软件分析与验证前沿

苏亭
软件科学与技术系

Delta Debugging

Simplifying and isolating failure-inducing input (TSE'02, by Andreas Zeller, citation: 1371)

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>
```

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

… to this?

<SELECT>

File → Print → Segmentation Fault

Your Solution

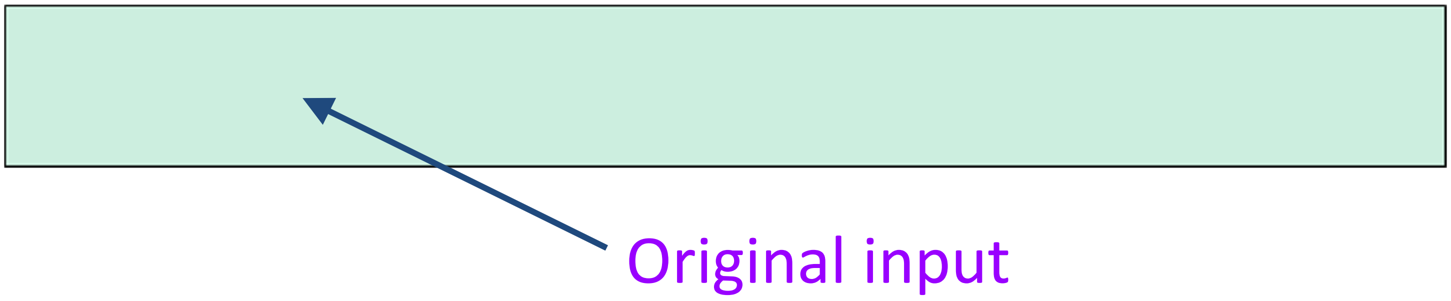
- How do you solve these problems?

- 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.

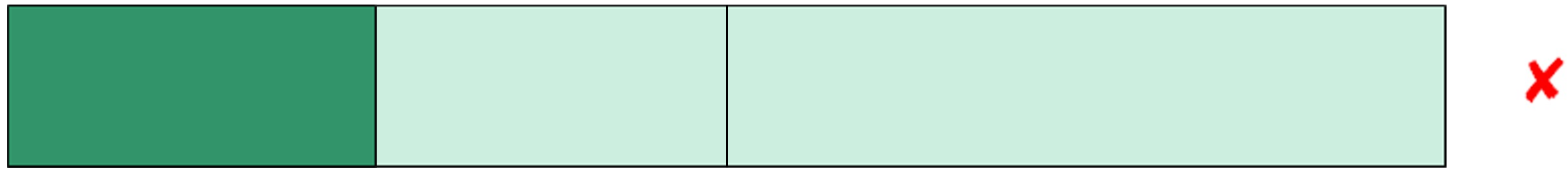
Binary Search

- 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.

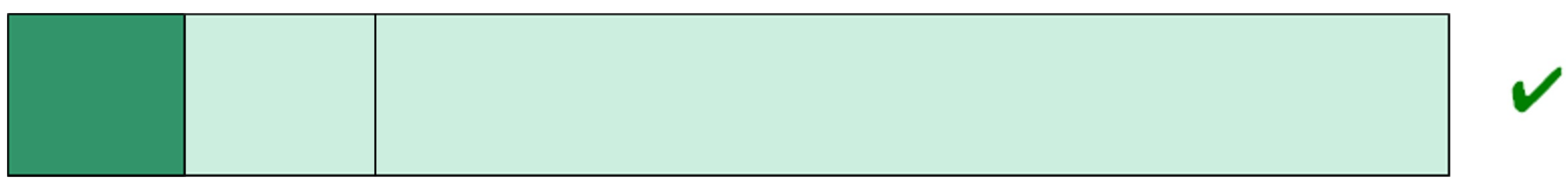
- If not, go back to the previous state and discard the other half of the input.



Binary Search

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

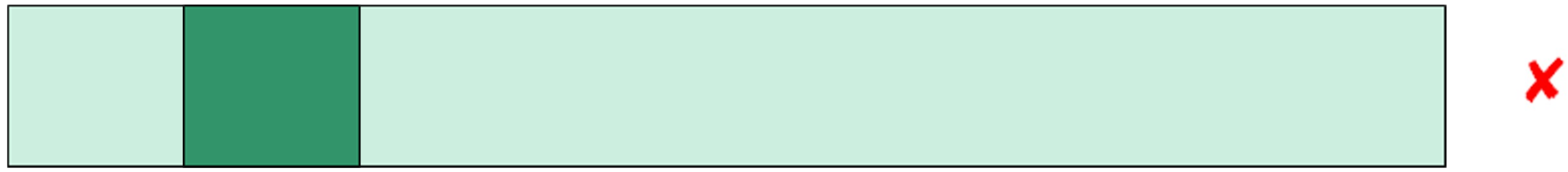
- If not, go back to the previous state and discard the other half of the input.



Binary Search

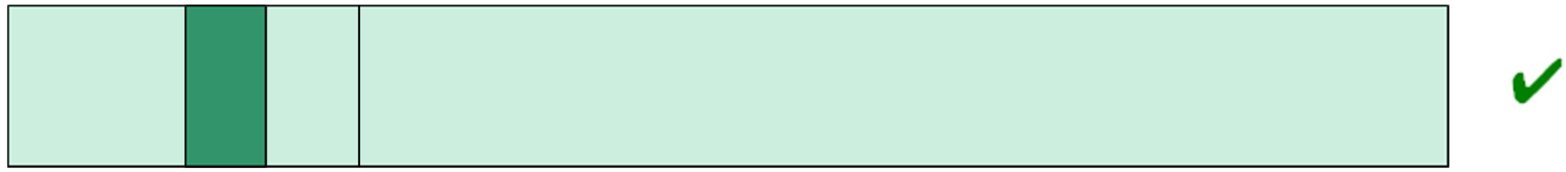
- Proceed by binary search. Throw away half the input and see if the output is still wrong.

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



Binary Search

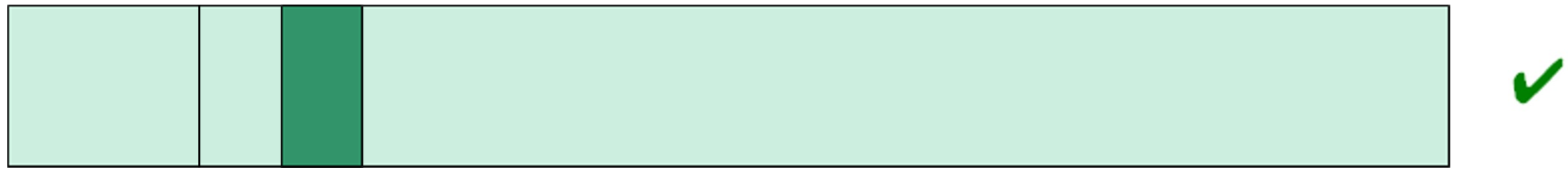
- Proceed by binary search. Throw away half the input and see if the output is still wrong.
- If not, go back to the previous state and discard the other half of the input.



Binary Search

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

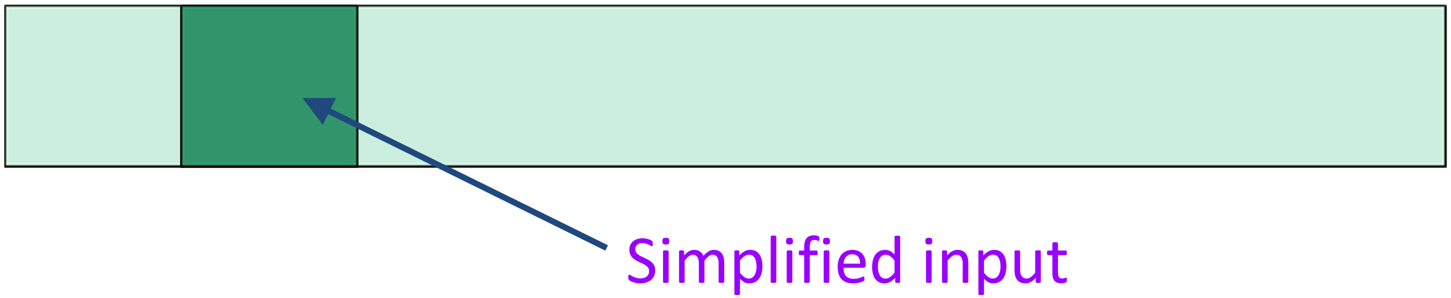
- If not, go back to the previous state and discard the other half of the input.



Binary Search

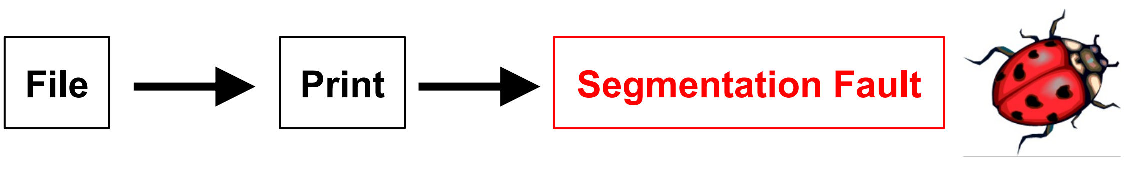
- Proceed by binary search. Throw away half the input and see if the output is still wrong.

- 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>
```



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> ✓

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>   ✔
```

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 $\boldsymbol{\delta}_1, \boldsymbol{\delta}_2, \ldots, \boldsymbol{\delta}_n$ where $\boldsymbol{\delta} = \boldsymbol{\delta}_1 \circ \boldsymbol{\delta}_2 \circ \ldots \circ \boldsymbol{\delta}_n$ 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 $\boldsymbol{\delta}' = \boldsymbol{\delta}_1 \circ \boldsymbol{\delta}_2 \circ \ldots \circ \boldsymbol{\delta}_{10}$

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 $c_F = \{ \boldsymbol{\delta}_1, \boldsymbol{\delta}_2, \ldots, \boldsymbol{\delta}_n \}$ such that:

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

- 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
  $\text{test}: \text{Powerset}(c_F) \longrightarrow \{P, F, ?\}$ such that,
  given $c = \{\boldsymbol{\delta}_1, \boldsymbol{\delta}_2, \ldots, \boldsymbol{\delta}_n\} \subseteq c_F$
  $\text{test}(c) = F$ iff $(\boldsymbol{\delta}_1 \circ \boldsymbol{\delta}_2 \circ \ldots \circ \boldsymbol{\delta}_n)(r_P)$ 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 c_F if:

  for all $\mathbf{c}' \subseteq \mathbf{c_F}$, $|\mathbf{c}'| < |\mathbf{c}| \Rightarrow \mathbf{test}(\mathbf{c}') \neq \mathbf{F}$
- 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 single 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 $\boldsymbol{\delta_i} \in \mathbf{c}$ **, test(c - {**$\boldsymbol{\delta_i}$ **}) ≠ 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:
- 1-minimal, of size 3:
- Local minimum but not smallest:
- 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	• 1-minimal, of size 3:	aab, aba, baa, bbb
• Local minimum but not smallest:	NONE	• 2-minimal, of size 3:	NONE

Naive Algorithm

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

```
if for all δᵢ ∈ c, test(c - {δᵢ}) ≠ F, then c is 1-minimal
else recurse on c - {δ} for some δ ∈ c, test(c - {δ}) = 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 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 $\mathbf{\Delta_1, \Delta_2, \ldots, \Delta_n}$
  - $\mathbf{\Delta_1, \Delta_2, \ldots, \Delta_n}$ are pairwise disjoint, and $\mathbf{c_F = \Delta_1} \cup \mathbf{\Delta_2} \cup \ldots \cup \mathbf{\Delta_n}$
- Define the complement of $\mathbf{\Delta_i}$ as $\nabla_i = \mathbf{c_F - \Delta_i}$
- 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**)

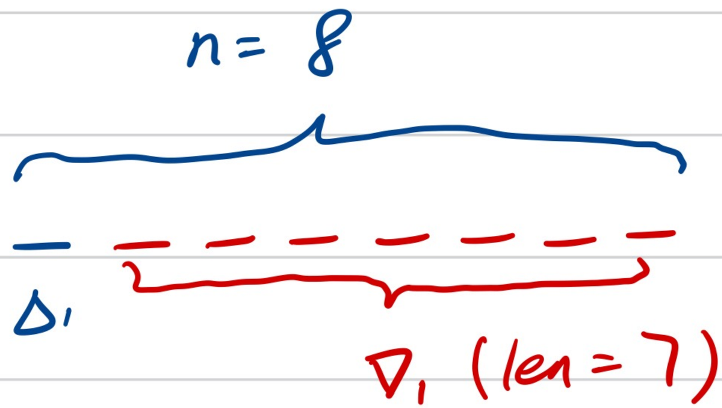
Steps of the Minimization Algorithm

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

      Otherwise: Done, found the 1-minimal subset

Δ_i failed, $\Delta_i = \nabla_i$, $n = n - 1$

example:



Delta Debugging中▽发现错误时，n-1的原因是保持粒度不变。如图，令n = 8，则▵长度为1，▽长度为7，转为在▽中查找时令n = n-1可保持查找的长度和▵一样（均为1）

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

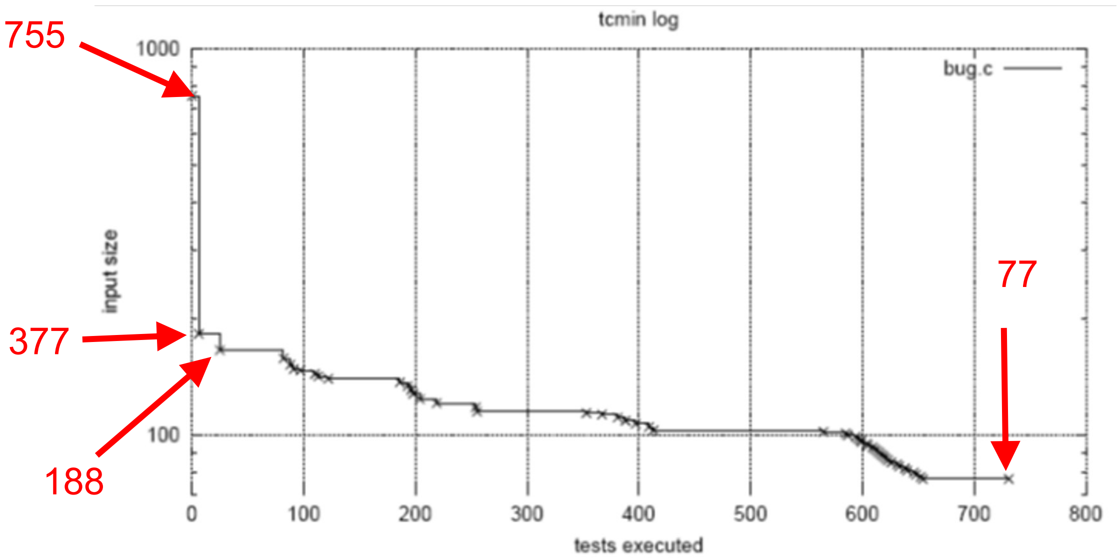
```
#define SIZE 20
double mult(double z[], int n) {
   int i, j;
   i = 0;
   for (j = 0; j < n; j++) {
                i = i + j + 1;
                z[i] = z[i] * (z[0] + 1.0);
   }
   return z[n];
}
void copy(double to[], double from[], int count) {
    int n = (count + 7) / 8;
    switch (count % 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:

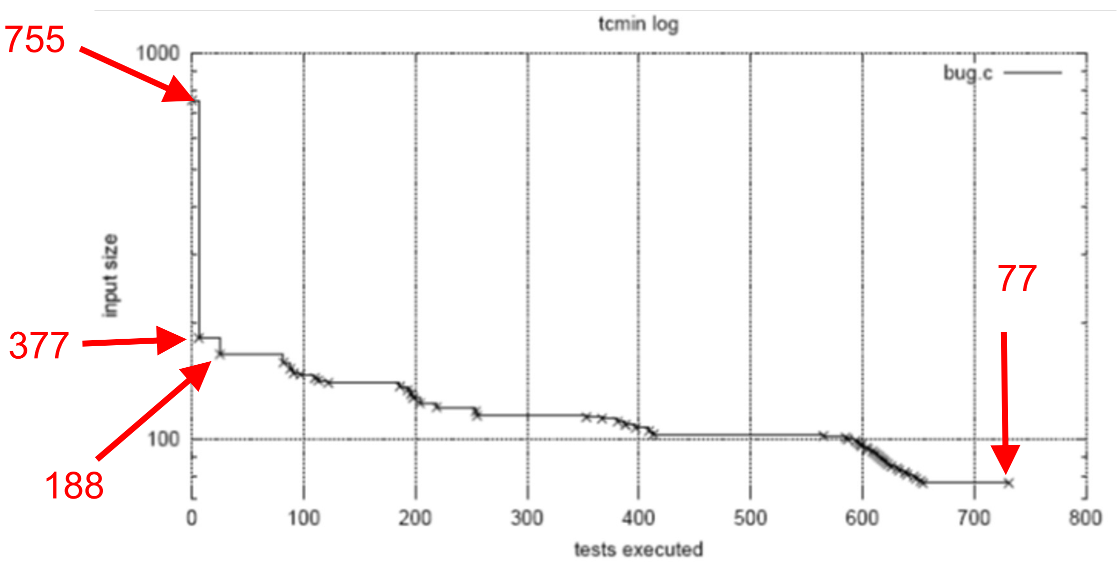
- create the appropriate subset of bug.c
- feed it to GCC
- return **Failed** if GCC crashes, **Passed** otherwise



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

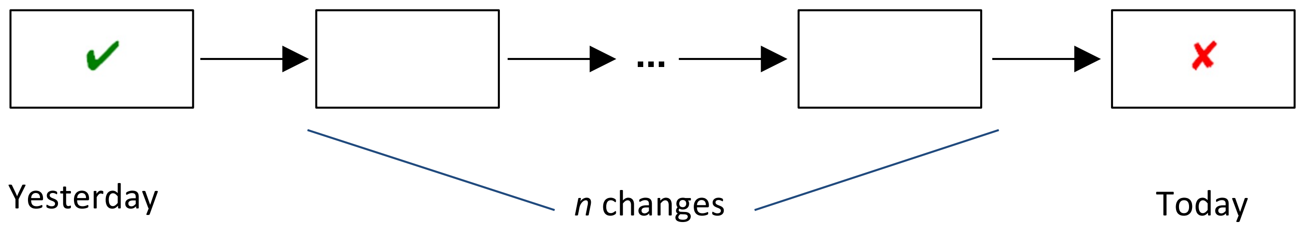
This is what we can send to the GCC maintainers:

- 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?

Yesterday, my program worked. Today, it does not. Why? Andreas Zeller. https://dl.acm.org/doi/pdf/10.1145/318774.318946

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