CS409
Software Testing

TAN, Shin Hwei

陈馨慧

Southern University of Science and Technology

Slides adapted from CS4218 in NUS

Administrative Info

- Project Proposal posted. Due on 4 December 2020
- No bugs posted in GitHub discussion so far
- Submit bug reports for bonus early!
  - If the bug is fixed or is old bug, then you couldn’t get the bonus points

Mutation testing in Google

- Use as a code review tool in Google
  - If an automated diff analyzer finding (e.g. a living mutant) is not useful, developers can report that with a single click on the finding. If any o a finding to be impor at to the diff autho hown in Figure 1.

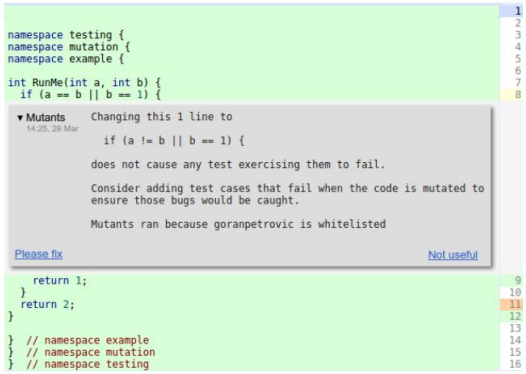


Figure 1: Mutant finding shown in the Critique - Google code review tool

Recap: Syntax-Based Coverage Criteria

Mutation Coverage (MC) : For each $m \in M$, TR contains exactly one requirement, to kill m.

- The RIPR model from chapter 2:
  - *Reachability* : The test causes the faulty statement to be reached (in mutation – the mutated statement)
  - *Infection* : The test causes the faulty statement to result in an incorrect state
  - *Propagation* : The incorrect state propagates to incorrect output
  - *Revealability* : The tester must observe part of the incorrect output
- The RIPR model leads to two variants of mutation coverage …

Strong Versus Weak Mutation

```
1    boolean isEven (int X)
2    {
3        if (X < 0)
4            X = 0 - X;
Δ 4          X = 0;
5        if (double) (X/2) == ((double) X) / 2.0
6            return (true);
7        else
8            return (false);
9    }
```

Reachability : X < 0

Infection : X != 0

(X = -6) will kill mutant 4 under weak mutation

Propagation :

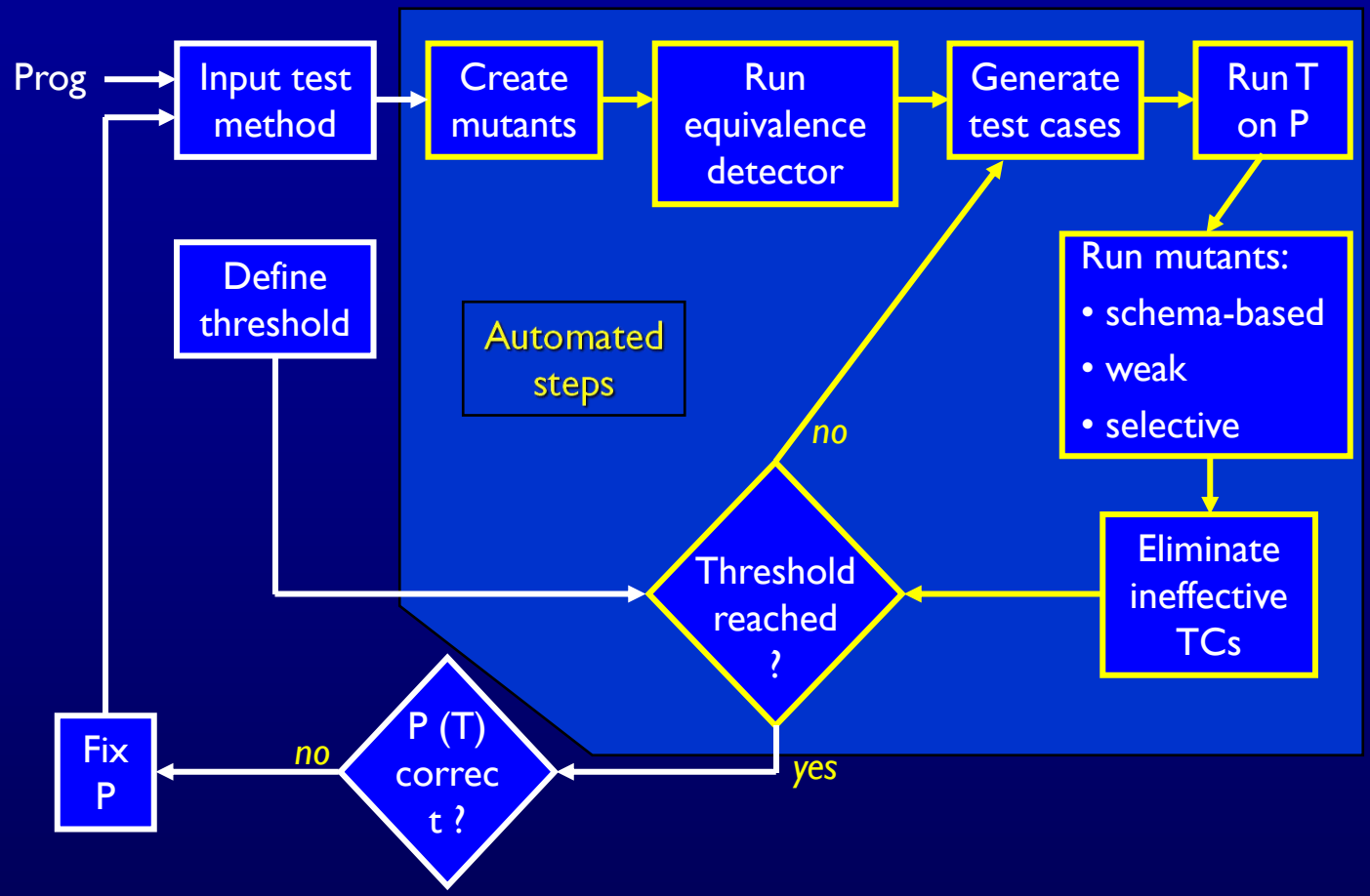
((double) ((0-X)/2) == ((double) 0-X) / 2.0)

!= ((double) (0/2) == ((double) 0) / 2.0)

That is, X is not even …

Thus (X = -6) does *not* kill the mutant under strong mutation

Testing Programs with Mutation



Why Mutation Works

Fundamental Premise of Mutation Testing

If the software contains a fault, there will usually be a set of mutants that can only be killed by a test case that also detects that fault

- This is not an absolute !
- The mutants guide the tester to an effective set of tests
- A very challenging problem :
  - Find a fault and a set of mutation-adequate tests that do not find the fault
- Of course, this depends on the mutation operators …

Designing Mutation Operators

- At the method level, mutation operators for different programming languages are similar
- Mutation operators do one of two things :
  - Mimic typical programmer mistakes (incorrect variable name)
  - Encourage common test heuristics (cause expressions to be 0)
- Researchers design lots of operators, then experimentally *select* the most useful

Effective Mutation Operators

If tests that are created specifically to kill mutants created by a collection of mutation operators *O* = {*o1*, *o2*, ...} also kill mutants created by all remaining mutation operators with very high probability, then *O* defines an *effective* set of mutation operators

Mutation Operators for Java

1. ABS — Absolute Value Insertion
2. AOR — Arithmetic Operator Replacement
3. ROR — Relational Operator Replacement
4. COR — Conditional Operator Replacement
5. SOR — Shift Operator Replacement
6. LOR — Logical Operator Replacement
7. ASR — Assignment Operator Replacement
8. UOI — Unary Operator Insertion
9. UOD — Unary Operator Deletion
10. SVR — Scalar Variable Replacement
11. BSR — Bomb Statement Replacement

Full definitions …

Mutation Operators for Java

1. ABS — Absolute Value Insertion:

Each arithmetic expression (and subexpression) is modified by the functions *abs()*, *negAbs()*, and *failOnZero()*.

Examples:

```
      a = m * (o + p);
Δ1    a = abs (m * (o + p));
Δ2    a = m * abs ((o + p));
Δ3    a = failOnZero (m * (o + p));
```

2. AOR — Arithmetic Operator Replacement:

Each occurrence of one of the arithmetic operators +, —, *, ⁄, and % is replaced by each of the other operators. In addition, each is replaced by the special mutation operators *leftOp*, and *rightOp*.

Examples:

```
      a = m * (o + p);
Δ1    a = m + (o + p);
Δ2    a = m * (o * p);
Δ3    a = m leftOp (o + p);
```

Mutation Operators for Java (2)

3. ROR — Relational Operator Replacement:

Each occurrence of one of the relational operators (<, ≤, >, ≥, =, ≠) is replaced by each of the other operators and by *falseOp* and *trueOp*.

Examples:

```
      if (X <= Y)
Δ1    if (X > Y)
Δ2    if (X < Y)
Δ3    if (X falseOp Y)  // always returns false
```

4. COR — Conditional Operator Replacement:

Each occurrence of one of the logical operators (and - &&, or - || , and with no conditional evaluation - &, or with no conditional evaluation - |, not equivalent - ^) is replaced by each of the other operators; in addition, each is replaced by *falseOp*, *trueOp*, *leftOp*, and *rightOp*.

Examples:

```
      if (X <= Y && a > 0)
Δ1    if (X <= Y || a > 0)
Δ2    if (X <= Y leftOp a > 0) // returns result of left clause
```

Mutation Operators for Java (4)

5. SOR — Shift Operator Replacement:

Each occurrence of one of the shift operators <<, >>, and >>> is replaced by each of the other operators. In addition, each is replaced by the special mutation operator *leftOp*.

Examples:

```
      byte b = (byte) 16;
      b = b >> 2;
Δ1    b = b << 2;
Δ2    b = b leftOp 2; // result is b
```

6. LOR — Logical Operator Replacement:

Each occurrence of one of the logical operators (bitwise and - &, bitwise or - |, exclusive or - ^) is replaced by each of the other operators; in addition, each is replaced by leftOp and rightOp.

Examples:

```
      int a = 60;    int b = 13;
      int c = a & b;
Δ1    int c = a | b;
Δ2    int c = a rightOp b; // result is b
```

Mutation Operators for Java (5)

7. ASR — Assignment Operator Replacement:

Each occurrence of one of the assignment operators (=, +=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=, >>>=) is replaced by each of the other operators.

Examples:

```
      a = m * (o + p);
Δ1    a += m * (o + p);
Δ2    a *= m * (o + p);
```

8. UOI — Unary Operator Insertion:

Each unary operator (arithmetic +, arithmetic -, conditional !, logical ~) is inserted in front of each expression of the correct type.

Examples:

```
      a = m * (o + p);
Δ1    a = m * -(o + p);
Δ2    a = -(m * (o + p));
```

Mutation Operators for Java (6)

9. UOD — Unary Operator Deletion:

Each unary operator (arithmetic +, arithmetic -, conditional !, logical~) is deleted.

Examples:

```
      if !(X <= Y && !Z)
Δ1    if (X > Y && !Z)
Δ2    if !(X < Y && Z)
```

10. SVR — Scalar Variable Replacement:

Each variable reference is replaced by every other variable of the appropriate type that is declared in the current scope.

Examples:

```
      a = m * (o + p);
Δ 1   a = o * (o + p);
Δ 2   a = m * (m + p);
Δ 3   a = m * (o + o);
Δ 4   p = m * (o + p);
```

Mutation Operators for Java (7)

11. BSR — Bomb Statement Replacement:

Each statement is replaced by a special Bomb() function.

Example:

```
    a = m * (o + p);
Δ1  Bomb() // Raises exception when reached
```

Form a team with 1-2 students

IN CLASS EXERCISE

```
public static int cal (int month1, int day1, int month2, int day2, int year){
    int numDays;
Δ1 if (month2!=month1)
    if (month2 == month1)
        numDays  = day2 - day1;
    else{
      // Skip month 0.
      int daysIn[] = {0, 31, 0, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
      // Are we in a leap year?
      int m4 = year % 4;
      int m100 = year % 100;
      int m400 = year % 400;
if ((m4 != 0) || ((m100 == 0) && (m400 != 0)))
        daysIn[2] = 28;
      else
        daysIn[2] = 29;

      // start with days in the two months
      numDays = day2 + (daysIn[month1] - day1);

      // add the days in the intervening months
      for (int i = month1 + 1; i <= month2-1; i++)
        numDays = daysIn[i] + numDays;
    }
    return (numDays);}
```

What is the mutation operator in mutation Δ1 ?

1. AOR — Arithmetic Operator Replacement
2. ROR — Relational Operator Replacement
3. COR — Conditional Operator Replacement
4. SOR — Shift Operator Replacement
5. LOR — Logical Operator Replacement
6. ASR — Assignment Operator Replacement

```
public static int cal (int month1, int day1, int month2, int day2, int year){
    int numDays;
Δ1 if (month2==month1)
    if (month2 != month1)
      numDays  = day2 - day1;
    else{
      // Skip month 0.
      int daysIn[] = {0, 31, 0, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
      // Are we in a leap year?
      int m4 = year % 4;
      int m100 = year % 100;
      int m400 = year % 400;
if ((m4 != 0) || ((m100 == 0) && (m400 != 0)))
        daysIn[2] = 28;
      else
        daysIn[2] = 29;

      // start with days in the two months
      numDays = day2 + (daysIn[month1] - day1);

      // add the days in the intervening months
      for (int i = month1 + 1; i <= month2-1; i++)
        numDays = daysIn[i] + numDays;
    }
    return (numDays);}
```

1. Find a test that weakly kills the mutant, but not strongly
2. Generalize :What must be true to weakly kill the mutant, but not strongly?
3. Try to write down the conditions needed to (i) reach the mutated statement, (ii) infect the program state, and (iii) propagate to output

```
public static int cal (int month1, int day1, int month2, int day2, int year){
    int numDays;
  if (month2 == month1)
  Δ2  numDays = day2 + day1;
      numDays  = day2 - day1;
    else{
      // Skip month 0.
      int daysIn[] = {0, 31, 0, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
      // Are we in a leap year?
      int m4 = year % 4;
      int m100 = year % 100;
      int m400 = year % 400;
if ((m4 != 0) || ((m100 == 0) && (m400 != 0)))
        daysIn[2] = 28;
      else
        daysIn[2] = 29;

      // start with days in the two months
      numDays = day2 + (daysIn[month1] - day1);

      // add the days in the intervening months
      for (int i = month1 + 1; i <= month2-1; i++)
        numDays = daysIn[i] + numDays;
    }
    return (numDays);}
```

What is the mutation operator in mutation Δ1 ?

1. AOR — Arithmetic Operator Replacement
2. ROR — Relational Operator Replacement
3. COR — Conditional Operator Replacement
4. SOR — Shift Operator Replacement
5. LOR — Logical Operator Replacement
6. ASR — Assignment Operator Replacement

```
public static int cal (int month1, int day1, int month2, int day2, int year){
    int numDays;
  if (month2 == month1)
  Δ2  numDays = day2 + day1;
      numDays  = day2 - day1;
    else{
      // Skip month 0.
      int daysIn[] = {0, 31, 0, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
      // Are we in a leap year?
      int m4 = year % 4;
      int m100 = year % 100;
      int m400 = year % 400;
if ((m4 != 0) || ((m100 == 0) && (m400 != 0)))
        daysIn[2] = 28;
      else
        daysIn[2] = 29;

      // start with days in the two months
      numDays = day2 + (daysIn[month1] - day1);

      // add the days in the intervening months
      for (int i = month1 + 1; i <= month2-1; i++)
        numDays = daysIn[i] + numDays;
    }
    return (numDays);}
```

1. Find a test that weakly kills the mutant, but not strongly
2. Generalize :What must be true to weakly kill the mutant, but not strongly?
3. Try to write down the conditions needed to (i) reach the mutated statement, (ii) infect the program state, and (iii) propagate to output

Summary : Subsuming Other Criteria

- Mutation is widely considered the strongest test criterion
  - And most expensive !
  - By far the most test requirements (each mutant)
  - Usually the most tests
- Mutation subsumes other criteria by including specific mutation operators
- Subsumption can only be defined for weak mutation – other criteria only impose local requirements
  - Node coverage, Edge coverage, Clause coverage
  - General active clause coverage: Yes–Requirement on single tests
  - Correlated active clause coverage: No–Requirement on test *pairs*
  - All-defs data flow coverage

Mutation testing in Google

- Use as a code review tool in Google
  - If an automated diff analyzer finding (e.g. a living mutant) is not useful, developers can report that with a single click on the finding. If any of the reviewers consider a finding to be important, they can indicate that to the diff author with a single click, as shown in Figure 1.

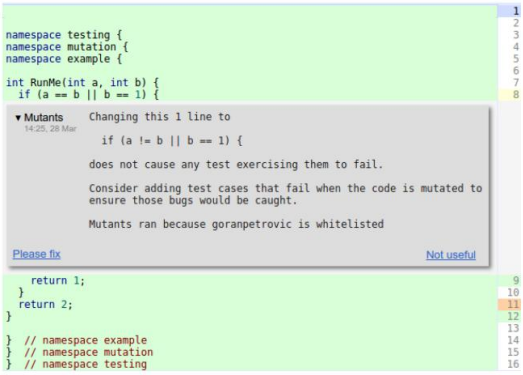


Figure 1: Mutant finding shown in the Critique - Google code review tool

Automated Debugging

Automated Debugging

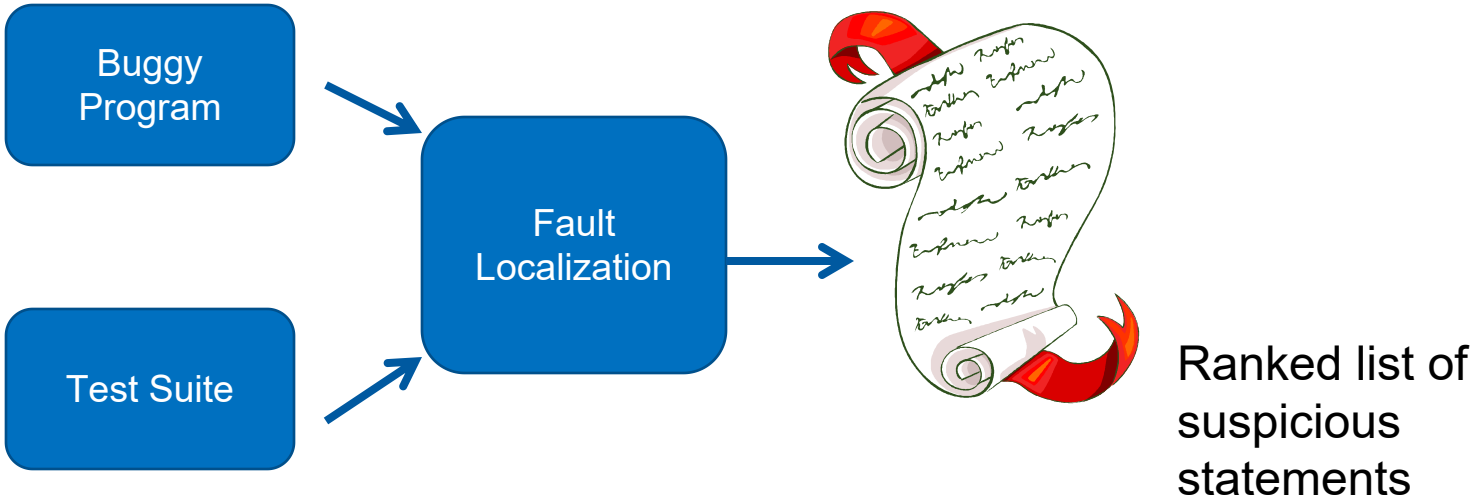
Debugging techniques that use program executions in different ways:

- Statistical Fault Localization
- Dynamic Slicing
- Delta Debugging

Automated Debugging

Statistical Fault Localization

Statistical Fault localization



Assign scores to program statements based on their occurrence in passing / failing tests. ***Correlation equals causation!***

$$\text{Score}(s) = \frac{\frac{\text{fail}(s)}{\text{allfail}}}{\frac{\text{fail}(s)}{\text{allfail}} + \frac{\text{pass}(s)}{\text{allpass}}}$$

An example of scoring scheme [Tarantula]

Producing Ranked Bug report

- We use the Tarantula toolkit.
- Given a test-suite T

$$\text{Score}(s) = \frac{\frac{\text{fail}(s)}{\text{allfail}}}{\frac{\text{fail}(s)}{\text{allfail}} + \frac{\text{pass}(s)}{\text{allpass}}}$$

  - fail(s) ≡ # of failing executions in which s occurs
  - pass(s) ≡ # of passing executions in which s occurs
  - allfail ≡ Total # of failing executions
  - allpass ≡ Total # of passing executions
    - allfail + allpass = |T|
- Can also use other metric like Ochiai.

Name	Formula	Name	Formula
Jaccard	$\frac{a_{ef}}{a_{ef}+a_{nf}+a_{ep}}$	Anderberg	$\frac{a_{ef}}{a_{ef}+2(a_{nf}+a_{ep})}$
Sørensen-Dice	$\frac{2a_{ef}}{2a_{ef}+a_{nf}+a_{ep}}$	Dice	$\frac{2a_{ef}}{a_{ef}+a_{nf}+a_{ep}}$
Kulczynski1	$\frac{a_{ef}}{a_{nf}+a_{ep}}$	Kulczynski2	$\frac{1}{2}\left(\frac{a_{ef}}{a_{ef}+a_{nf}}+\frac{a_{ef}}{a_{ef}+a_{ep}}\right)$
Russell and Rao	$\frac{a_{ef}}{a_{ef}+a_{nf}+a_{ep}+a_{np}}$	Hamann	$\frac{a_{ef}+a_{np}-a_{nf}-a_{ep}}{a_{ef}+a_{nf}+a_{ep}+a_{np}}$
Simple Matching	$\frac{a_{ef}+a_{np}}{a_{ef}+a_{nf}+a_{ep}+a_{np}}$	Sokal	$\frac{2(a_{ef}+a_{np})}{2(a_{ef}+a_{np})+a_{nf}+a_{ep}}$
M1	$\frac{a_{ef}+a_{np}}{a_{nf}+a_{ep}}$	M2	$\frac{a_{ef}}{a_{ef}+a_{np}+2(a_{nf}+a_{ep})}$
Rogers-Tanimoto	$\frac{a_{ef}+a_{np}}{a_{ef}+a_{np}+2(a_{nf}+a_{ep})}$	Goodman	$\frac{2a_{ef}-a_{nf}-a_{ep}}{2a_{ef}+a_{nf}+a_{ep}}$
Hamming etc.	$a_{ef}+a_{np}$	Euclid	$\sqrt{a_{ef}+a_{np}}$
Ochiai	$\frac{a_{ef}}{\sqrt{(a_{ef}+a_{nf})(a_{ef}+a_{ep})}}$	Overlap	$\frac{a_{ef}}{\min(a_{ef},a_{nf},a_{ep})}$
Tarantula	$\frac{\frac{a_{ef}}{a_{ef}+a_{nf}}}{\frac{a_{ef}}{a_{ef}+a_{nf}}+\frac{a_{ep}}{a_{ep}+a_{np}}}$	Zoltar	$\frac{a_{ef}}{a_{ef}+a_{nf}+a_{ep}+\frac{10000a_{nf}a_{ep}}{a_{ef}}}$
Ample	$\left\lvert\frac{a_{ef}}{a_{ef}+a_{nf}}-\frac{a_{ep}}{a_{ep}+a_{np}}\right\rvert$	Wong1	a_{ef}
Wong2	$a_{ef}-a_{ep}$		
Wong3	$a_{ef}-h$, where $h=\begin{cases} a_{ep} & \text{if } a_{ep}\leq 2 \\ 2+0.1(a_{ep}-2) & \text{if } 2<a_{ep}\leq 10 \\ 2.8+0.001(a_{ep}-10) & \text{if } a_{ep}>10 \end{cases}$		
Ochiai2	$\frac{a_{ef}a_{np}}{\sqrt{(a_{ef}+a_{ep})(a_{np}+a_{nf})(a_{ef}+a_{nf})(a_{ep}+a_{np})}}$		
Geometric Mean	$\frac{a_{ef}a_{np}-a_{nf}a_{ep}}{\sqrt{(a_{ef}+a_{ep})(a_{np}+a_{nf})(a_{ef}+a_{nf})(a_{ep}+a_{np})}}$		
Harmonic Mean	$\frac{(a_{ef}a_{np}-a_{nf}a_{ep})((a_{ef}+a_{ep})(a_{np}+a_{nf})+(a_{ef}+a_{nf})(a_{ep}+a_{np}))}{(a_{ef}+a_{ep})(a_{np}+a_{nf})(a_{ef}+a_{nf})(a_{ep}+a_{np})}$		
Arithmetic Mean	$\frac{2a_{ef}a_{np}-2a_{nf}a_{ep}}{(a_{ef}+a_{ep})(a_{np}+a_{nf})+(a_{ef}+a_{nf})(a_{ep}+a_{np})}$		
Cohen	$\frac{2a_{ef}a_{np}-2a_{nf}a_{ep}}{(a_{ef}+a_{ep})(a_{np}+a_{ep})+(a_{ef}+a_{nf})(a_{nf}+a_{np})}$		
Scott	$\frac{4a_{ef}a_{np}-4a_{nf}a_{ep}-(a_{nf}-a_{ep})^2}{(2a_{ef}+a_{nf}+a_{ep})(2a_{np}+a_{nf}+a_{ep})}$		
Fleiss	$\frac{4a_{ef}a_{np}-4a_{nf}a_{ep}-(a_{nf}-a_{ep})^2}{(2a_{ef}+a_{nf}+a_{ep})+(2a_{np}+a_{nf}+a_{ep})}$		
Rogot1	$\frac{1}{2}\left(\frac{a_{ef}}{2a_{ef}+a_{nf}+a_{ep}}+\frac{a_{np}}{2a_{np}+a_{nf}+a_{ep}}\right)$		
Rogot2	$\frac{1}{4}\left(\frac{a_{ef}}{a_{ef}+a_{ep}}+\frac{a_{ef}}{a_{ef}+a_{nf}}+\frac{a_{np}}{a_{np}+a_{ep}}+\frac{a_{np}}{a_{np}+a_{nf}}\right)$		

Can use several other available metrics for ranking statements, e.g. Ochiai metric

$$\text{Score}(s)=\frac{\text{fail}(s)}{\sqrt{\text{allfail}*(\text{fail}(s)+\text{pass}(s))}}$$

A model for spectra-based software diagnosis, Naish et. al., TOSEM 20(3), 2011.

Example 1

$$\text{Score}(s) = \frac{\frac{\text{fail}(s)}{\text{allfail}}}{\frac{\text{fail}(s)}{\text{allfail}} + \frac{\text{pass}(s)}{\text{allpass}}}$$

	mid() { int x,y,z,m;	3,3,5	1,2,3	3,2,1	5,5,5	5,3,4	2,1,3	suspic	rank
1:	read("Enter 3 numbers:",x,y,z);	●	●	●	●	●	●	0.5	7
2:	m = z;	●	●	●	●	●	●	0.5	7
3:	if (y<z)	●	●	●	●	●	●	0.5	7
4:	if (x<y)	●	●			●	●	0.63	3
5:	m = y;		●					0.0	13
6:	else if (x<z)	●				●	●	0.71	2
7:	m = y; // *** bug ***	●					●	0.83	1
8:	else			●	●			0.0	13
9:	if (x>y)			●	●			0.0	13
10:	m = y;			●				0.0	13
11:	else if (x>z)				●			0.0	13
12:	m = x;							0.0	13
13:	print("Middle number is:",m);	●	●	●	●	●	●	0.5	7
	} Pass/Fail Status	P	P	P	P	P	F		

Executed mostly by failing test

- High suspiciousness score

Example 2

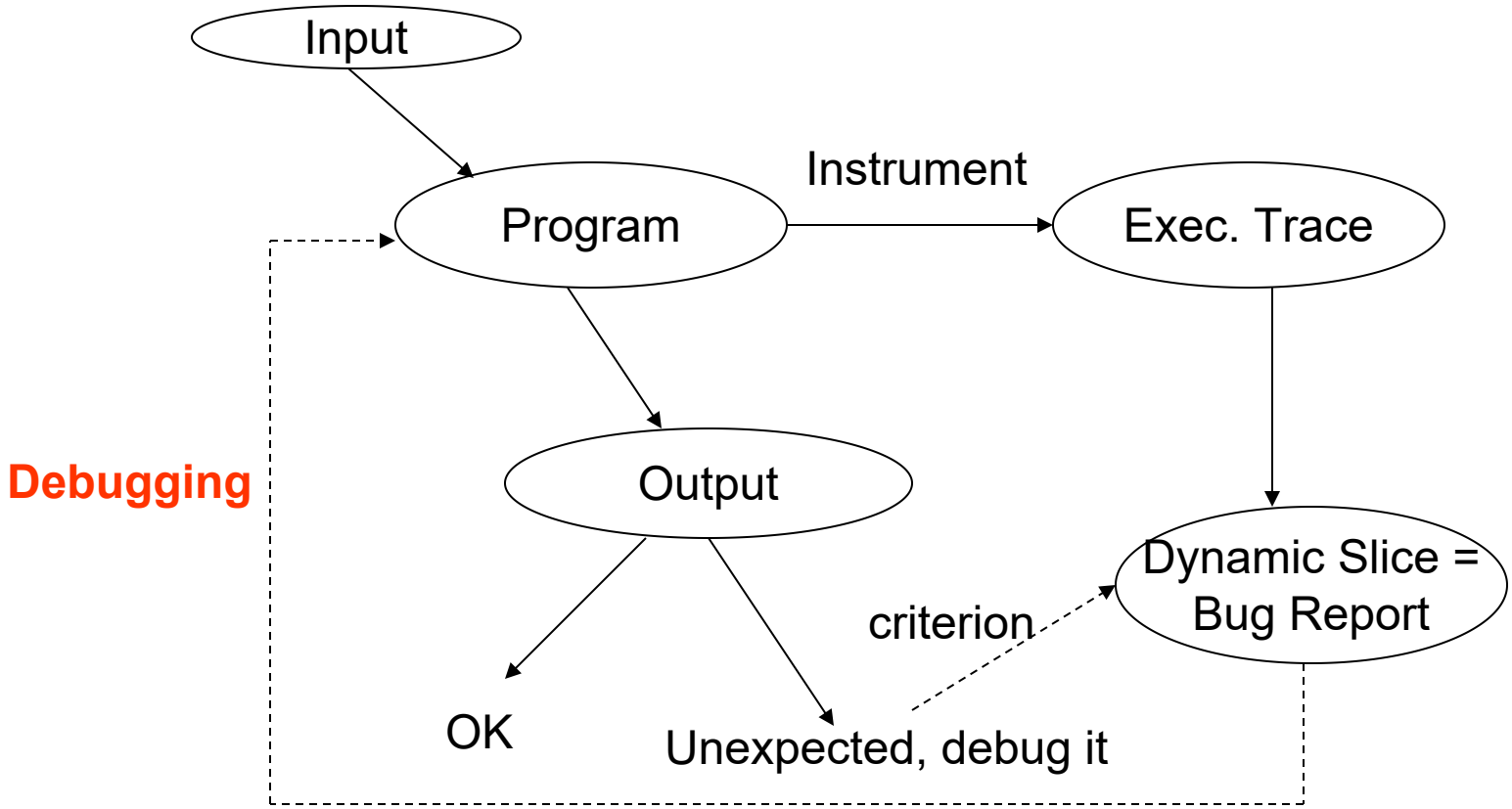
		Test Cases			
		T15	T16	T17	T18
1	int count; int n; Ele *proc; List *src_queue, *dest_queue; if (prio >= MAXPRIO) /*maxprio=3*/	●	●	●	●
2	{return;}		●	●	●
3	src_queue = prio_queue[prio]; dest_queue = prio_queue[prio+1]; count = src_queue->mem_count; *if (count > 1) /* Bug*//* supposed : count>0*/ {*	●	●	●	●
4	n = (int) (count*ratio + 1); proc = find_nth(src_queue, n); if (proc) {		●	●	
5	src_queue = del_ele(src_queue, proc); proc->priority = prio; dest_queue = append_ele(dest_queue, proc); }		●	●	
	Pass/Fail of Test Case Execution :	Pass	Pass	Pass	Fail

Which basic block is the most suspicious?

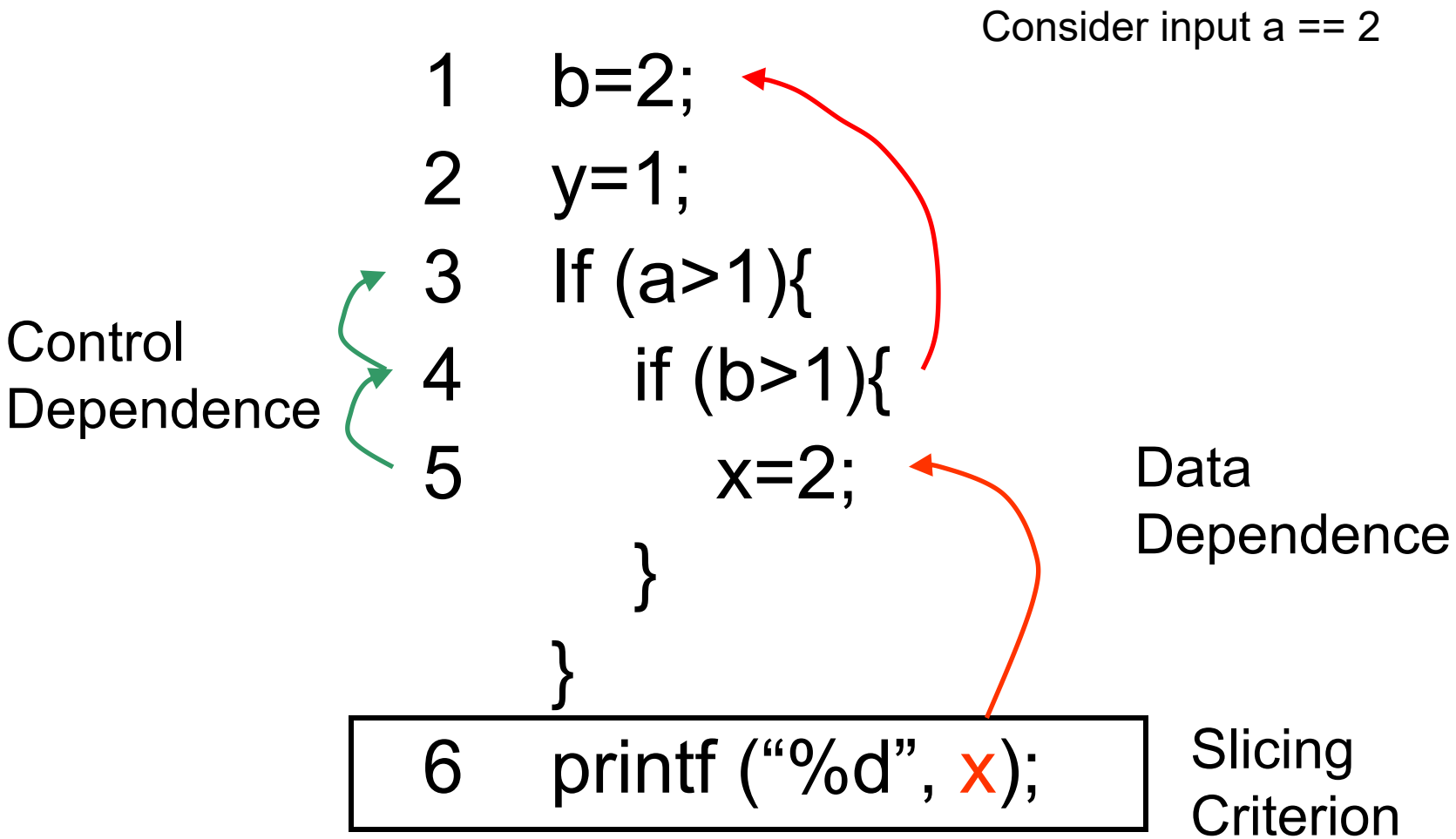
Automated Debugging

Dynamic Slicing

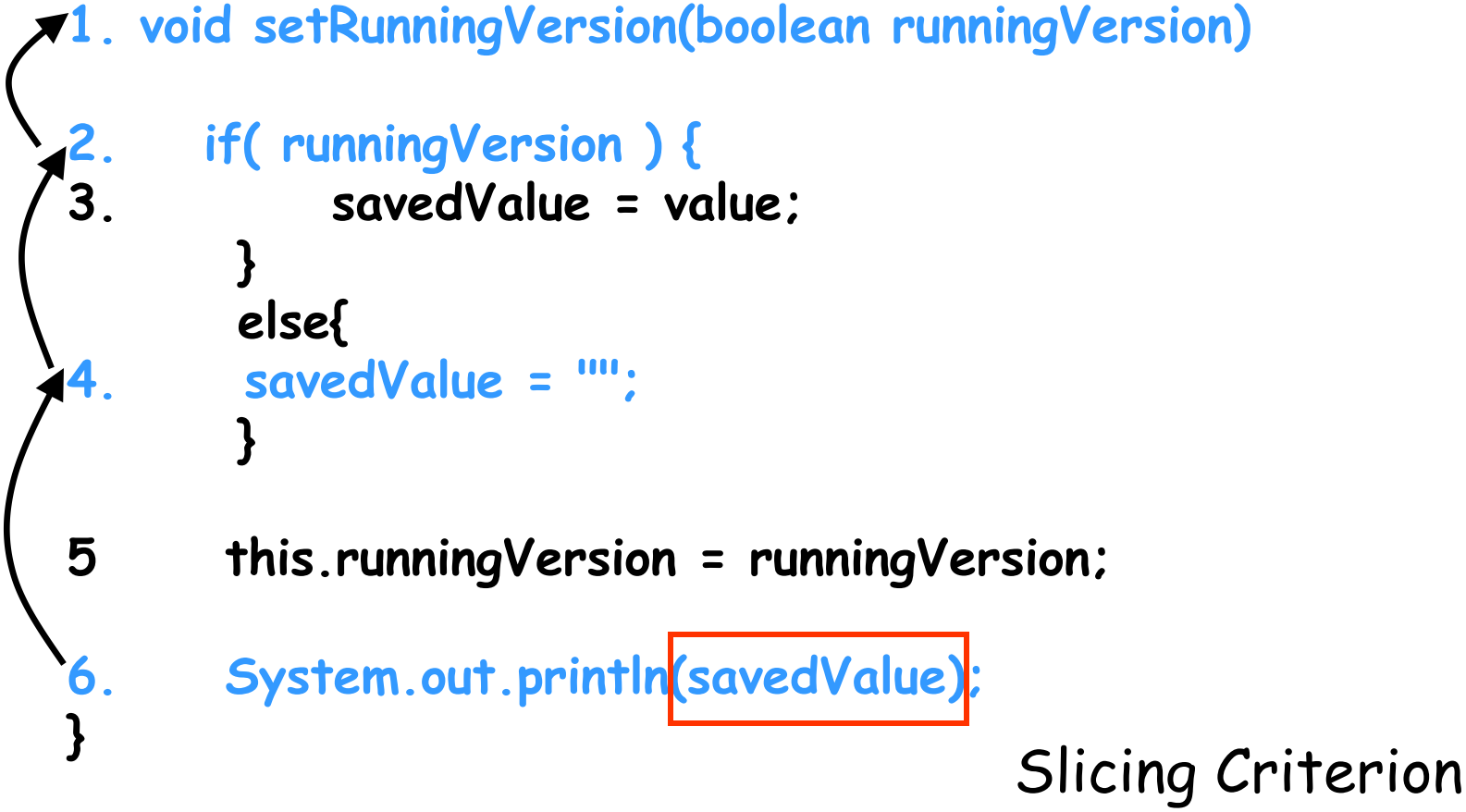
Dynamic Slicing for Debugging



Dynamic Slicing



Dynamic Slice

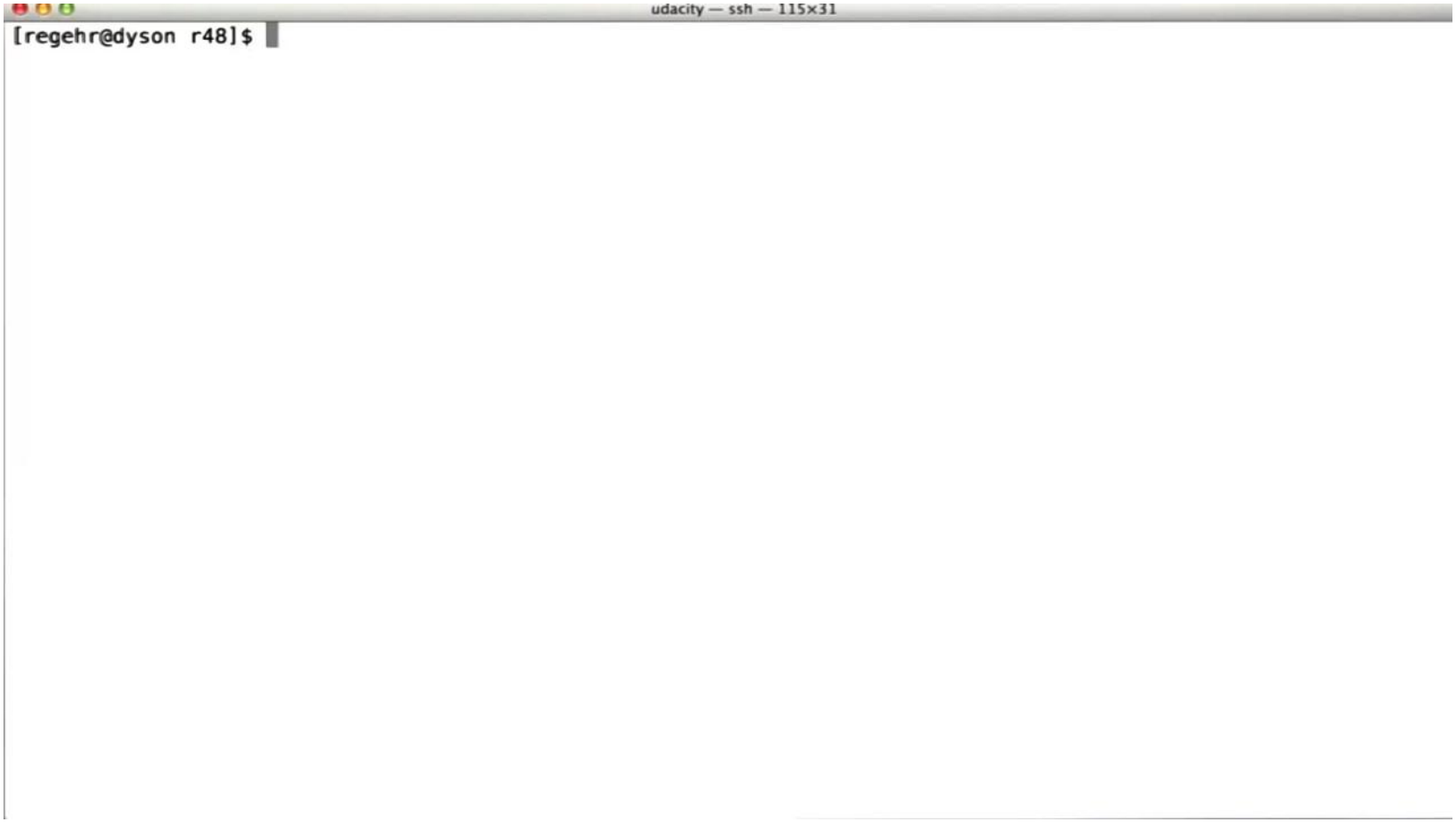


Dynamic slicing for debugging?

- Scalability
  - Large traces to analyze (and store?)
    - Optimizations exist – online compression.
  - Slice is too huge – slice comprehension
    - Tools such as WHYLINE have made it more user friendly
- Slicing still does not tell us what is actually wrong
  - Where did we veer off from the intended behavior? (我们从什么地方开始驶离原来的路径？)
  - What *is* the intended behavior? Often not documented! Lack of specifications is a problem.

Automated Debugging

Delta Debugging


udacity — ssh — 115×31
[regehr@dyson r48]$

Simplification

Once we have reproduced a program failure, we must find out what's relevant:

- Does failure really depend on 10,000 lines of code?
- Does failure really require this exact schedule of events?
- Does failure really need this sequence of function calls?

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

Real-World Scenario

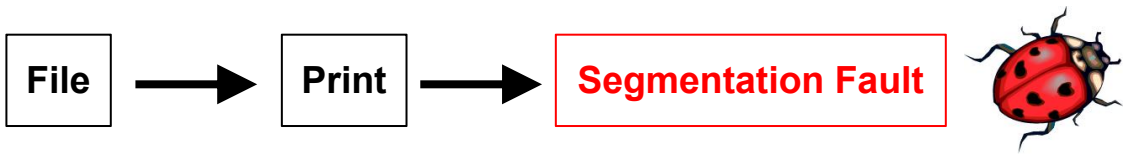
In July 1999, Bugzilla listed more than 370 open bug reports for Mozilla's web browser

- These were not even simplified
- Mozilla engineers were overwhelmed with the work
- They created the Mozilla BugAThon: a call for volunteers to simplify bug reports

> *When you've cut away as much* HTML, CSS, *and JavaScript as you can, and cutting away any more causes the bug to disappear, you're done.*
>
> — Mozilla BugAThon call

How do we go from this …

```
<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 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>
```



… to this?

<SELECT>

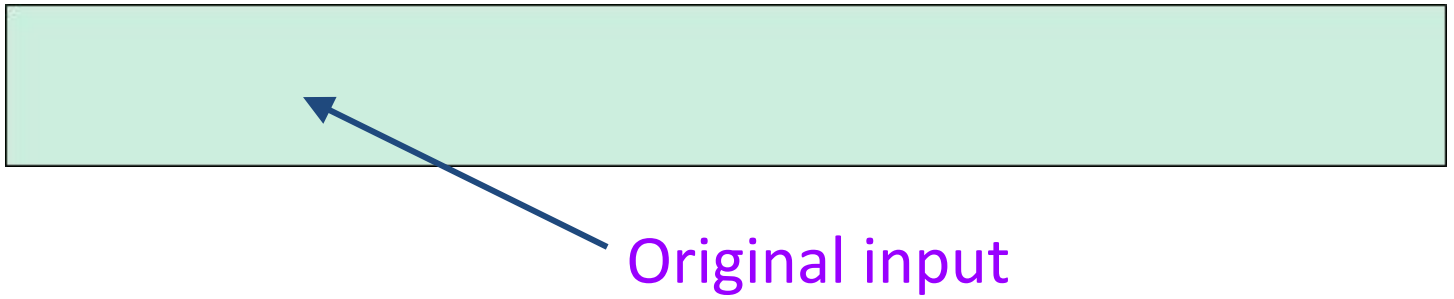
File → Print → Segmentation Fault

Your Solution

- How do you solve these problems?
- Binary Search
  - Cut the test-case in half
  - Iterate
- Brilliant idea: why not automate this?

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.



Binary Search

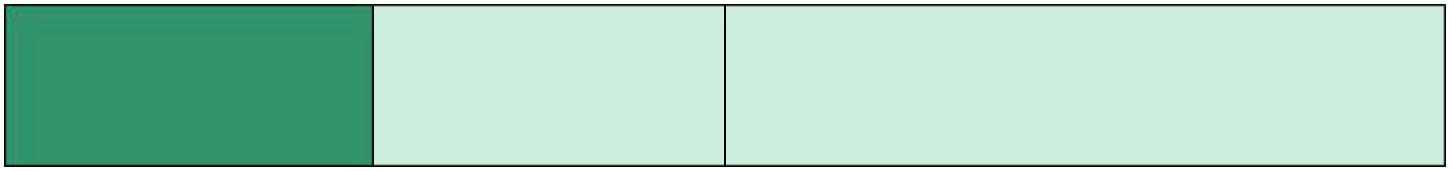
- Proceed by binary search. Throw away half the input and see if the output is still wrong.
- If not, go back to the previous state and discard the other half of the input.

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.

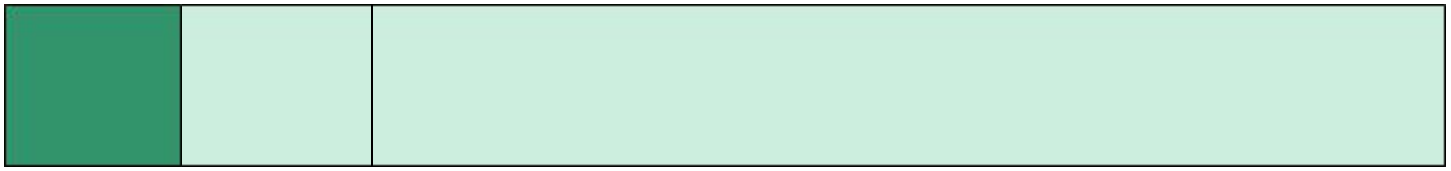
Binary Search

- 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.



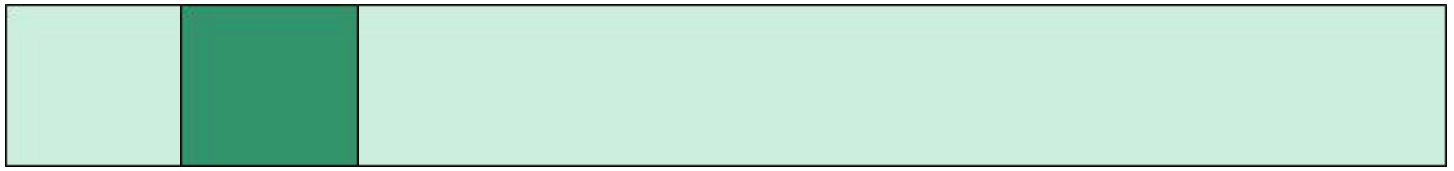
Binary Search

- 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.



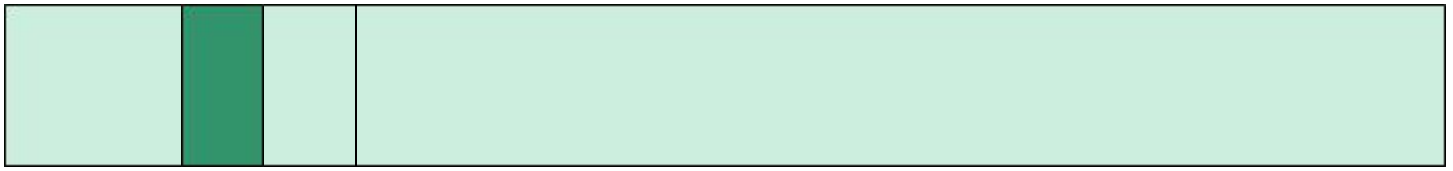
Binary Search

- 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.



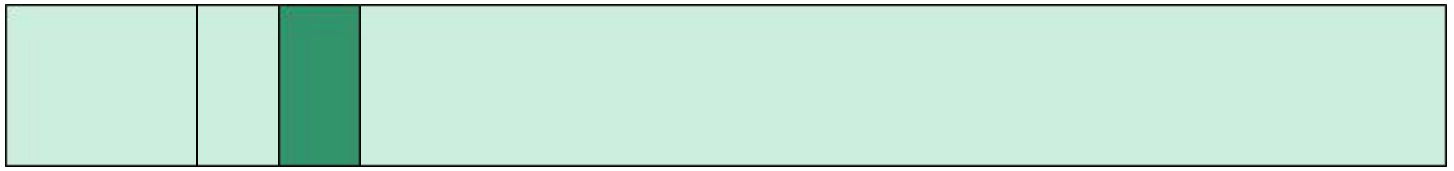
Binary Search

- 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.



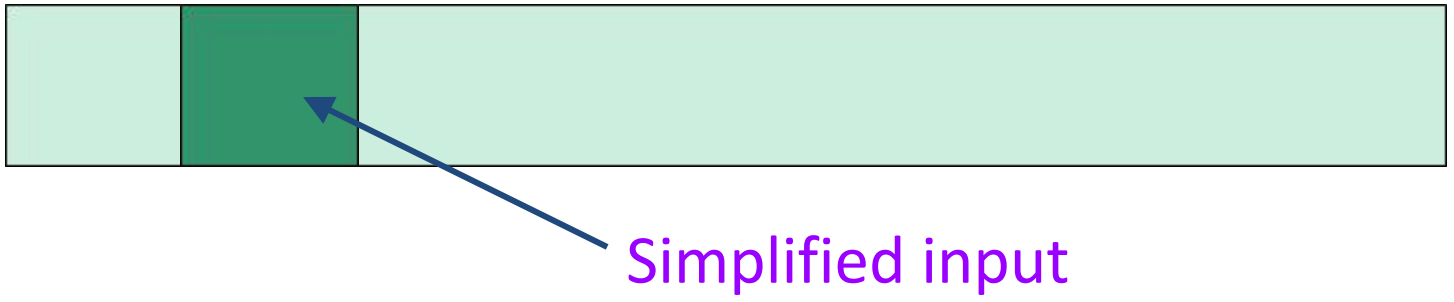
Binary Search

- 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.



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.



Complex Input

```
<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 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>
```