CS409
Software Testing

TAN, Shin Hwei
陈馨慧
Southern University of Science and Technology

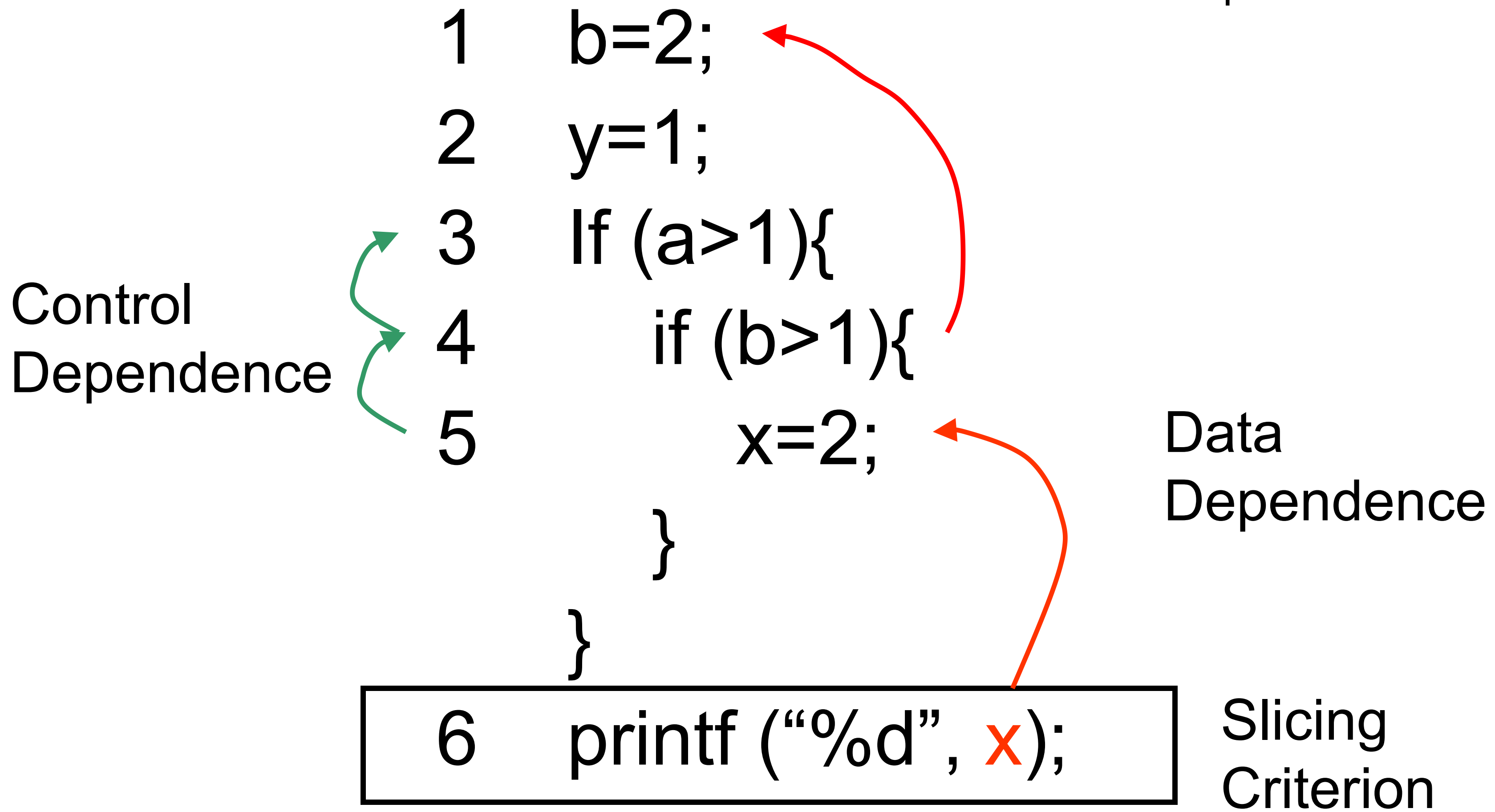
Slides adapted from cs4218 in NUS

Administrative Info

- Project Progress Report uploaded
  - Due on December 4 (this week), 11.59pm
- MP2 score uploaded
  - Check your score (Some of you forget to complete certain part)
- No bug report posted in GitHub discussion so far!
  - Use your app frequently and test it to find more bugs for the bonus!

Recap: Dynamic Slicing

Consider input a == 2



```
1    b=2;
2    y=1;
3    If (a>1){
4        if (b>1){
5            x=2;
         }
     }
6    printf ("%d", x);
```

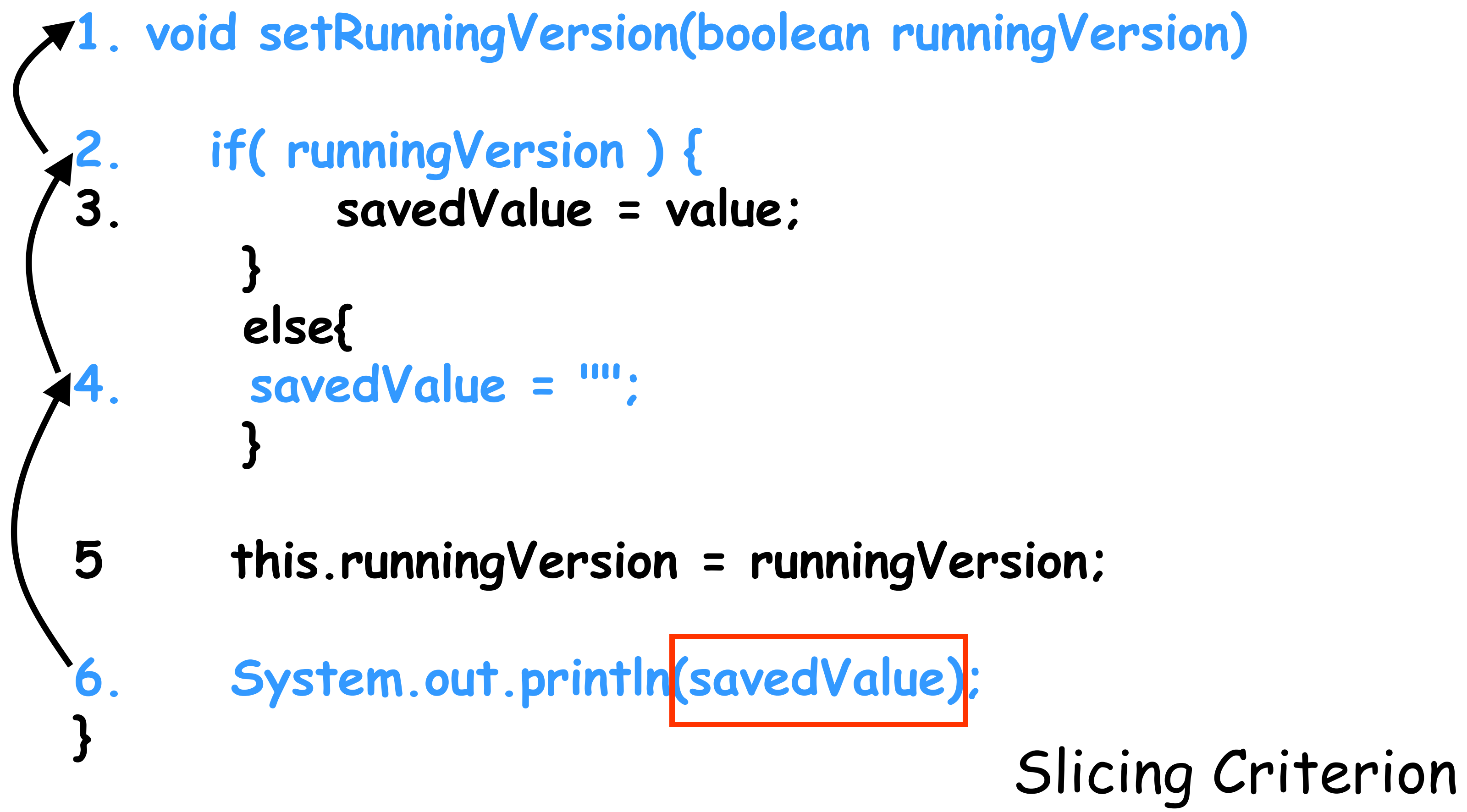
Dynamic Slice

```
1. void setRunningVersion(boolean runningVersion)

2.     if( runningVersion ) {
3.           savedValue = value;
        }
        else{
4.      savedValue = "";
        }

5      this.runningVersion = runningVersion;

6.     System.out.println(savedValue);
}
```

Slicing Criterion

What is the dynamic slice for Slicing criterion= savedValue?

Dynamic Slice



```
1. void setRunningVersion(boolean runningVersion)

2.     if( runningVersion ) {
3.           savedValue = value;
        }
        else{
4.      savedValue = "";
        }

5      this.runningVersion = runningVersion;

6.     System.out.println(savedValue);
}
```

Slicing Criterion

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

```
[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>
```

$$\begin{tikzcd}\boxed{\text{File}} \arrow[r] & \boxed{\text{Print}} \arrow[r] & \boxed{\text{Segmentation Fault}}\end{tikzcd}$$

… to this?

<SELECT>

$$\begin{tikzcd}\boxed{\text{File}} \arrow[r] & \boxed{\text{Print}} \arrow[r] & \boxed{\text{Segmentation Fault}}\end{tikzcd}$$

Your Solution

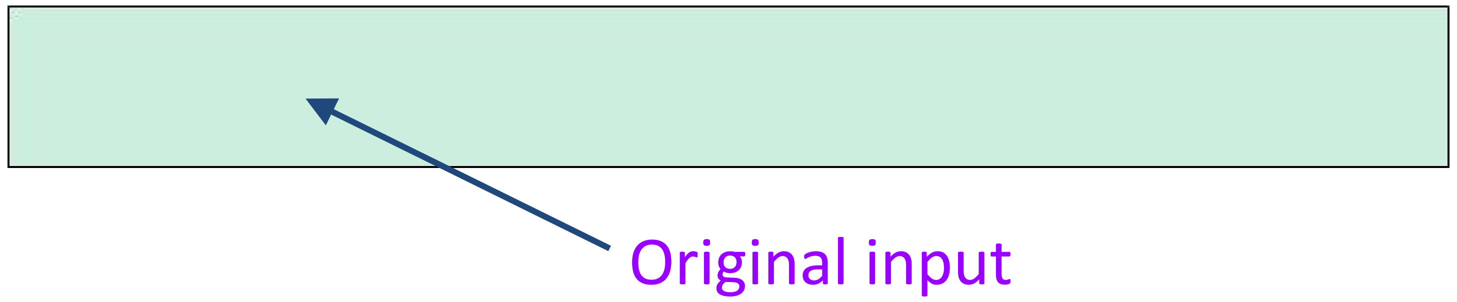
- How do you solve these problems?

- 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.



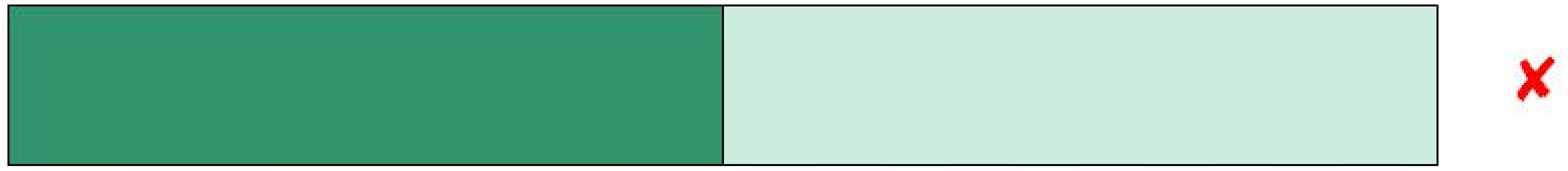
Binary Search

- 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.

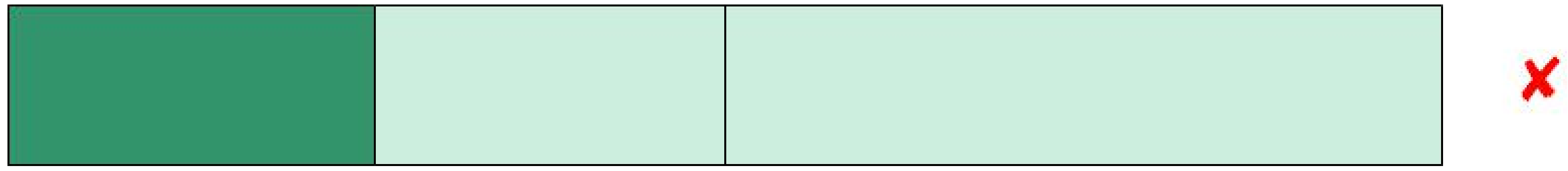
Binary Search

- 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.



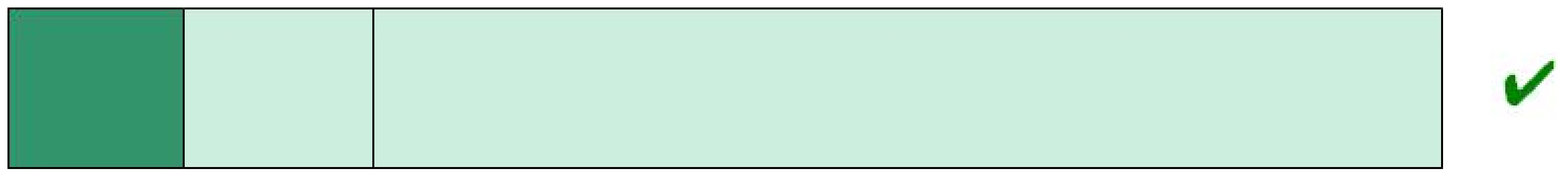
Binary Search

- 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.



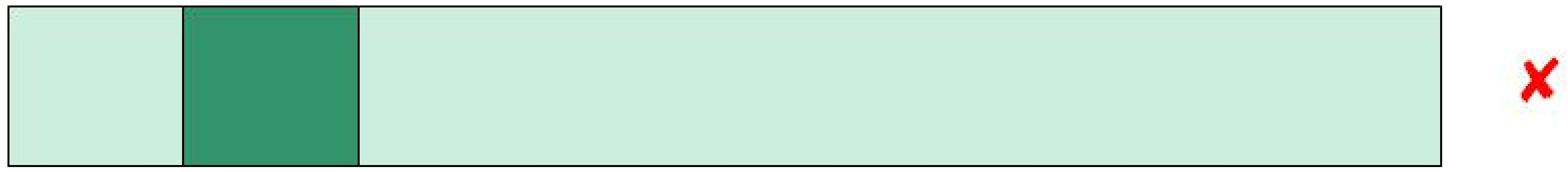
Binary Search

- 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.



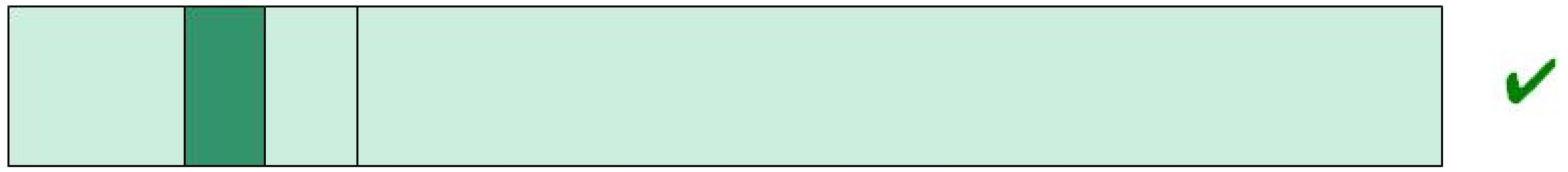
Binary Search

- 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.



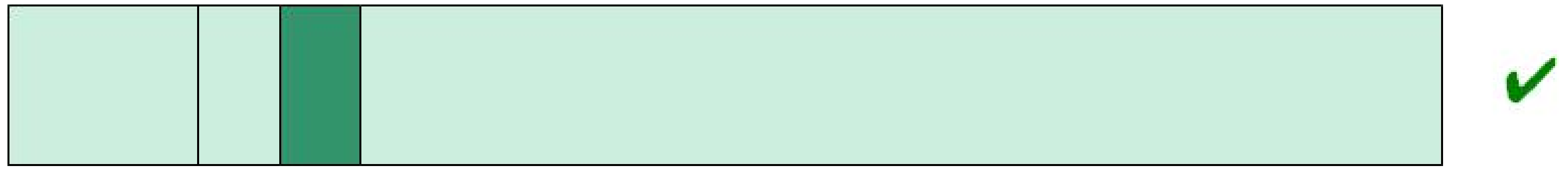
Binary Search

- 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.



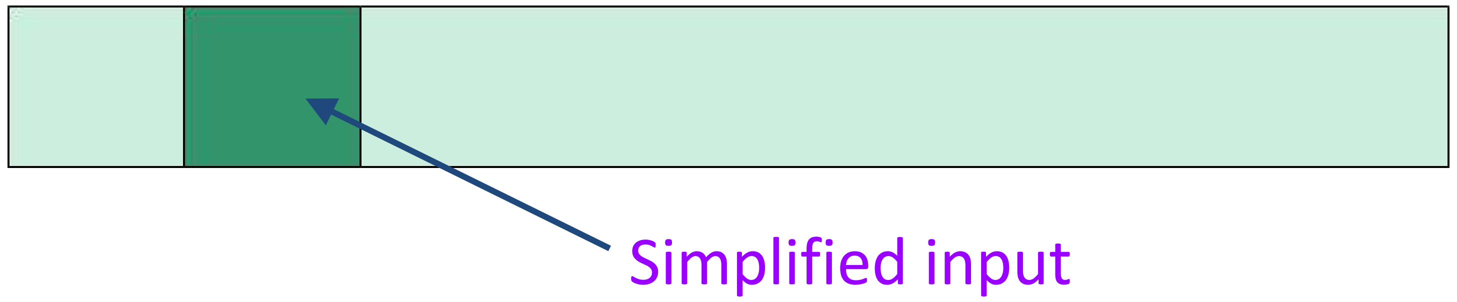
Binary Search

- 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>
```

$$\begin{tikzcd}\boxed{\text{File}} \arrow[r] & \boxed{\text{Print}} \arrow[r] & \boxed{\text{Segmentation Fault}}\end{tikzcd}$$

Recap: Simplification

- Running example: a smaller bug report:

  When Mozilla tries to print the following HTML input it crashes:

  <SELECT NAME="priority" MULTIPLE SIZE=7>
- How do we go about simplifying this input?
  - Manually remove parts of the input and see if it still causes the program to crash
- For the above example assume that we remove characters from the input file

Simplified Input

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

Simplified from 896 lines to one single line
in only 57 tests!

Binary Search

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

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

Binary Search

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

✗

Binary Search

```
<SELECT NAME="priority" MULTIPLE SIZE=7>          ✗
<SELECT NAME="priority" MULTIPLE SIZE=7>          ✓
```

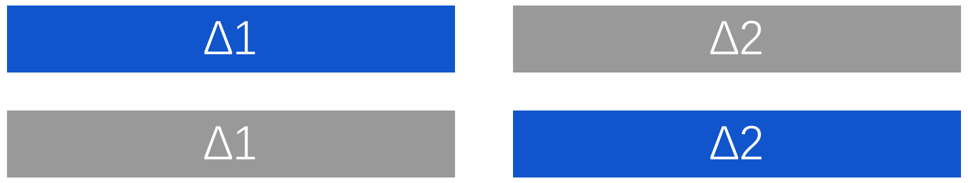
Binary Search

```
<SELECT NAME="priority" MULTIPLE SIZE=7>    ✗
<SELECT NAME="priority" MULTIPLE SIZE=7>    ✓
<SELECT NAME="priority" MULTIPLE SIZE=7>    ✓
```

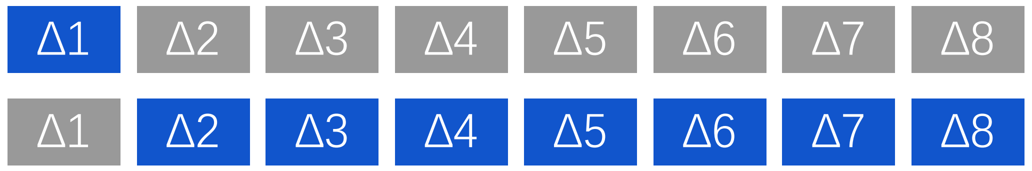
What do we do if both halves pass?

Two Conflicting Solutions

- Few and large changes:



- More and smaller changes:



… (many more)

QUIZ: Impact of Input Granularity

Input granularity:	Finer	Coarser
Chance of finding a failing input subset		
Progress of the search		

A. Slower B. Higher C. Faster D. Lower

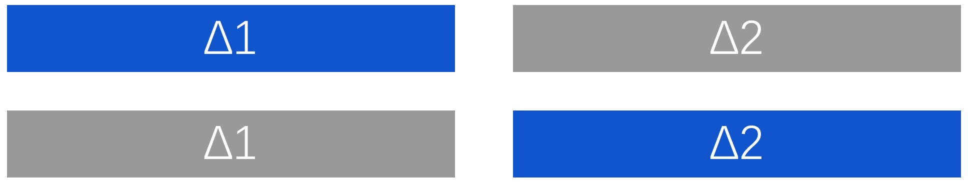
QUIZ: Impact of Input Granularity

Input granularity:	Finer	Coarser
Chance of finding a failing input subset	B. Higher	D. Lower
Progress of the search	A. Slower	C. Faster

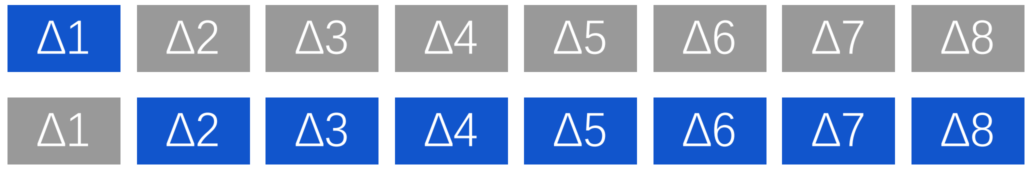
A. Slower B. Higher C. Faster D. Lower

General Delta-Debugging Algorithm

- Few and large changes: start first with these two



- More and smaller changes: apply if both above pass



… (many more)

Example: Delta Debugging

```
<SELECT NAME="priority" MULTIPLE SIZE=7>    ✗
<SELECT NAME="priority" MULTIPLE SIZE=7>    ✓
<SELECT NAME="priority" MULTIPLE SIZE=7>    ✓
```

Example: Delta Debugging

```
<SELECT NAME="priority" MULTIPLE SIZE=7>    ✗
<SELECT NAME="priority" MULTIPLE SIZE=7>    ✓
<SELECT NAME="priority" MULTIPLE SIZE=7>    ✓
<SELECT NAME="priority" MULTIPLE SIZE=7>    ✓
```

Example: Delta Debugging

```
<SELECT NAME="priority" MULTIPLE SIZE=7>          ✗
<SELECT NAME="priority" MULTIPLE SIZE=7>          ✓
<SELECT NAME="priority" MULTIPLE SIZE=7>          ✓
<SELECT NAME="priority" MULTIPLE SIZE=7>          ✓
<SELECT NAME="priority" MULTIPLE SIZE=7>          ✗
```

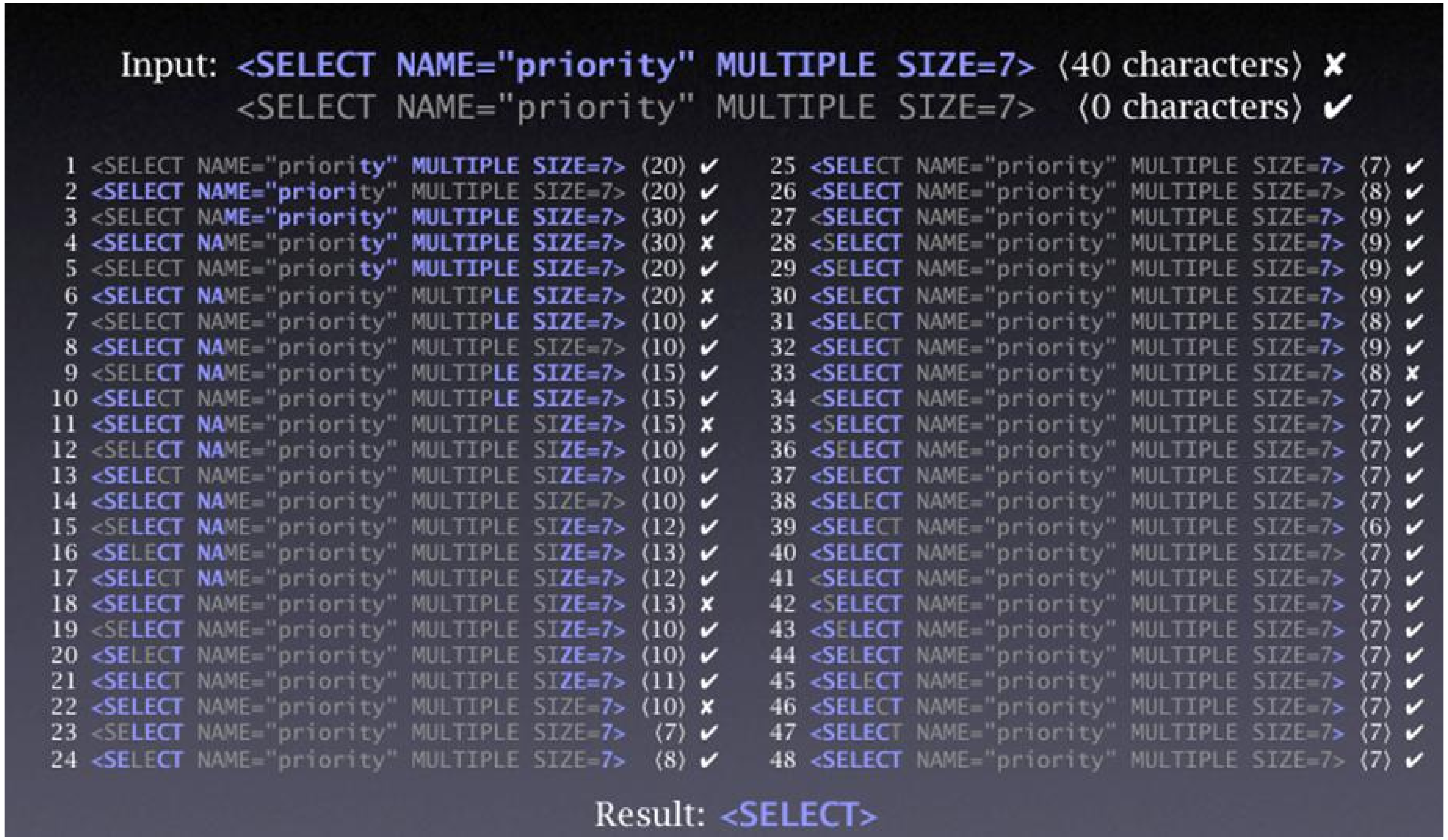
Example: Delta Debugging

```
<SELECT NAME="priority" MULTIPLE SIZE=7>    ✗
<SELECT NAME="priority" MULTIPLE SIZE=7>    ✓
<SELECT NAME="priority" MULTIPLE SIZE=7>    ✓
<SELECT NAME="priority" MULTIPLE SIZE=7>    ✓
<SELECT NAME="priority" MULTIPLE SIZE=7>    ✗
<SELECT NAME="priority" MULTIPLE SIZE=7>    ✗
```

Example: Delta Debugging

```
<SELECT NAME="priority" MULTIPLE SIZE=7>    ✗
<SELECT NAME="priority" MULTIPLE SIZE=7>    ✓
<SELECT NAME="priority" MULTIPLE SIZE=7>    ✓
<SELECT NAME="priority" MULTIPLE SIZE=7>    ✓
<SELECT NAME="priority" MULTIPLE SIZE=7>    ✗
<SELECT NAME="priority" MULTIPLE SIZE=7>    ✗
<SELECT NAME="priority" MULTIPLE SIZE=7>    ✓
<SELECT NAME="priority" MULTIPLE SIZE=7>
```

Continuing Delta Debugging



```
Input: <SELECT NAME="priority" MULTIPLE SIZE=7> ⟨40 characters⟩ ✘
       <SELECT NAME="priority" MULTIPLE SIZE=7>  ⟨0 characters⟩ ✔

 1 <SELECT NAME="priority" MULTIPLE SIZE=7> ⟨20⟩ ✔    25 <SELECT NAME="priority" MULTIPLE SIZE=7> ⟨7⟩ ✔
 2 <SELECT NAME="priority" MULTIPLE SIZE=7> ⟨20⟩ ✔    26 <SELECT NAME="priority" MULTIPLE SIZE=7> ⟨8⟩ ✔
 3 <SELECT NAME="priority" MULTIPLE SIZE=7> ⟨30⟩ ✔    27 <SELECT NAME="priority" MULTIPLE SIZE=7> ⟨9⟩ ✔
 4 <SELECT NAME="priority" MULTIPLE SIZE=7> ⟨30⟩ ✘    28 <SELECT NAME="priority" MULTIPLE SIZE=7> ⟨9⟩ ✔
 5 <SELECT NAME="priority" MULTIPLE SIZE=7> ⟨20⟩ ✔    29 <SELECT NAME="priority" MULTIPLE SIZE=7> ⟨9⟩ ✔
 6 <SELECT NAME="priority" MULTIPLE SIZE=7> ⟨20⟩ ✘    30 <SELECT NAME="priority" MULTIPLE SIZE=7> ⟨9⟩ ✔
 7 <SELECT NAME="priority" MULTIPLE SIZE=7> ⟨10⟩ ✔    31 <SELECT NAME="priority" MULTIPLE SIZE=7> ⟨8⟩ ✔
 8 <SELECT NAME="priority" MULTIPLE SIZE=7> ⟨10⟩ ✔    32 <SELECT NAME="priority" MULTIPLE SIZE=7> ⟨9⟩ ✔
 9 <SELECT NAME="priority" MULTIPLE SIZE=7> ⟨15⟩ ✔    33 <SELECT NAME="priority" MULTIPLE SIZE=7> ⟨8⟩ ✘
10 <SELECT NAME="priority" MULTIPLE SIZE=7> ⟨15⟩ ✔    34 <SELECT NAME="priority" MULTIPLE SIZE=7> ⟨7⟩ ✔
11 <SELECT NAME="priority" MULTIPLE SIZE=7> ⟨15⟩ ✘    35 <SELECT NAME="priority" MULTIPLE SIZE=7> ⟨7⟩ ✔
12 <SELECT NAME="priority" MULTIPLE SIZE=7> ⟨10⟩ ✔    36 <SELECT NAME="priority" MULTIPLE SIZE=7> ⟨7⟩ ✔
13 <SELECT NAME="priority" MULTIPLE SIZE=7> ⟨10⟩ ✔    37 <SELECT NAME="priority" MULTIPLE SIZE=7> ⟨7⟩ ✔
14 <SELECT NAME="priority" MULTIPLE SIZE=7> ⟨10⟩ ✔    38 <SELECT NAME="priority" MULTIPLE SIZE=7> ⟨7⟩ ✔
15 <SELECT NAME="priority" MULTIPLE SIZE=7> ⟨12⟩ ✔    39 <SELECT NAME="priority" MULTIPLE SIZE=7> ⟨6⟩ ✔
16 <SELECT NAME="priority" MULTIPLE SIZE=7> ⟨13⟩ ✔    40 <SELECT NAME="priority" MULTIPLE SIZE=7> ⟨7⟩ ✔
17 <SELECT NAME="priority" MULTIPLE SIZE=7> ⟨12⟩ ✔    41 <SELECT NAME="priority" MULTIPLE SIZE=7> ⟨7⟩ ✔
18 <SELECT NAME="priority" MULTIPLE SIZE=7> ⟨13⟩ ✘    42 <SELECT NAME="priority" MULTIPLE SIZE=7> ⟨7⟩ ✔
19 <SELECT NAME="priority" MULTIPLE SIZE=7> ⟨10⟩ ✔    43 <SELECT NAME="priority" MULTIPLE SIZE=7> ⟨7⟩ ✔
20 <SELECT NAME="priority" MULTIPLE SIZE=7> ⟨10⟩ ✔    44 <SELECT NAME="priority" MULTIPLE SIZE=7> ⟨7⟩ ✔
21 <SELECT NAME="priority" MULTIPLE SIZE=7> ⟨11⟩ ✔    45 <SELECT NAME="priority" MULTIPLE SIZE=7> ⟨7⟩ ✔
22 <SELECT NAME="priority" MULTIPLE SIZE=7> ⟨10⟩ ✘    46 <SELECT NAME="priority" MULTIPLE SIZE=7> ⟨7⟩ ✔
23 <SELECT NAME="priority" MULTIPLE SIZE=7>  ⟨7⟩ ✔    47 <SELECT NAME="priority" MULTIPLE SIZE=7> ⟨7⟩ ✔
24 <SELECT NAME="priority" MULTIPLE SIZE=7>  ⟨8⟩ ✔    48 <SELECT NAME="priority" MULTIPLE SIZE=7> ⟨7⟩ ✔

Result: <SELECT>
```

Inputs and Failures

- Let R be the set of possible inputs
- $r_P \in R$ corresponds to an input that passes
- $r_F \in R$ corresponds to an input that fails

Example: Delta Debugging

```
<SELECT NAME="priority" MULTIPLE SIZE=7>     ✗
<SELECT NAME="priority" MULTIPLE SIZE=7>     ✓
<SELECT NAME="priority" MULTIPLE SIZE=7>     ✓
<SELECT NAME="priority" MULTIPLE SIZE=7>     ✓
<SELECT NAME="priority" MULTIPLE SIZE=7>     ✗
<SELECT NAME="priority" MULTIPLE SIZE=7>     ✗
<SELECT NAME="priority" MULTIPLE SIZE=7>     ✓
<SELECT NAME="priority" MULTIPLE SIZE=7>
```

Changes

- Let R denote the set of all possible inputs
- We can go from one input r1 to another input r2 by a series of changes
- A change $\boldsymbol{\delta}$ is a mapping R $\longrightarrow$R which takes one input and changes it to another input

Changes

Example: **δ** ’ = insert ME="priori at input position 10

```
r1 = <SELECT NAty" MULTIPLE SIZE=7>
δ ’(r1) = <SELECT NAME="priority" MULTIPLE SIZE=7>
```

Decomposing Changes

- A change $\boldsymbol{\delta}$ can be decomposed to a number of elementary changes $\boldsymbol{\delta}_1, \boldsymbol{\delta}_2, \cdots, \boldsymbol{\delta}_n$ where $\boldsymbol{\delta} = \boldsymbol{\delta}_1 \circ \boldsymbol{\delta}_2 \circ \cdots \circ \boldsymbol{\delta}_n$ and $(\boldsymbol{\delta}_i \circ \boldsymbol{\delta}_j)(r) = \boldsymbol{\delta}_j(\boldsymbol{\delta}_i(r))$
- For example, deleting a part of the input file can be decomposed to deleting characters one by one from the file
- In other words: by composing the deletion of single characters, we can get a change that deletes part of the input file

Decomposing Changes

Example: $\boldsymbol{\delta}'$ = insert ME="priori at input position 10

can be decomposed as $\boldsymbol{\delta}' = \boldsymbol{\delta}_1 \circ \boldsymbol{\delta}_2 \circ \cdots \circ \boldsymbol{\delta}_{10}$

where $\boldsymbol{\delta}_1$ = insert M at position 10

$\boldsymbol{\delta}_2$ = insert E at position 11

. . .

Summary

- We have an input without failure: r_P
- We have an input with failure: r_F
- We have a set of changes $c_F = \{ \boldsymbol{\delta}_1, \boldsymbol{\delta}_2, \cdots, \boldsymbol{\delta}_n \}$ such that:

$$r_F = (\boldsymbol{\delta}_1 \circ \boldsymbol{\delta}_2 \circ \cdots \circ \boldsymbol{\delta}_n)(r_P)$$

- Each subset c of c_F is a test case

Testing Test Cases

- Given a test case c, we would like to know if the input generated by applying changes in c to r_P causes the same failure as r_F

- We define the function $test: Powerset(c_F) \longrightarrow \{P, F, ?\}$ such that, given $c = \{\boldsymbol{\delta}_1, \boldsymbol{\delta}_2, \cdots, \boldsymbol{\delta}_n\} \subseteq cF$ $test(c) = F$ iff $(\boldsymbol{\delta}_1 \circ \boldsymbol{\delta}_2 \circ \cdots \circ \boldsymbol{\delta}_n)(r_P)$ is a failing input

Minimizing Test Cases

- Goal: find the smallest test case **c** such that **test(c) = F**
- A failing test case $\mathbf{c} \subseteq \mathbf{c_F}$ is called the global minimum of $\mathbf{c_F}$ if:

  for all $\mathbf{c' \subseteq c_F}$, $\mathbf{|c'| < |c| \Rightarrow test(c') \neq F}$
- The global minimum is the smallest set of changes which will make the program fail
- Finding the global minimum may require performing an exponential number of tests

Search for 1-minimal Input

- Different problem formulation:

  Find a set of changes that cause the failure, but removing any change causes the failure to go away

- This is 1-minimality

Minimizing Test Cases

- A failing test case $c \subseteq c_F$ is called a **local minimum** of $\mathbf{c_F}$ if:

  for all $\mathbf{c' \subset c}$ **, test(c') ≠ F**

- A failing test case $\mathbf{c \subseteq c_F}$ is **n-minimal** if:

  for all $\mathbf{c' \subset c}$, $\mathbf{|c| - |c'| \leq n \Rightarrow}$ **test(c') ≠ F**

- A failing test case is 1-minimal if:

  for all $\boldsymbol{\delta}_{\mathbf{i}} \in \mathbf{c}$ **, test(c - {**$\boldsymbol{\delta}_{\mathbf{i}}$**}) ≠ F**

QUIZ: Minimizing Test Cases

A program takes a string of **a**’s and **b**’s as input. It crashes on inputs with an odd number of b’s AND an even number of **a**’s. Write a crashing test case (or **NONE** if none exists) that is a sub-sequence of input **babab** and is:

- Smallest:
- Local minimum but not smallest:
- 1-minimal, of size 3:
- 2-minimal, of size 3:

QUIZ: Minimizing Test Cases

A program takes a string of **a**’s and **b**’s as input. It crashes on inputs with an odd number of b’s AND an even number of **a**’s. Write a crashing test case (or **NONE** if none exists) that is a sub-sequence of input **babab** and is:

- Smallest: b
- Local minimum but not smallest: NONE
- 1-minimal, of size 3: aab, aba, baa, bbb
- 2-minimal, of size 3: NONE

Naive Algorithm

- To find a 1-minimal subset of **c** :

```
if for all $\delta_i \in c$, test(c - {$\delta_i$}) $\neq$ F, then c is 1-minimal
else recurse on c - {$\delta$ } for some $\delta \in c$, test(c - {$\delta$ }) = F
```

Running-Time Analysis

- In the worst case,
  - We remove one element from the set per iteration
  - After trying every other element

- Work is potentially $N + (N-1) + (N-2) + \cdots$

- This is $O(N^2)$

Work Smarter, Not Harder

- We can often do better

- It is silly to start removing one element at a time
  - Try dividing the change set in two initially
  - Increase the number of subsets if we can’t make progress
  - If we get lucky, search will converge quickly

Minimization Algorithm

- The delta debugging algorithm finds a 1-minimal test case
- It partitions the set $\mathbf{c_F}$ to $\mathbf{\Delta_1, \Delta_2, \cdots, \Delta_n}$
  - $\mathbf{\Delta_1, \Delta_2, \cdots, \Delta_n}$ are pairwise disjoint, and $\mathbf{c_F = \Delta_1} \cup \mathbf{\Delta_2} \cup \cdots \cup \mathbf{\Delta_n}$
- Define the complement of $\mathbf{\Delta_i}$ as $\mathbf{\nabla_i = c_F - \Delta_i}$
- Start with $\mathbf{n = 2}$
- Tests each test case defined by each partition and its complement
- Reduces the test case if a smaller failure inducing set is found, otherwise it refines the partition (i.e. $\mathbf{n = n * 2}$)

Steps of the Minimization Algorithm

1. Start with $\mathbf{n = 2}$ and $\mathbf{\Delta}$ as test set
2. Test each $\mathbf{\Delta_1, \Delta_2, \cdots, \Delta_n}$ and each $\mathbf{\nabla_1, \nabla_2, \cdots, \nabla_n}$
3. There are three possible outcomes:
   a. Some $\mathbf{\Delta_i}$ causes failure:
      Go to step (1) with $\mathbf{\Delta = \Delta_i}$ and $\mathbf{n = 2}$
   b. Some $\mathbf{\nabla_i}$ causes failure:
      Go to step (1) with $\mathbf{\Delta = \nabla_i}$ and $\mathbf{n = n - 1}$
   c. No test causes failure:
      If granularity can be refined: Go to step (1) with $\mathbf{\Delta = \Delta}$ and $\mathbf{n = n * 2}$

      Otherwise: Done, found the 1-minimal subset

Asymptotic Analysis

- Worst case is still quadratic
- Subdivide until each set is of size 1
  - reduced to the naive algorithm
- Good news:
  - For single failure, converges in log N
  - Binary search again

QUIZ: Minimization Algorithm

A program crashes when its input contains 42. Fill in the data in each iteration of the minimization algorithm assuming character granularity.

Iteration	n	**Δ**	**$\Delta_1, \Delta_2, \ldots, \Delta_n, \nabla_1, \nabla_2, \ldots, \nabla_n$**
1		2424	
2			
3			
4			

QUIZ: Minimization Algorithm

A program crashes when its input contains 42. Fill in the data in each iteration of the minimization algorithm assuming character granularity.

Iteration	n	**Δ**	**$\Delta_1, \Delta_2, \ldots, \Delta_n$, $\nabla_1, \nabla_2, \ldots, \nabla_n$**
1	2	2424	24
2	4	2424	2, 4, 242, 224, 424, 244
3	3	242	2, 4, 24, 42, 22
4	2	42	4, 2

Case Study: GNU C Compiler

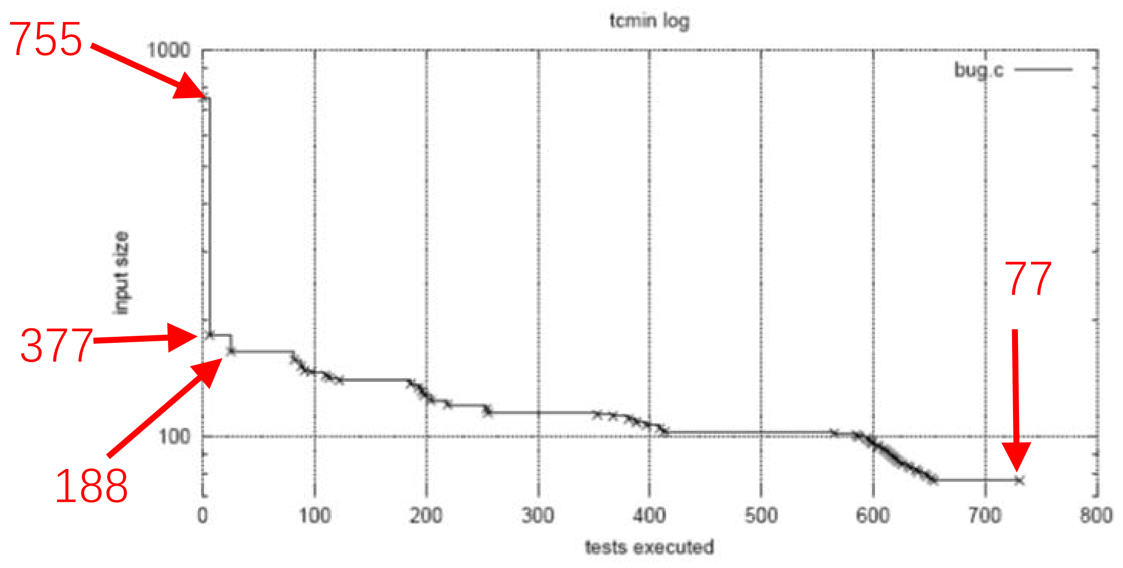
```
#define SIZE 20
double mult(double z[], int n) {
   int i, j;
   i = 0;
   for (j = 0; j < n; j++) {
                i = i + j + 1;
                z[i] = z[i] * (z[0] + 1.0);
   }
   return z[n];
}
void copy(double to[], double from[], int count) {
    int n = (count + 7) / 8;
    switch (count % 8) do {
                case 0: *to++ = *from++;
    case 7: *to++ = *from++;
    case 6: *to++ = *from++;
    case 5: *to++ = *from++;
    case 4: *to++ = *from++;
    case 3: *to++ = *from++;
    case 2: *to++ = *from++;
    case 1: *to++ = *from++;
  } while (--n > 0);
  return mult(to, 2);
}
int main(int argc, char *argv[]) {
   double x[SIZE], y[SIZE];
   double *px = x;
   while (px < x + SIZE)
                *px++ = (px – x) * (SIZE + 1.0);
   return copy(y, x, SIZE)
}
```

- This program (bug.c) crashes GCC 2.95.2 when optimization is enabled
- Goal: minimize this program to file a bug report
- For GCC, a passing run is the empty input
- For simplicity, model each change as insertion of a single character
  - test $\mathbf{r_P}$ = running GCC on an empty input
  - test $\mathbf{r_F}$ = running GCC on bug.c
  - change $\boldsymbol{\delta}_i$ = insert ith character of bug.c

Case Study: GNU C Compiler

The test procedure:

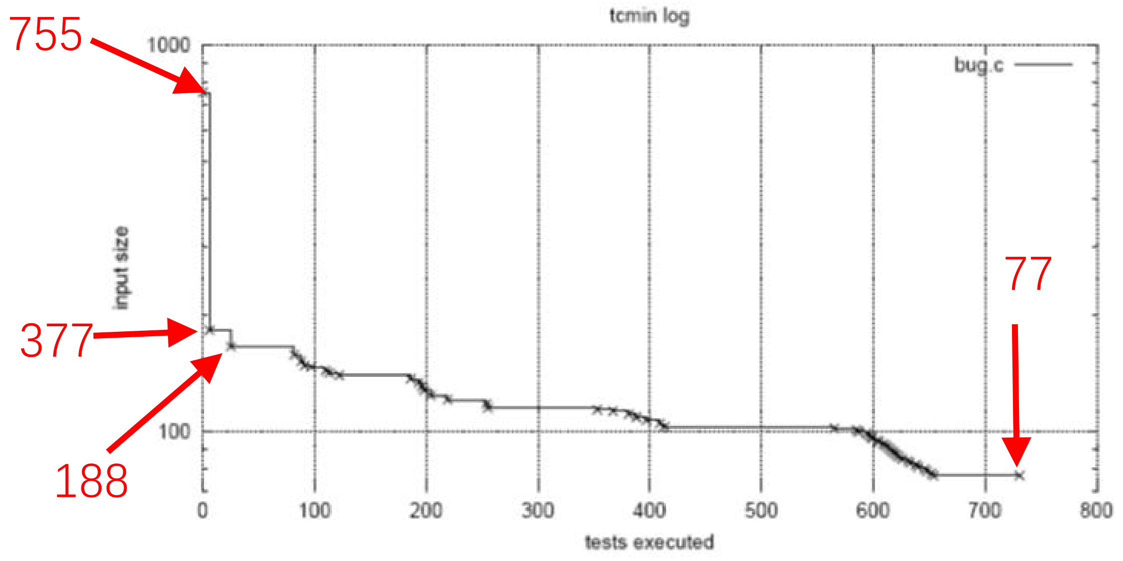
- create the appropriate subset of bug.c
- feed it to GCC
- return **Failed** if GCC crashes, **Passed** otherwise



Case Study: GNU C Compiler

```
t(double z[],int n){int i,j;for(;;){i=i+j+1;z[i]=z[i]*(z[0]+0);}return z[n];}
```

```
double mult(double z[], int n) {
   int i, j;
   i = 0;
   for (j = 0; j < n; j++) {
          i = i + j + 1;
          z[i] = z[i] * (z[0] +
1.0);
   }
   return z[n];
}
```



Case Study: GNU C Compiler

```
t(double z[],int n){int i,j;for(;;){i=i+j+1;z[i]=z[i]*(z[0]+0);}return z[n];}
```

```
double mult(double z[], int n) {
   int i, j;
   i = 0;
   for (j = 0; j < n; j++) {
          i = i + j + 1;
          z[i] = z[i] * (z[0] +
1.0);
   }
   return z[n];
}
```

- This test case is 1-minimal
  - No single character can be removed while still causing the crash
  - Even every superfluous whitespace has been removed
  - The function name has shrunk from **mult** to a single **t**
  - Has infinite loop, but GCC still isn't supposed to crash

- So where could the bug be?
  - We already know it is related to optimization
  - Crash disappears if we remove **-O** option to turn off optimization

Case Study: GNU C Compiler

- The GCC documentation lists 31 options to control optimization:

```
-ffloat-store               -fno-default-inline           -fno-defer-pop
-fforce-mem                 -fforce-addr                  -fomit-frame-pointer
-fno-inline                 -finline-functions            -fkeep-inline-functions
-fkeep-static-consts        -fno-function-cse             -ffast-math
-fstrength-reduce           -fthread-jumps                -fcse-follow-jumps
-fcse-skip-blocks           -frerun-cse-after-loop        -frerun-loop-opt
-fgcse                      -fexpensive-optimizations     -fschedule-insns
-fschedule-insns2           -ffunction-sections           -fdata-sections
-fcaller-saves              -funroll-loops                -funroll-all-loops
-fmove-all-movables         -freduce-all-givs             -fno-peephole
-fstrict-aliasing
```

- Applying all of these options causes the crash to disappear
  - Some option(s) prevent the crash

Case Study: GNU C Compiler

- Use test cases minimization to find the crash-preventing option(s)
  - test $\mathbf{r_P}$ = run GCC with all options
  - test $\mathbf{r_F}$ = run GCC with no option
  - change $\boldsymbol{\delta}$ i = remove i^th option
- After 7 tests, option **-ffast-math** is found to prevent the crash
  - Not good candidate for workaround as it may alter program's semantics
  - Thus, remove **-ffast-math** from the list of options and repeat
  - After 7 tests, option **-fforce-addr** is also found to prevent the crash
  - Further tests show that no other option prevents the crash

Case Study: GNU C Compiler

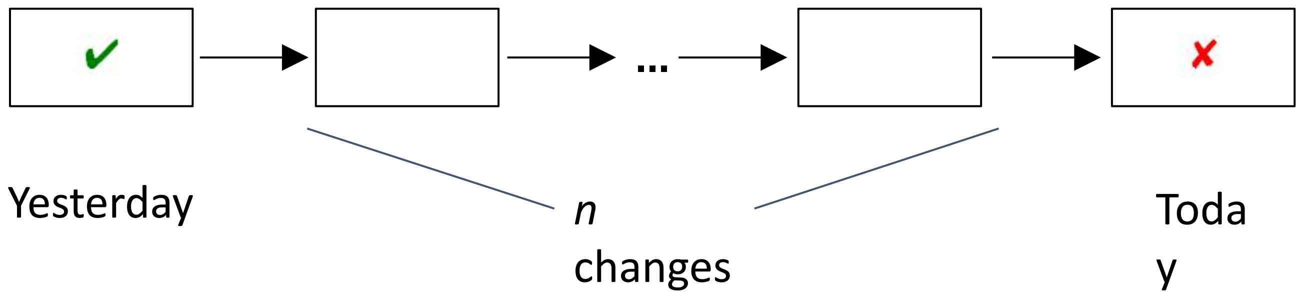
This is what we can send to the GCC maintainers:

- The minimal test case
- “The crash only occurs with optimization”
- “**-ffast-math** and **-fforce-addr** prevent the crash”

Case Study: Minimizing Fuzz Input

- Random Testing (a.k.a. Fuzzing): feed program with randomly generated input and check if it crashes
- Typically generates large inputs that cause program failure
- Use delta debugging to minimize such inputs
- Successfully applied to subset of UNIX utility programs from Bart Miller’s original fuzzing experiment
  - Example: reduced 10^6 character input crashing CRTPLOT to single character in only 24 tests!

Another Application



- Yesterday, my program worked. Today, it does not. Why?
  - The new release 4.17 of GDB changed 178,000 lines
  - No longer integrated properly with DDD (a graphical front-end)
  - How do we isolate the change that caused the failure?

QUIZ: Delta Debugging

Check the statements that are true about delta debugging:

- ○ Is fully automatic.
- ○ Finds the smallest failing subset of a failing input in polynomial time.
- ○ Finds 1-minimal instead of local minimum test case due to performance.
- ○ May find a different sized subset of a failing input depending upon the order in which it tests different input partitions.
- ○ Is also effective at reducing non-deterministically failing inputs.

QUIZ: Delta Debugging

Check the statements that are true about delta debugging:

- ○ Is fully automatic.
- ○ Finds the smallest failing subset of a failing input in polynomial time.
- [✓ ● Finds 1-minimal instead of local minimum test case due to performance.]
- [✓ ● May find a different sized subset of a failing input depending upon the order in which it tests different input partitions.]
- ○ Is also effective at reducing non-deterministically failing inputs.

What Have We Learned?

- Delta Debugging is a technique, not a tool
- Bad news:
  - Probably must be re-implemented for each significant system to exploit knowledge changes
- Good news:
  - Relatively simple algorithm, big payoff
  - It is worth re-implementing

Symbolic Techniques for Debugging and Testing

What is symbolic execution?

Testing/ Concrete Execution

- Each test only explores one possible execution
  - `assert(f(3) == 5)`

Symbolic execution

- *Generalizes* test cases
  - Allows *unknown* symbolic variables in evaluation
    - `y = α;`
    - `assert(f(y) == 2*y-1);`
  - If execution path depends on *unknown*, conceptually *fork* symbolic executor

    -
      ```
      int f(int x) {
        if (x > 0) then return 2*x - 1;
        else return 10;
      }
      ```

Concrete Execution Versus Symbolic Execution

```
int foo(int i){
        int j = 2*i;
        i = i++;
        i = i * j;
        if ( i < 1 )
                i =   -i ;
        return i;
}
```

```
i = 1
i = 1, j = 2
i = 2, j = 2
i = 4, j = 2



return 4
```

Concrete Execution Versus Symbolic Execution

```
int foo(int i){
        int j = 2*i;
        i = i++;
        i = i * j;
        if ( i < 1 )
                i =   -i;
        return i;
}
```

i_{input}

$i = i_{input}, \ j = 2^* \ i_{input}$

$i = i_{input} + 1, \ j = 2^* \ i_{input}$

$i = 2^* \ i_{input}$^2 + $2^* \ i_{input}$

Concrete Execution Versus Symbolic Execution

```
int foo(int i){
        int j = 2*i;
        i = i++;
        i = i * j;
        if ( i < 1 )
                i =   -i;
        return i;
}
```

i_{input}

$i = i_{input}, \; j = 2 * i_{input}$

$i = i_{input} + 1, \; j = 2 * i_{input}$

$i = 2 * i_{input} \wedge 2 + 2 * i_{input}$

$$\boxed{\begin{array}{l} i = - 2 * i_{input} \wedge 2 - 2 * i_{input} \\ (2 * i_{input} \wedge 2 + 2 * i_{input} < 1) \end{array}}$$

OR

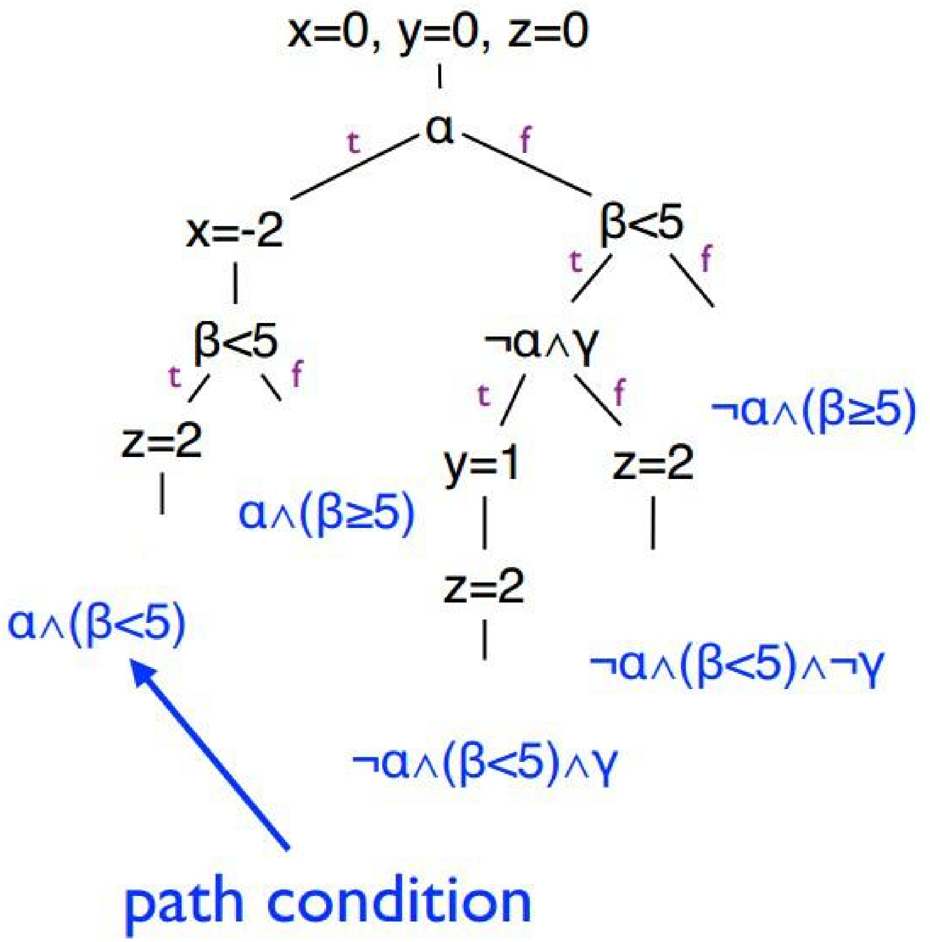
$$\boxed{\begin{array}{l} i = 2 * i_{input} \wedge 2 + 2 * i_{input} \\ (2 * i_{input} \wedge 2 + 2 * i_{input} >= 1) \end{array}}$$

Some Insights about Symbolic Execution

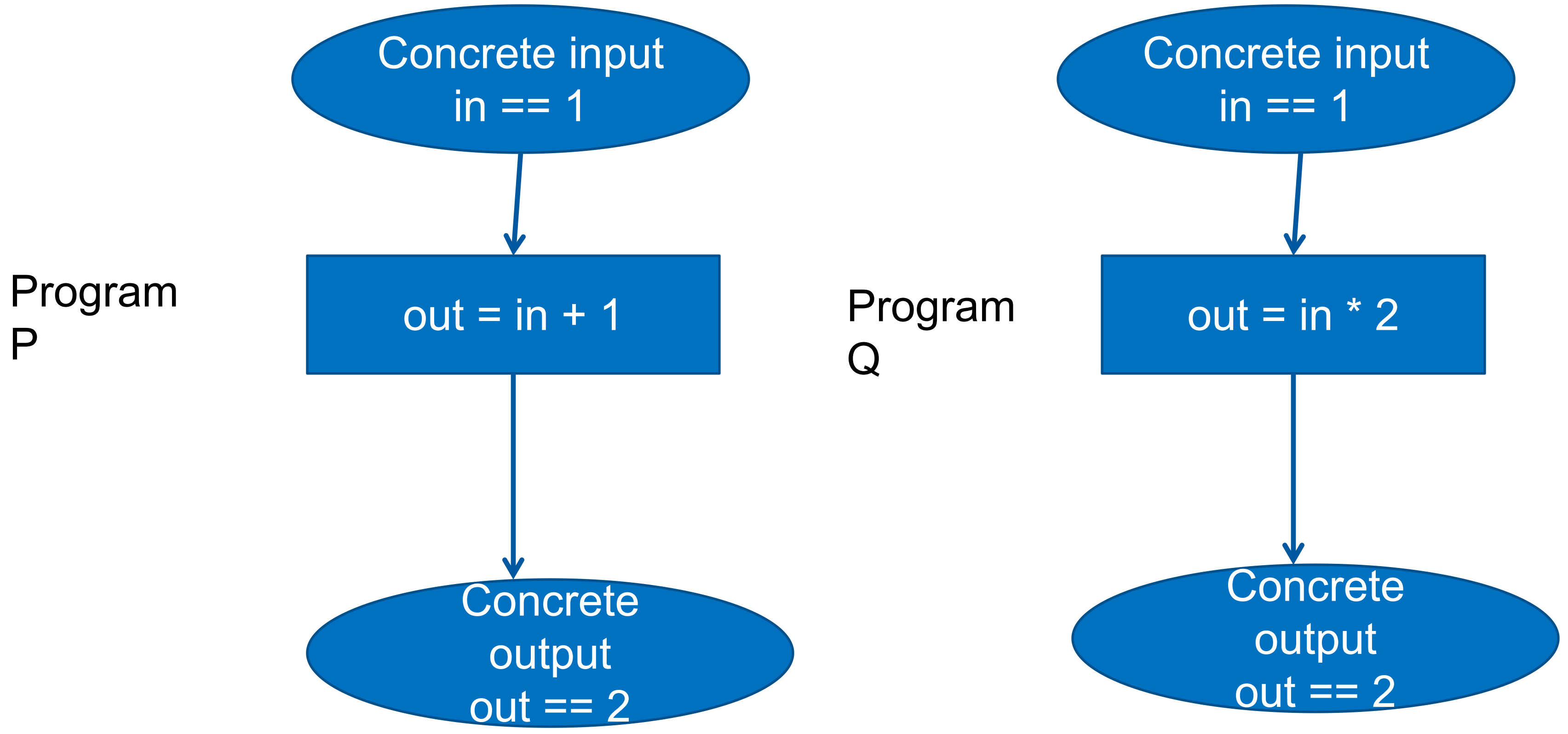
- ”Execute” programs with symbols: we track symbolic state rather than concrete input
- ”Execute” many program paths simultaneously: when execution path diverges, fork and add constraints on symbolic values
- When ”execute” one path, we actually simulate many test runs, since we are considering all the inputs that can exercise the same path

Symbolic Execution Tree

```
int a = α, b = β, c = γ;
                // symbolic
int x = 0, y = 0, z = 0;
if (a) {
  x = -2;
}
if (b < 5) {
  if (!a && c)  { y = 1; }
  z = 2;
}
assert(x+y+z!=3)
```

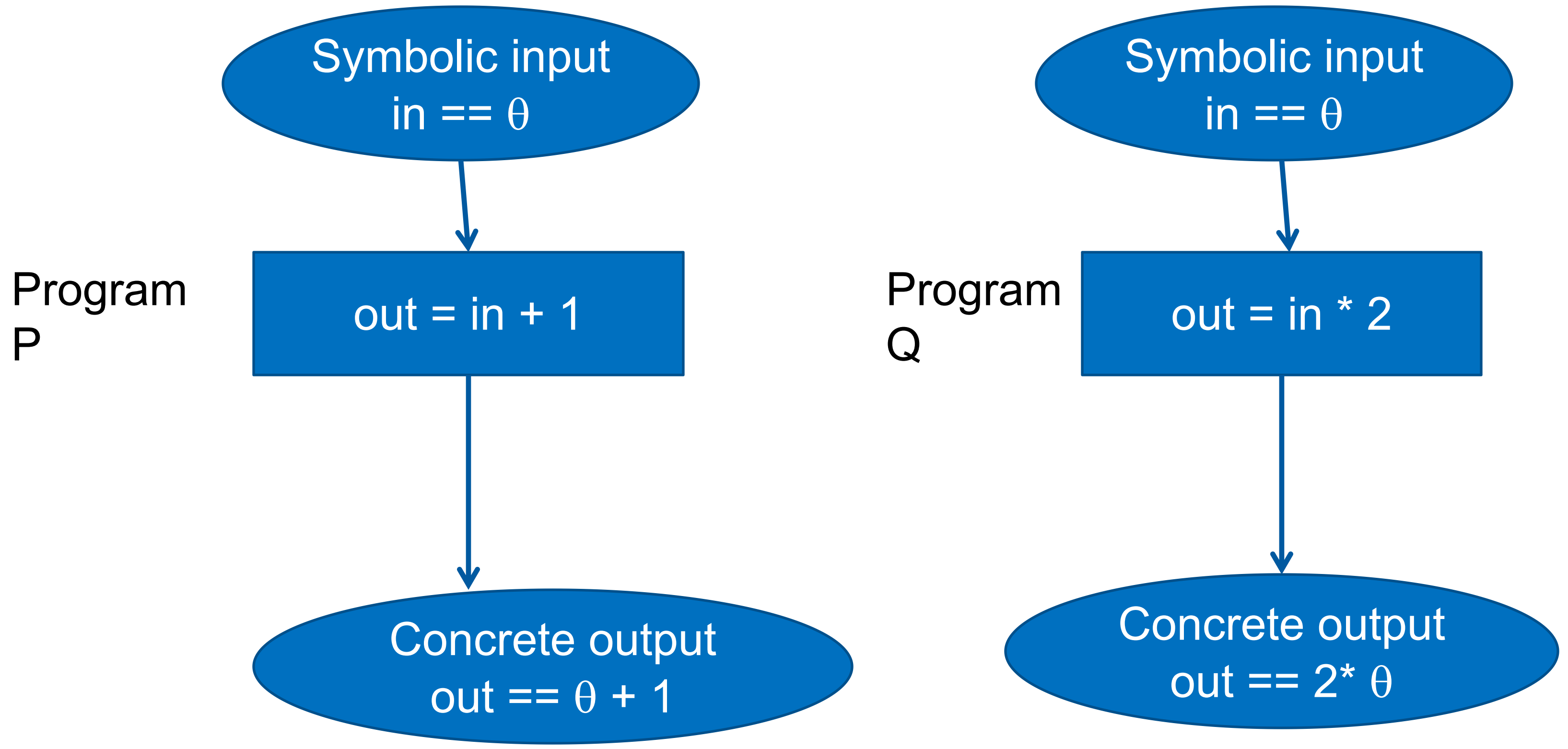


Concrete execution



No observable difference!

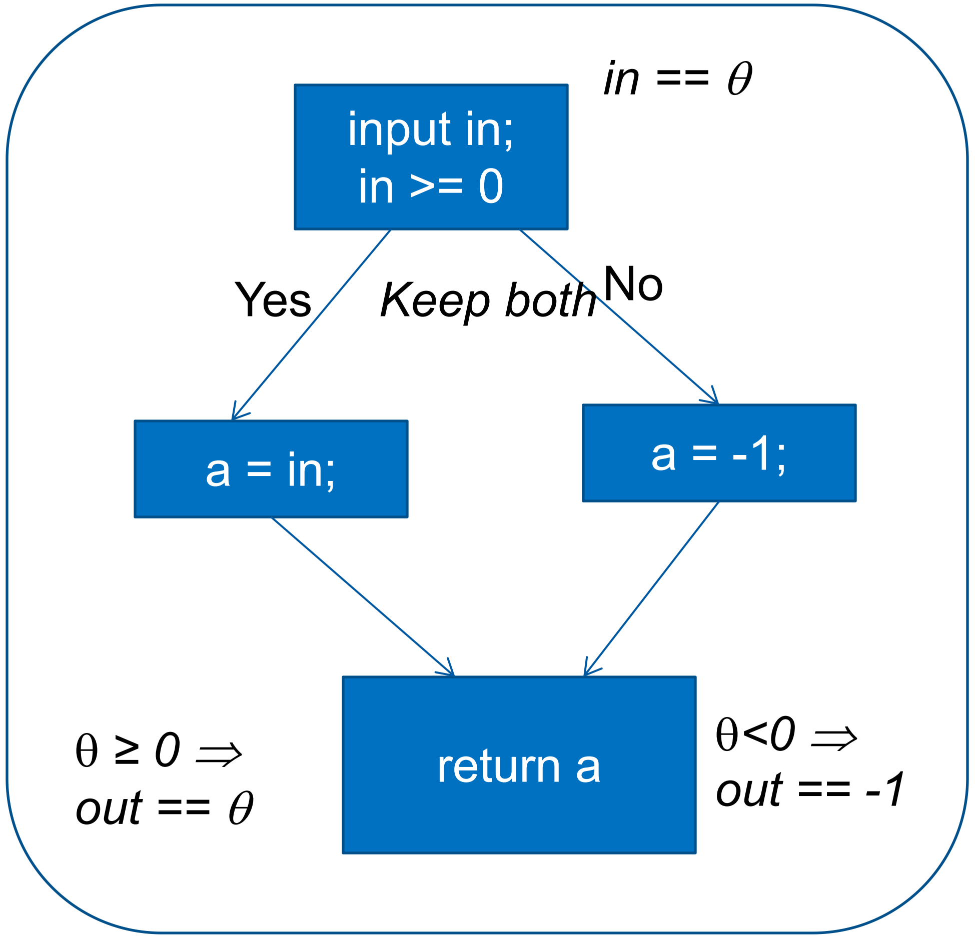
Execution with symbolic inputs



To expose difference, try to find θ such that $\theta + 1 \neq 2 * \theta$

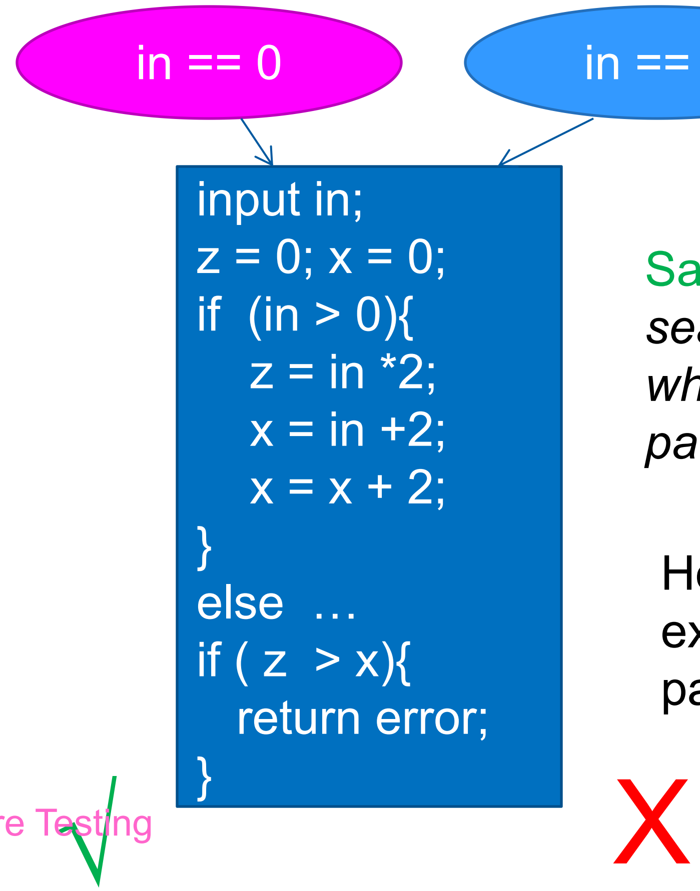
Path exploration based symbolic execution

```
input in;

if  (in >= 0)
     a = in;
else
     a = -1;

return a;
```



On-the-fly path exploration

Instead of analyzing the whole program, shift from one program path to another.



```
input in;
z = 0; x = 0;
if  (in > 0){
    z = in *2;
    x = in +2;
    x = x + 2;
}
else  …
if ( z  > x){
   return error;
}
```

Sample exploration: *Continue the search for failing inputs. Try those which do not go through the "same" path.*

How to perform symbolic execution along a single path?

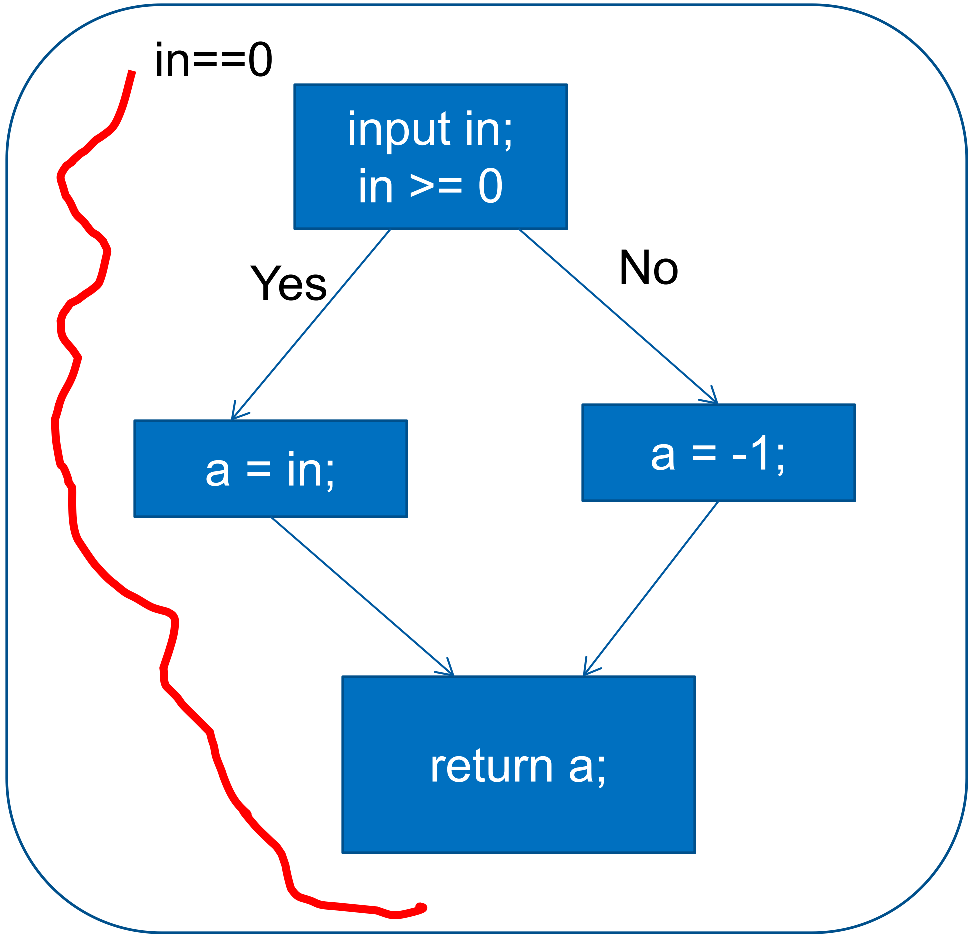
Exploring one path

Useful to find:

“the set of all inputs which trace a given path”

-> *Path condition*

in ≥ 0



Path condition computation

in == 5

```
1 input in;
2 z = 0; x = 0;
3 if  (in > 0){
4     z = in *2;
5   x = in +2;
6   x = x + 2;
7  }
8 else  …
9 if ( z  > x){
    return error;
}
```

Line#	Assignment store	Path condition
1	{}	*true*
2	{(z,0),(x,0)}	*true*
3	{(z,0),(x,0)}	$in > 0$
4	{(z,2*in), (x,0)}	$in > 0$
5	{(z,2*in), (x,in+2)}	$in > 0$
6	{(z,2*in), (x, in+4)}	$in > 0$
7	{(z, 2*in), (x, in+4)}	$in > 0$
9	{(z, 2*in), (x, in+4)}	$in>0 \wedge (2*in > in +4)$

Use of path conditions – (1)

in == 5

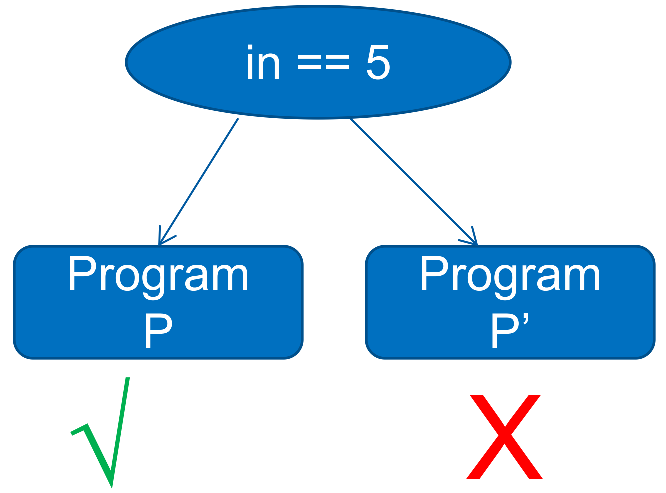
```
input in;
z = 0; x = 0;
if  (in > 0){
    z = in *2;
    x = in +2;
    x = x + 2;
}
else  …
if ( z  > x){
   return error;
}
```

X

Failing inputs: *Which other inputs follow the "same" error path?*

Path condition of in == 5

$in>0 \wedge (2*in > in+4) \equiv in > 4$

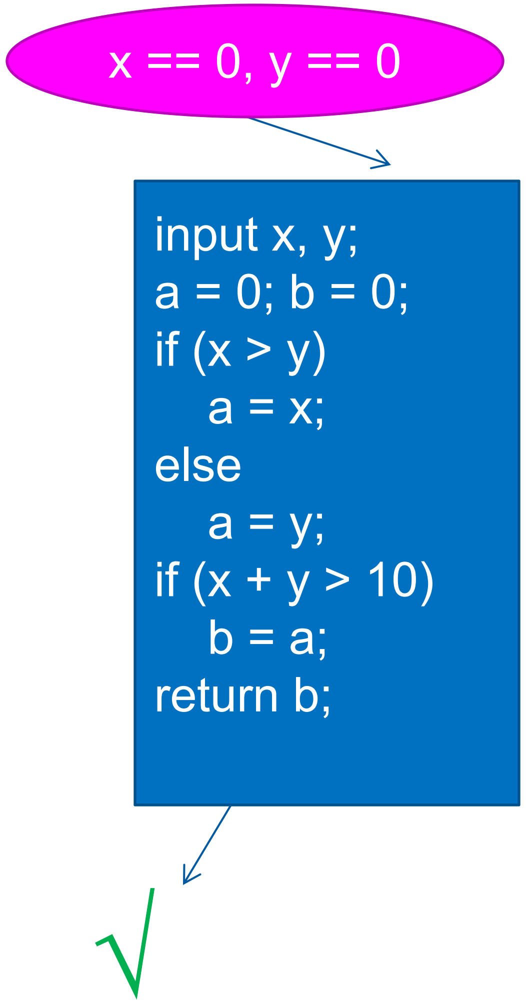


Can help in

Regression debugging

DARWIN (later)

Use of path conditions – (2)



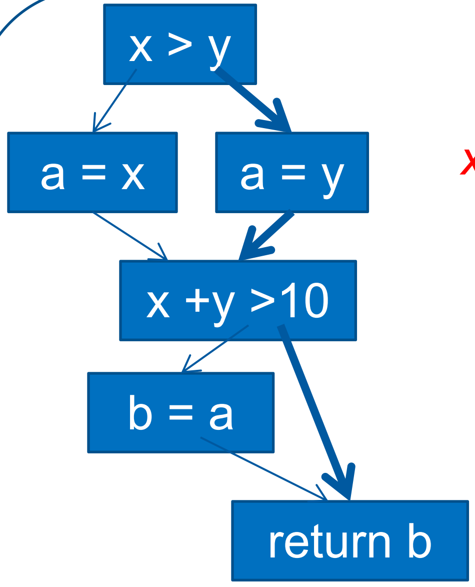
```
input x, y;
a = 0; b = 0;
if (x > y)
    a = x;
else
    a = y;
if (x + y > 10)
    b = a;
return b;
```

√

Passing inputs: *Continue the search for failing inputs, those which do not go through the "same" path.*

Path condition of (x == 0, y == 0)

$x \le y \wedge x + y \le 10$



Cover more paths

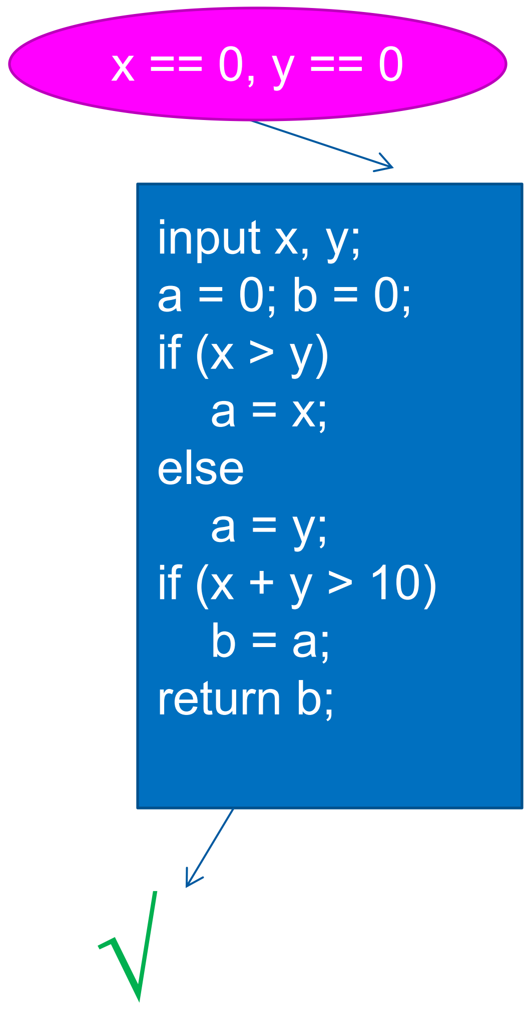
$x \le y \wedge x + y \le 10$

$x \le y \wedge \neg x + y \le 10$

$\neg x \le y$

Directed	D
Automated	A
Random	R
Testing	T

Use of path conditions – (1)

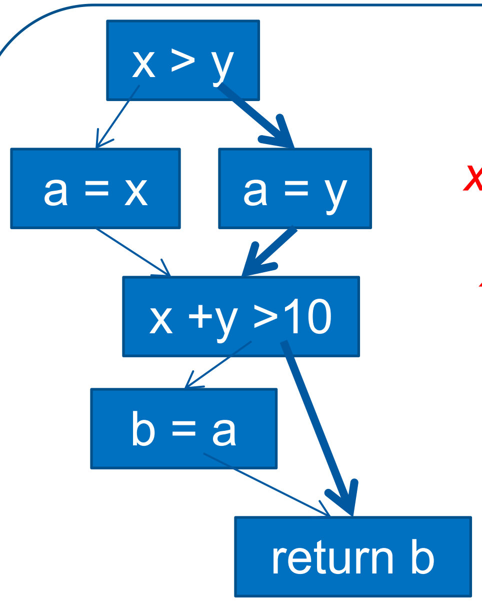


```
input x, y;
a = 0; b = 0;
if (x > y)
    a = x;
else
    a = y;
if (x + y > 10)
    b = a;
return b;
```

Passing inputs: *Continue the search for failing inputs, those which do not go through the “same” path.*

Path condition of (x == 0, y == 0)

$x \leq y \wedge x + y \leq 10$



Cover more paths

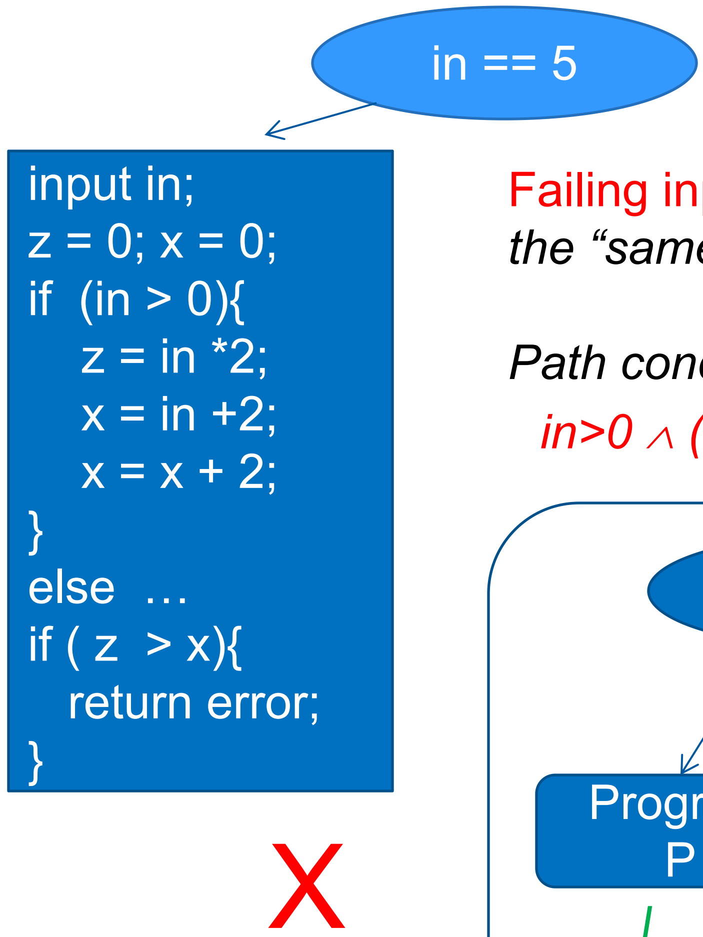
$x \leq y \wedge x + y \leq 10$

$x \leq y \wedge \neg x + y \leq 10$

$\neg x \leq y$

Directed	D
Automated	A
Random	R
Testing	T

Use of path conditions – (2)

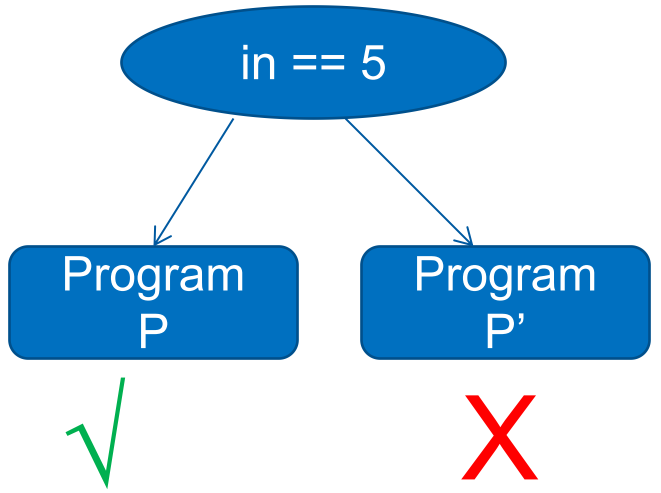


```
input in;
z = 0; x = 0;
if  (in > 0){
    z = in *2;
    x = in +2;
    x = x + 2;
}
else  …
if ( z  > x){
   return error;
}
```

Failing inputs: *Which other inputs follow the “same” error path?*

Path condition of in == 5

$in>0 \wedge (2*in > in + 4) \equiv in > 4$



Can help in

Regression debugging

DARWIN (later)

Outline of today’s briefing

- Symbolic execution
- Debugging
- Sample symbolic debugging techniques
  - Regression errors [FSE09, 10, …]
  - Cause clue clauses [PLDI11]
  - Error invariants [FM12]
  - Angelic debugging [ICSE11]

A quote from many years ago

"Even today, debugging remains very much of an art. Much of the computer science community has largely ignored the debugging problem….. over 50 percent of the problems resulted from the time and space chasm between symptom and root cause or inadequate debugging tools."

Hailpern & Santhanam, IBM Systems Journal, 41(1), 2002

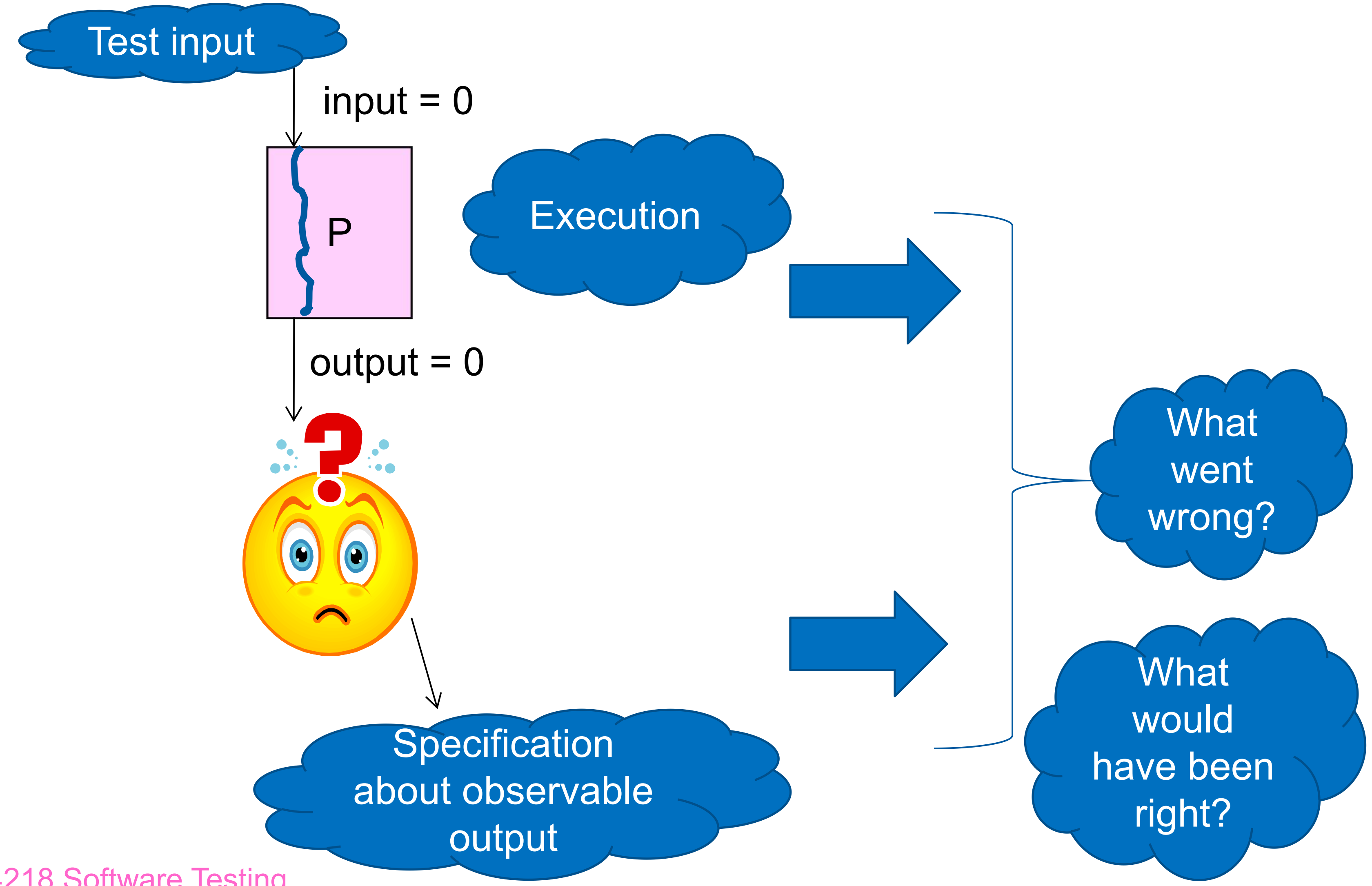
Any progress in 2002 – 2014?

- How can symbolic execution help?

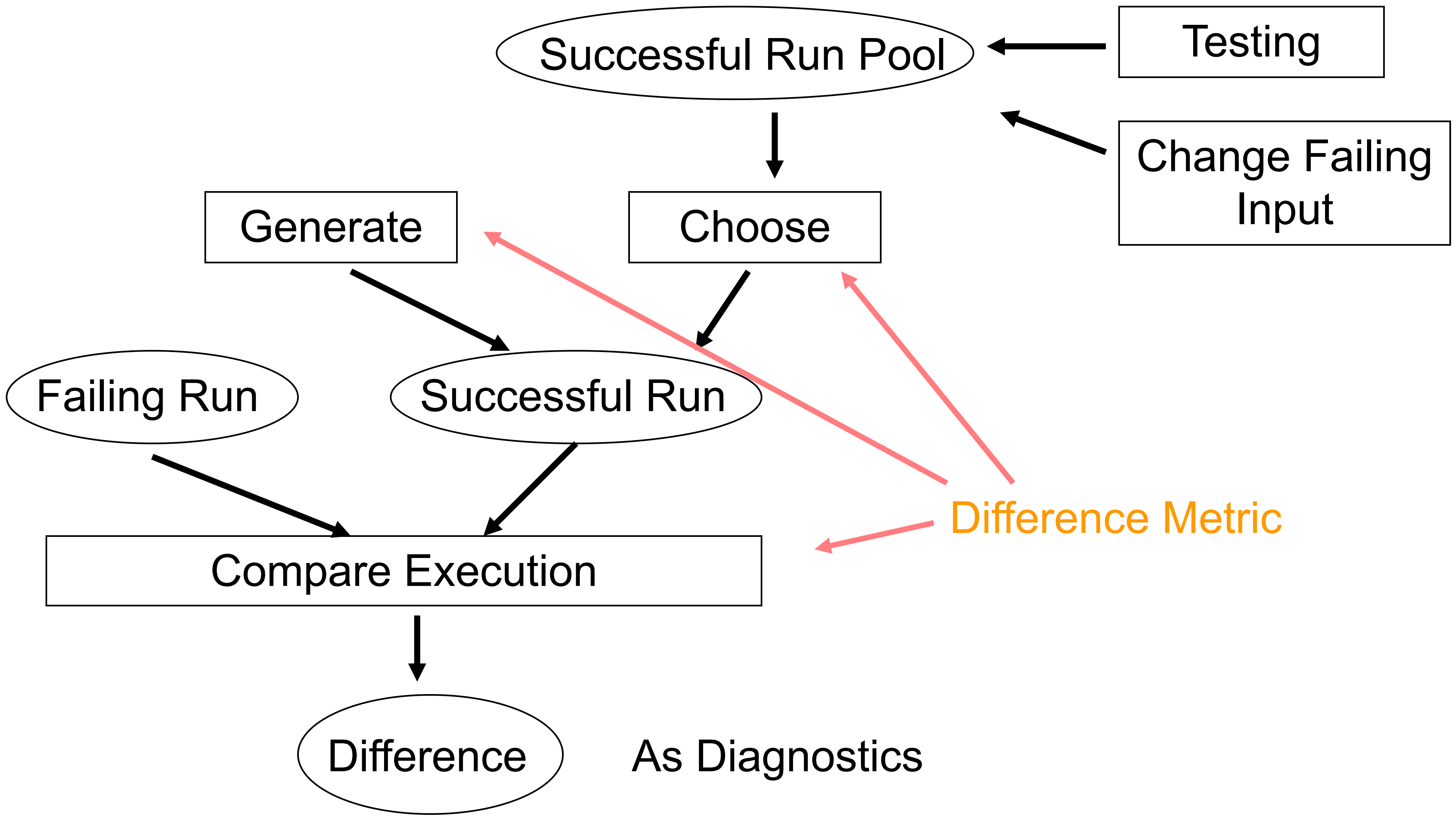
Debugging vs. Bug Hunting

- Debugging
  - Have a problematic input i, or ``counter-example" trace.
  - Does not match expected output for i.
  - Not sure what desired "property" is violated.
  - Amounts to implicitly alerting programmer about program's intended specifications as well.
- Bug Hunting via Model Checking / SE
  - Have a desired "property"
  - Tries to find a counter-example trace, and hence an input which violates the property.

Debugging: comparing to the intended behavior


Test input
input = 0
P
output = 0
Execution
Specification about observable output
What went wrong?
What would have been right?

Trace Comparison based Debugging


Successful Run Pool
Testing
Change Failing Input
Generate
Choose
Failing Run
Successful Run
Difference Metric
Compare Execution
Difference
As Diagnostics