ***passed***(***e***) is the number of passed test cases that executed statement ***e*** one or more times.
- ***failed***(***e***) is the number of failed test cases that executed statement ***e*** one or more times.
- ***totalpassed*** and ***totalfailed*** are the total numbers of test cases that pass and fail, respectively, in the entire test suite.

The Tarantula Approach

$$suspiciousness(e) = \frac{\frac{failed(e)}{totalfailed}}{\frac{passed(e)}{totalpassed} + \frac{failed(e)}{totalfailed}}$$

- Not very suspicious =0: appears in almost every passing test and almost every failing test
- Highly suspicious = 1: appears much more frequently in failing than passing tests

The Tarantula Approach

```
mid()
    int x, y, z, m;
1   read (x, y, z);
2   m = z;
3   if (y < z)
4     if (x < y)
5       m = y;
6     else if (x < z)
7       m = y;
8   else
9     if (x > y)
10      m = y;
11    else if (x > z)
12      m = x;

13    print (m);
```

Program **mid()** takes three integers (x,y, and z) as input and outputs the **median** value.

The program contains a fault on **line 7**—this line should read "**m = x**;".

The Tarantula Approach

Run some tests. . .

$$suspiciousness(e) = \frac{\frac{failed(e)}{totalfailed}}{\frac{passed(e)}{totalpassed} + \frac{failed(e)}{totalfailed}}$$

(3,3,5) (1,2,3) (3,2,1) (5,5,5) (5,3,4) (2,1,3)

```
mid()
    int x, y, z, m;
1   read (x, y, z)
2   m = z;
3   if (y < z)
4     if (x < y)
5       m = y;
6     else if (x < z)
7       m = y;
8   else
9     if (x > y)
10      m = y;
11    else if (x > z)
12      m = x;
13    print (m);
```

Oregon State UNIVERSITY

The Tarantula Approach

Run some tests. . .

Look at whether they pass or fail

$$suspicious\,ness(e) = \frac{\frac{failed(e)}{totalfaile\,d}}{\frac{passed(e)}{totalpasse\,d} + \frac{failed(e)}{totalfaile\,d}}$$

(3,3,5) (1,2,3) (3,2,1) (5,5,5) (5,3,4) (2,1,3)

```
mid()
    int x, y, z, m;
1   read (x, y, z)
2   m = z;
3   if (y < z)
4     if (x < y)
5       m = y;
6     else if (x < z)
7       m = y;
8   else
9     if (x > y)
10      m = y;
11    else if (x > z)
12      m = x;
13    print (m);
```

The Tarantula Approach

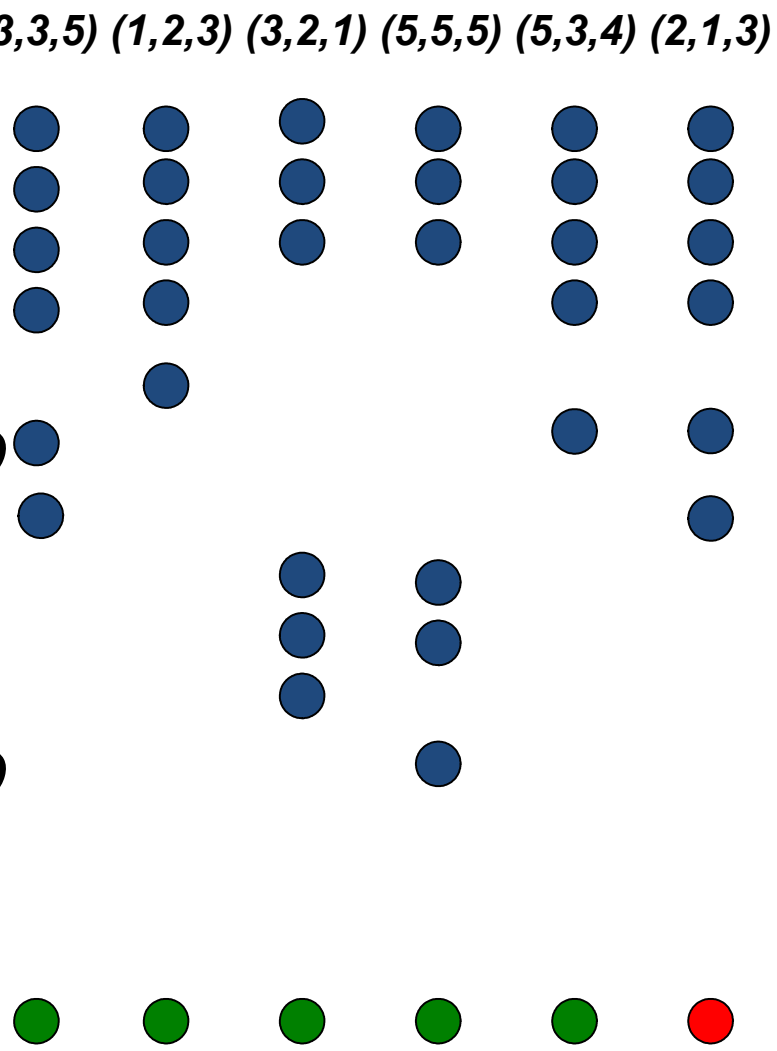
Run some tests. . .

Look at whether they pass or fail

Look at coverage of entities

$$suspiciousness(e) = \frac{\frac{failed(e)}{totalfailed}}{\frac{passed(e)}{totalpassed} + \frac{failed(e)}{totalfailed}}$$

```
mid()
    int x, y, z, m;
1   read (x, y, z)
2   m = z;
3   if (y < z)
4     if (x < y)
5       m = y;
6     else if (x < z)
7       m = y;
8   else
9     if (x > y)
10      m = y;
11    else if (x > z)
12      m = x;
13    print (m);
```



Oregon State University

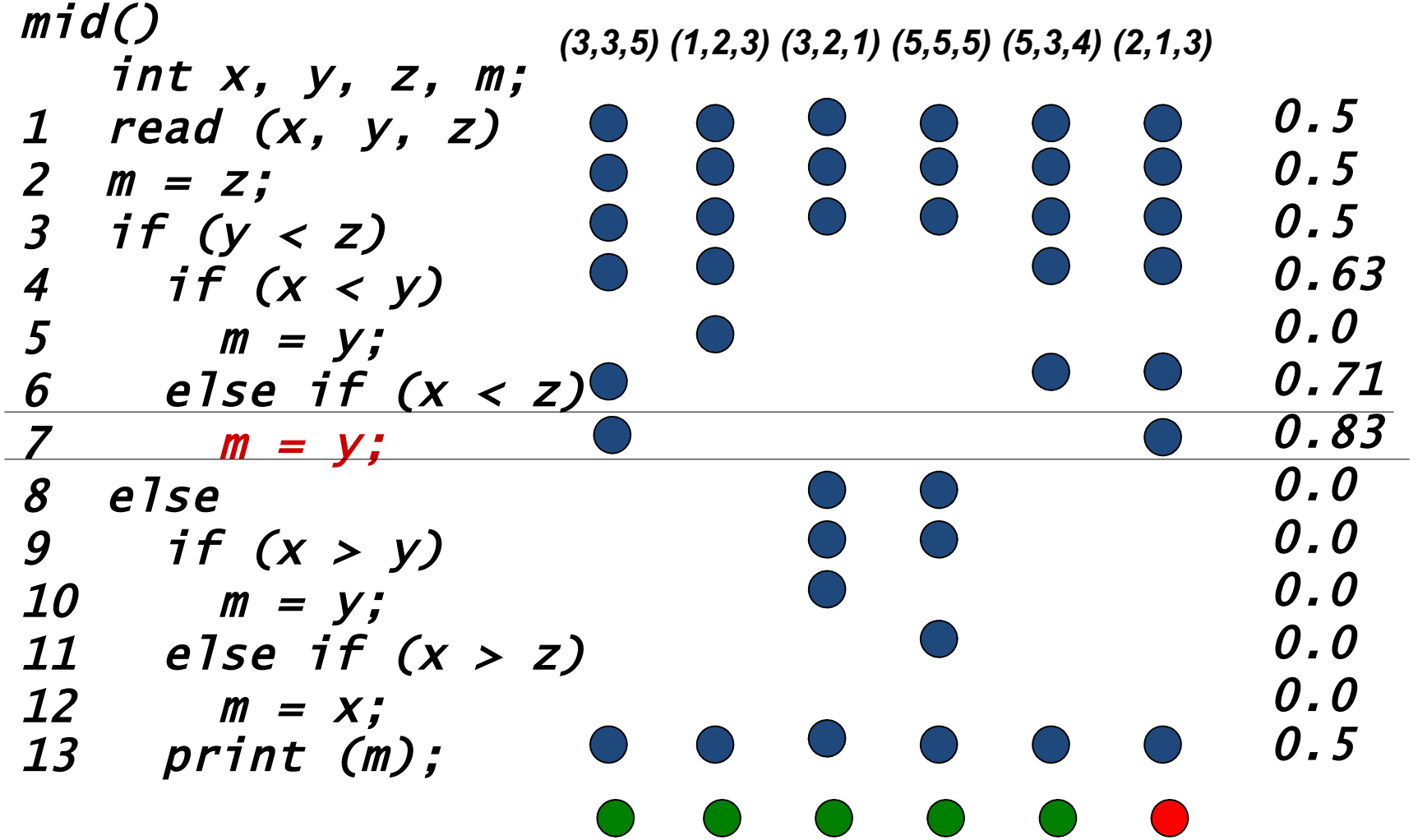
The Tarantula Approach

Run some tests. . .

Look at whether they pass or fail

Look at coverage of entities

$$suspiciousness(e) = \frac{\frac{failed(e)}{totalfailed}}{\frac{passed(e)}{totalpassed} + \frac{failed(e)}{totalfailed}}$$



Compute suspiciousness using the formula

Fault is indeed most suspicious!

Oregon State University

The Tarantula Approach

- Obvious benefits:
  - Provides a ranking of every statement, instead of just a set of nodes – directions on where to look next
    - Numerical, even – how much more suspicious is X than Y?
  - The pretty visualization may be quite helpful in seeing relationships between suspicious statements

References:

Zeller, Andreas. "Yesterday, my program worked. Today, it does not. Why?." Software Engineering —ESEC/FSE'99. Springer Berlin Heidelberg, 1999.
http://classes.engr.oregonstate.edu/eecs/summer2015/cs362-002/
Renieres, Manos, and Steven P. Reiss. "Fault localization with nearest neighbor queries." Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on. IEEE, 2003.
Jones, James A., and Mary Jean Harrold. "Empirical evaluation of the tarantula automatic fault-localization technique." Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering. ACM, 2005.
https://classes.soe.ucsc.edu/cmps290g/Winter04/lectures/flanagan-290g-8.pdf