Lecture 21: Delta Debugging

CS 5150, Spring 2025

Administrative Reminders

- Select a final presentation slot as early as possible!
  - Can be anytime between April 18 and May 10
- Assignment A4 released!
  - Due May 5

- Client meeting:
  - 1 meeting in sprint 4 (mar 25 – April 18) and
  - 1 meeting for last sprint (final presentation)

Debugging

Goal: Simplify a bug-revealing/failing input

Why is this needed?

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 explain
- Easier debugging: smaller test cases result in smaller states and shorter executions
- Identify duplicates: simplified test cases subsume several duplicates

A 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<OPTI ON
VALUE="minor">minor<OPTION VALUE="trivial">trivial<OPTION VALUE="enhancement">enhancement</SELECT>
</tr>
</table>
```

File → **Print** → **Segmentation Fault**

… to this?

<SELECT>

File → Print → Segmentation Fault

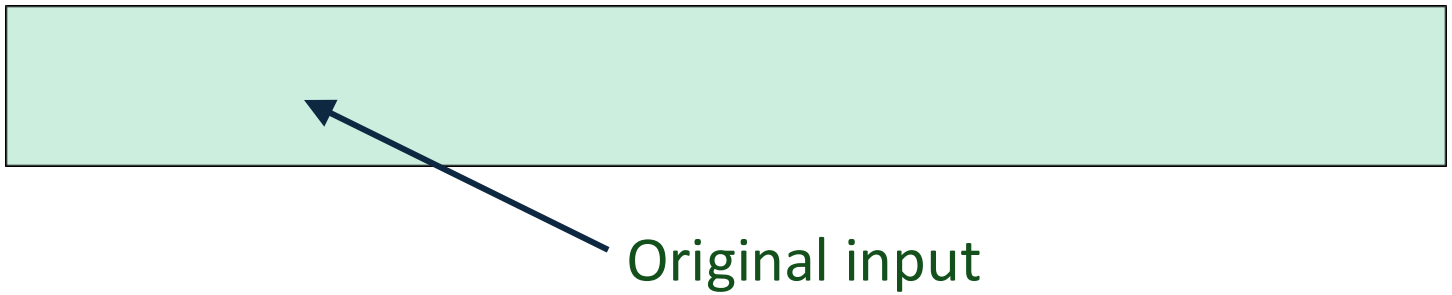
A First Solution

- 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.

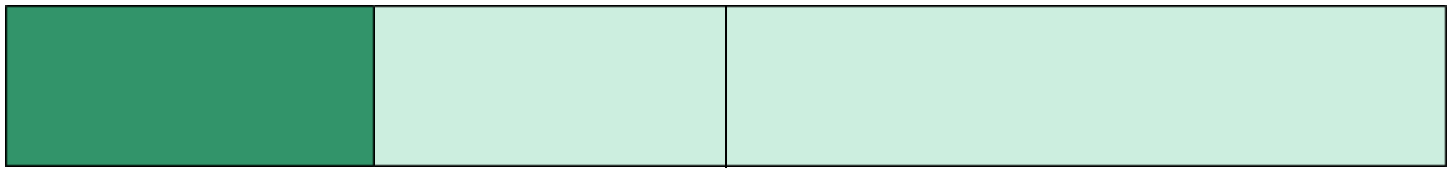
Binary Search

- 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.

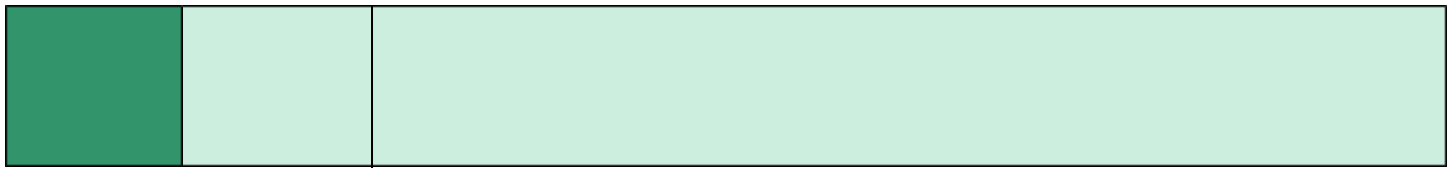
- If not, go back to the previous state and discard the other half of the input.



Binary Search

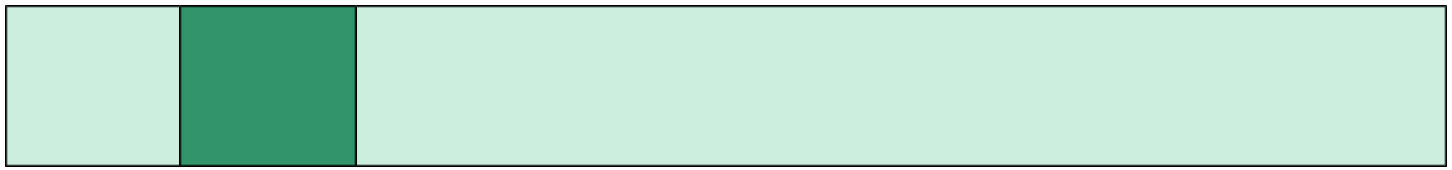
- Proceed by binary search. Throw away half the input and see if the output is still wrong.

- If not, go back to the previous state and discard the other half of the input.



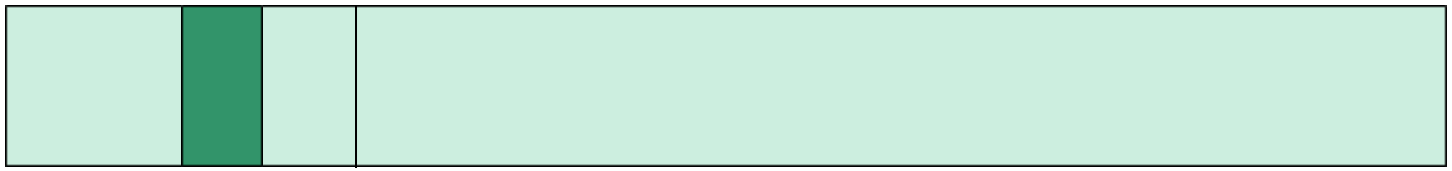
Binary Search

- 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

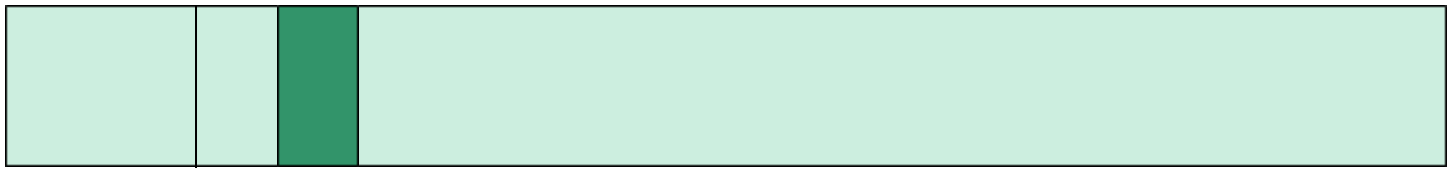
- Proceed by binary search. Throw away half the input and see if the output is still wrong.
- If not, go back to the previous state and discard the other half of the input.



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.

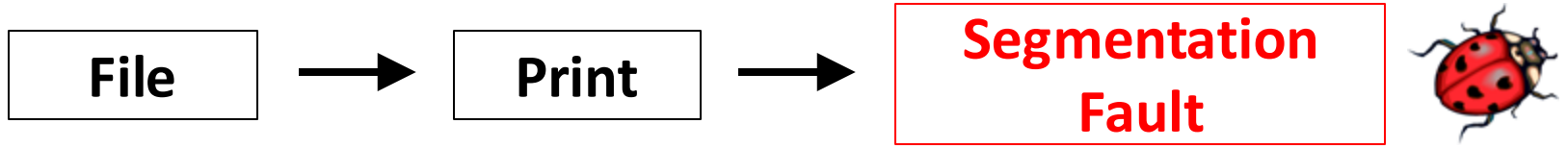
- If not, go back to the previous state and discard the other half of the input.

Delta Debugging Groundwork

Choosing Input Granularity

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<OPTI ON
VALUE="minor">minor<OPTION VALUE="trivial">trivial<OPTION VALUE="enhancement">enhancement</SELECT>
</tr>
</table>
```



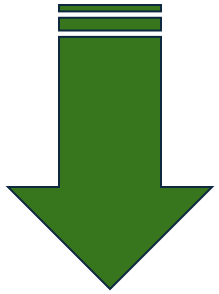
Simplified Input

```
<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

- Fewer and larger 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

Key Insight of Delta Debugging

Key Insight of Delta Debugging

- Fewer and larger 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

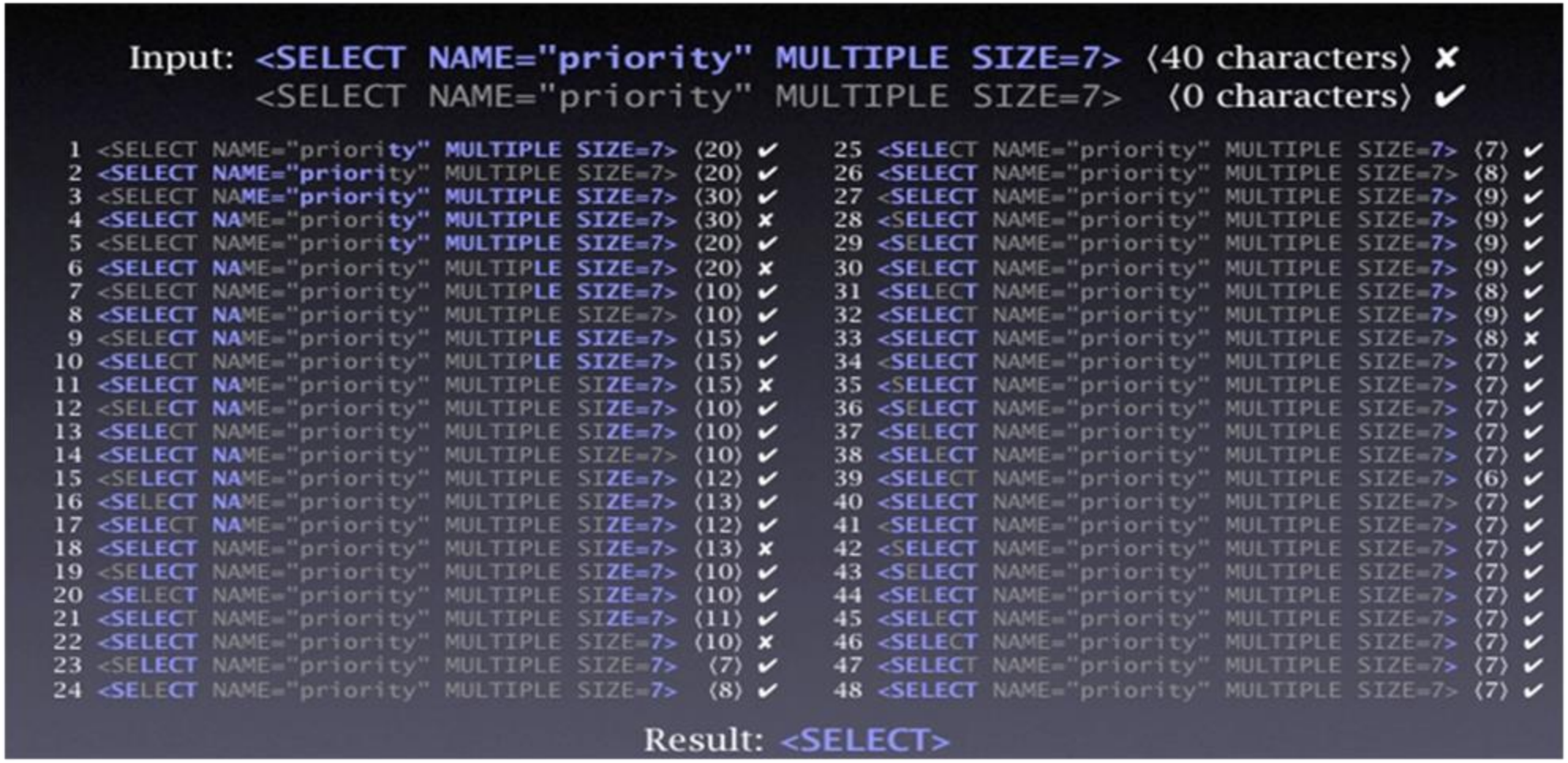
```
<SELECT NAME="priority" MULTIPLE SIZE=7>
<SELECT NAME="priority" MULTIPLE SIZE=7>
<SELECT NAME="priority" MULTIPLE SIZE=7>
<SELECT NAME="priority" MULTIPLE SIZE=7>
<SELECT NAME="priority" MULTIPLE SIZE=7>
<SELECT NAME="priority" MULTIPLE SIZE=7>
```

✘
✔
✔
✔
✘
✘

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>
```

Continuing Delta Debugging



Test Cases as Sets of Changes

Inputs and Failures

- Let **R** denote 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>
```

✘
✔
✔
✔
✘
✘
✔

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 δ 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
  $\boldsymbol{\delta}_1, \boldsymbol{\delta}_2, \ldots, \boldsymbol{\delta}_n$ where

$$\boldsymbol{\delta} = \boldsymbol{\delta}_1 \circ \boldsymbol{\delta}_2 \circ \ldots \circ \boldsymbol{\delta}_n \text{ and } (\boldsymbol{\delta}_i \circ \boldsymbol{\delta}_j)(\mathbf{r}) = \boldsymbol{\delta}_j(\boldsymbol{\delta}_i(\mathbf{r}))$$

- 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: δ' = insert `ME="priori` at input position 10

can be decomposed as $\delta' = \delta_1 \circ \delta_2 \circ \ldots \circ \delta_{10}$

where δ_1 = insert `M` at position 10

δ_2 = insert `E` at position 11

. . .

Summary

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

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

- Each subset $\mathbf{c}$ of $\mathbf{c_F}$ is a test case

Delta Debugging Algorithm

Minimizing Test Cases

Testing Test Cases

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

- We define the function test: Powerset($\mathbf{c_F}$) $\longrightarrow$ { P, F, ? } such that, given $\mathbf{c} = \{\boldsymbol{\delta}_1, \boldsymbol{\delta}_2, \ldots, \boldsymbol{\delta}_n\} \subseteq \mathbf{c_F}$

$$\text{test}(\mathbf{c}) = \text{F iff } (\boldsymbol{\delta}_1 \circ \boldsymbol{\delta}_2 \circ \ldots \circ \boldsymbol{\delta}_n)(\mathbf{r_P}) \text{ is a failing input}$$

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 $\mathbf{c'} \subseteq \mathbf{c_F}$, $|\mathbf{c'}| < |\mathbf{c}| \Rightarrow$ test(**c'**) ≠ 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 (2^n if $\mathbf{c_F}$ has size n)

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 called 1-minimality

Minimizing Test Cases

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

$$\text{for all } \mathbf{c'} \subset \mathbf{c}\ ,\ \text{test}(\mathbf{c'}) \neq F$$

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

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

- A failing test case is 1-minimal if:

$$\text{for all } \boldsymbol{\delta_i} \in \mathbf{c}\ ,\ \text{test}(\mathbf{c} - \{\boldsymbol{\delta_i}\}) \neq F$$

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 **bababab** 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

Minimization Algorithm

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 $N + (N-1) + (N-2) + \ldots$

- This is $O(N^2)$

Work Smarter, Not Harder

- We can often do better

- It is silly to start removing only one element at a time
  - Try dividing the change set into 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 searches for a 1-minimal test case
- It partitions the set c_F to $\Delta_1, \Delta_2, \ldots, \Delta_n$
  - $\Delta_1, \Delta_2, \ldots, \Delta_n$ are pairwise disjoint, and $c_F = \Delta_1 \cup \Delta_2 \cup \ldots \cup \Delta_n$
- Define the complement of Δ_i as $\nabla_i = c_F - \Delta_i$
- Start with **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. **n = n * 2**)

Steps of the Minimization Algorithm

1. Start with $n = 2$ and Δ as test set
2. Test each $\Delta_1, \Delta_2, \ldots, \Delta_n$ and each $\nabla_1, \nabla_2, \ldots, \nabla_n$
3. There are three possible outcomes:
   a. Some Δ_i causes failure: Go to step (1) with $\Delta = \Delta_i$ and $n = 2$
   b. Some ∇_i causes failure: Go to step (1) with $\Delta = \nabla_i$ and $n = n - 1$
   c. No test causes failure:

   If granularity can be refined: Go to step (1) with $\Delta = \Delta$ and $n = n * 2$

   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	2	2424	24
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

Real-World Applications

Debugging a Crashing Compiler

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)
}
```

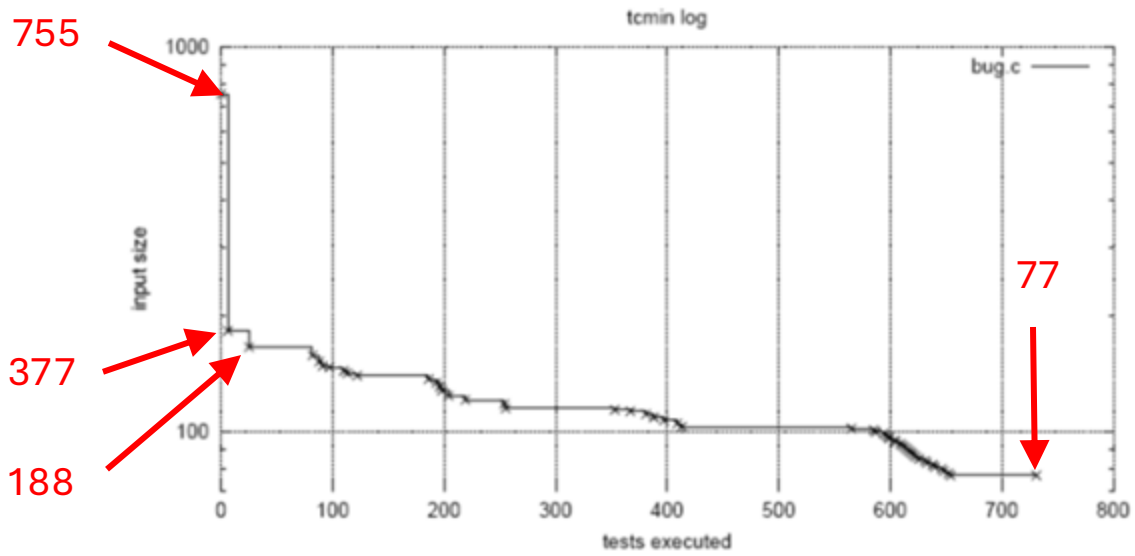
- This program (bug.c) crashes GCC 2.95.2 when optimization is enabled
- Goal: minimize this program to file a bug report
- For GCC, a passing run is the empty input
- For simplicity, model each change as insertion of a single character
  - test $\mathbf{r_P}$ = running GCC on an empty input
  - test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

[https://www.st.cs.uni-saarland.de/publications/files/zeller-tse-2002.pdf]

Case Study: GNU C Compiler

The test procedure:

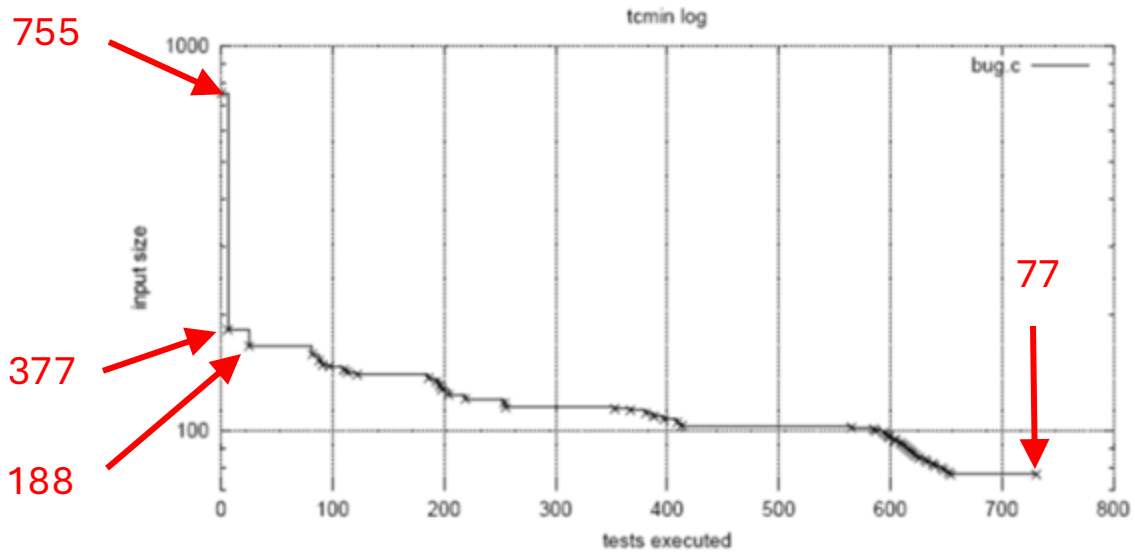
- create the appropriate subset of bug.c
- feed it to GCC
- return **Failed** if GCC crashes, **Passed** otherwise



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 r_P = run GCC with all options
  - test r_F = run GCC with no option
  - change δ_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:

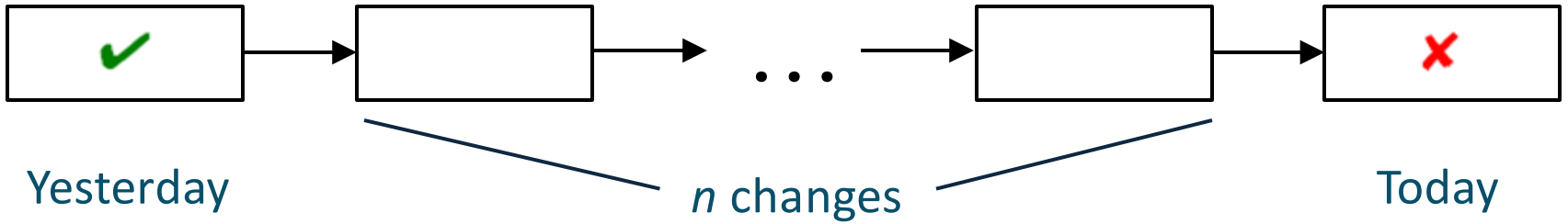
- The minimal test case
- “The crash only occurs with optimization”
- “**-ffast-math** and **-fforce-addr** prevent the crash”

Other Debugging Applications

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 a 10^6 character input crashing CRTPLOT to a single character in only 24 tests!

Case Study: Isolating Failure-Inducing Changes



- 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 1-minimal instead of local minimum test case due to performance.
- ◯ Finds the smallest failing subset of a failing input in polynomial time.
- ◯ 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 1-minimal instead of local minimum test case due to performance.
- ○ Finds the smallest failing subset of a failing input in polynomial time.
- ● 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?

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