CS 563: Software Maintenance And Evolution

Delta Debugging

Oregon State University, Spring 2024

Today’s Plan

- Learn about Delta Debugging technique
- Finalize project ideas and groups
- Start in-class exercise#2

Why Simply Failing Test Input?

Once we have reproduced a program failure, we must find out what’s relevant.

- Does failure really depend on 10,000 lines code? (compiler testing)
- Does failure really require the exact schedule of events? (concurrency testing)
- Does failure really need this sequence of function calls? (GUI events)

Why Simply Failing Test Input?

- Ease of communication: a simplified test case is easier to explain
- Easier debugging: small test cases result in smaller states and shorter executions
- Identify duplicate issues: simplified test cases subsume duplicate test cases from several bug reports

A Real-World Scenario

In July 1999, Bugzilla listed more than 800 open bug reports for Mozilla's Firefox 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 …

Multiple bug reports on browser crashing for some 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>
```

File → Print → Segmentation Fault

… to this?

The crash occurs when SELECT tag is not closed

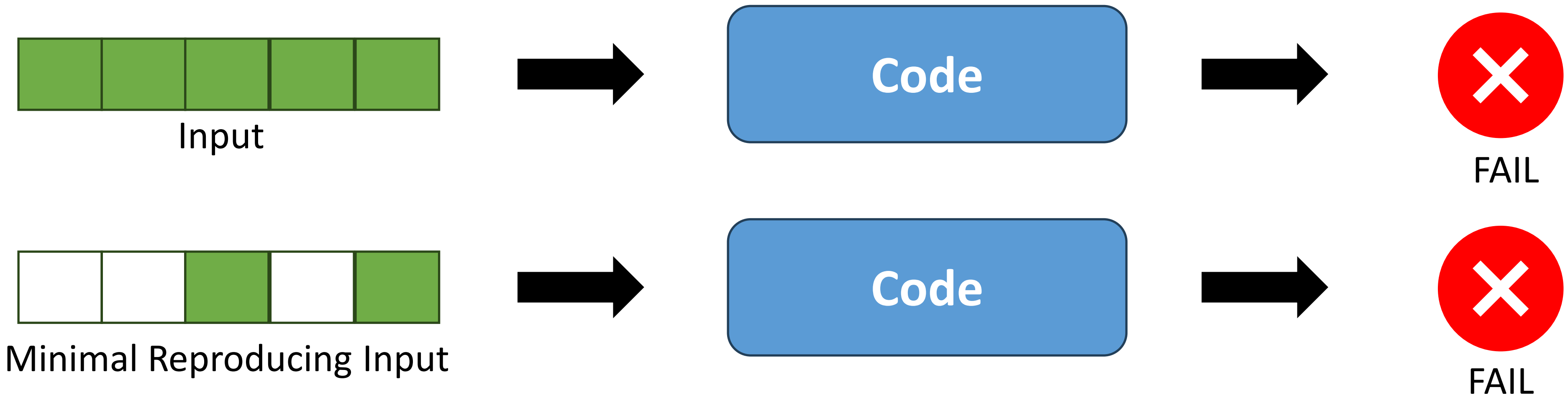
<SELECT>

File → Print → Segmentation Fault

Delta Debugging Simplified Usage Scenario

- Given: A program exhibits a failure on an input
  - Input = a source file for compiler
  - Input = a set of numbers for sorting
  - Input = a set of rows in DB for pre-processing
  - Input = a config file listing a set of configuration parameters
  - Input = a set of HTTP requests for a web application
  - Input = a sequence of API calls to a module
- Goal: Find the **smallest subset** of the input for which the program still has the **same failure**

Minimizing Bug Reproducing Input



- Important
  - Some subsets of the input may not cause the exact failure (FAIL)
    - Code may produce some other result
    - Code may give some other error message (e.g., syntax error for compiler testing)

A Generic Algorithm

- How do researchers solve these problem?
- Naïve brute-force
  - Select 1 element of input and try to reproduce bug
  - Select 2 elements of input and try to reproduce bug
  - Select n elements of input and try to reproduce bug
  - $O(2^n)$
- Binary search
  - Cut the input in halves
  - Try to reproduce the bug with half input
  - Recurse
  - O(Log n)

Delta Debugging: Assumptions

- There is a set of input elements *I*
  - If we use the entire *I*, we get a failure (FAIL)
  - Need to find the minimal reproducing input (MRI) that results in the same failure (MRI subset of *I*)
- Assumptions
  - A1: Every subset of the input that contains MRI will result in the same failure (monotonicity)

Delta Debugging: Monotonicity

- Examples of monotonicity:
  - Compiler crashes when “1e-100” appears in the code
    - “1e-100” is the MRI
    - The compiler will still crash if other tokens appear in addition to the MRI
  - Sorting fails when input list contains one negative and one positive number
    - {-1, 2} is the MRI
    - The sort function will still fail if other numbers appear in addition to the MRI
  - Web service crashed because of one HTTP request
    - The MRI is that request
    - The web service will still fail if other requests are made in addition to the MRI
- Examples of non-monotonicity?

Delta Debugging: Monotonicity

- Examples of monotonicity:
  - Compiler crashes when “1e-100” appears in the code, except if “1e+100” also appears
    - “1e-100” is the MRI
    - But “1e-100 * 1e+100” contains the MRI but does not crash compiler
  - Sorting fails when input list contains one negative and one positive number and when the input list have even number of elements
    - {-1, 2} is the MRI
    - But {-1, 0, 2} contains the MRI but it does not cause failure

Delta Debugging: Version 1

- One more simplifying assumption (for now)
  - A2: There exists **one** input element that causes the failure by itself
    - MRI is of size 1
    - E.g., one of the HTTP requests crashes the server
    - E.g., sorting fails when 0 is part of the input

Binary Search

- Proceed by binary search
- If input *I* results in failure
- Try the first half of the input, and see if it results in the same failure
  - If yes, continue search in first half of the input
  - If no, continue search in second half of the input

Version 1: Example

- Assume *I* = {1, 2, 3, 4, 5, 6, 7, 8}
  - The bug is due to the input element 7 (NOTE: Delta Debugging doesn't know this!)

Configuration	Result
{1, 2, 3, 4, 5, 6, 7, 8}	FAIL
{1, 2, 3, 4 }	PASS
{ 5, 6, 7, 8}	FAIL
{ 5, 6, }	PASS
{ 7, 8}	FAIL
{ 7, }	FAIL

MRI

Delta Debugging Algorithm for Version 1

- Invariant: P fails with inputs $i_1,\ldots,i_n$ i.e. $\{i_1, \ldots, i_n\}$ -> FAIL
- A2: There is one i_k that makes P fail
- Find: i_k

DD($\{i_1, \ldots, i_n\}$) =

if n == 1 return $\{i_1\}$

let $H_1 = \{i_1, \ldots, i_{n/2}\}$

let $H_2 = \{i_{n/2+1}, \ldots, i_n\}$

if H_1 -> FAIL

return DD(H_1)

else

assert H_2 -> FAIL (because of invariant must hold to invoke DD)

return DD(H_2)

Does invariant hold true?

Assertion can FAIL for non-monotonicity

Delta Debugging - Version 1: Comments

- Let's look at the assumptions we used so far
  - A2: There exists **one** input element that causes the failure by itself
    - MRI is of size 1
    - $(H_1 + H_2)$ -> FAIL then
      - Either H_1 -> FAIL and H_2 -> PASS
      - Or H_2 -> FAIL and H_1 -> PASS

- It becomes interesting when the MRI is of size larger than 1

Delta Debugging – MRI of size more than 1

- A sorting function that fails if the input contains both negative and positive values
  - MRI size is of size 2
- If you make two HTTP requests to delete all items from shopping cart, where second request crashes the service
  - MRI size is at least 2
- A compiler crash typically cannot be reproduced with a 1-character source file
  - MRI has more than 1 character

We need to drop the assumption that MRI is of size 1 to use delta debugging for these scenarios

Scenarios

- Try binary search
  - Partition input I into two halves H_1 and
  - If H_1 -> FAIL, recurse with H_1
  - Otherwise, if H_2 -> FAIL, recurse with H_2

Occurs when
MRI is of size > 1

- Notes:
  - The only other possibility is: H_1 -> PASS and H_2 -> PASS
  - How to deal with this case?

Interference

- By monotonicity, if H_1 -> PASS and H_2 -> PASS then
  - No subset of H_1 or H_2 causes failure by itself
  - Yet (H_1 + H_2) -> FAIL
- So the failure must be due to a combination of elements from H_1 and H_2
- This is called interference (the two halves interfere)
- Addressing interference is the major innovation in Delta Debugging, and the main enabler in practice.

Interference Example

- Assume I = {1, 2, 3, 4, 5, 6, 7, 8}
  - The bug is due to the input elements 3, 5, 7 (NOTE: Delta Debugging doesn't know this!)

Interference Example

- Assume I = {1, 2, 3, 4, 5, 6, 7, 8}
  - The bug is due to the input elements 3, 5, 7 (NOTE: Delta Debugging doesn't know this!)

Configuration	Result
{1 2 3 4 5 6 7 8}	FAIL
{1 2 3 4 }	PASS
{ 5 6 7 8}	PASS

Interference occurs, indicating we need a combination of the elements in the two halves

Interference Example

- Assume I = {1, 2, 3, 4, 5, 6, 7, 8}
  - The bug is due to the input elements 3, 5, 7 (NOTE: Delta Debugging doesn't know this!)

Configuration	**Result**
{1 2 3 4 5 6 7 8}	FAIL
{1 2 3 4 }	PASS
{ 5 6 7 8}	PASS
{1 2 3 4 5 6 }	PASS

IDEA: keep the first half and search or subset of second half such that the subset in combination with the first half results in FAIL

Interference Example

- Assume I = {1, 2, 3, 4, 5, 6, 7, 8}
  - The bug is due to the input elements 3, 5, 7 (NOTE: Delta Debugging doesn't know this!)

Configuration	**Result**
{1 2 3 4 5 6 7 8}	FAIL
{1 2 3 4 }	PASS
{ 5 6 7 8}	PASS
{1 2 3 4 5 6 }	PASS
{1 2 3 4 7 8}	PASS

Another interference

Interference Example

- Assume I = {1, 2, 3, 4, 5, 6, 7, 8}
  - The bug is due to the input elements 3, 5, 7 (NOTE: Delta Debugging doesn't know this!)

Configuration	**Result**
{1 2 3 4 5 6 7 8}	FAIL
{1 2 3 4 }	PASS
{ 5 6 7 8}	PASS
{1 2 3 4 5 6 }	PASS
{1 2 3 4 7 8}	PASS
{1 2 3 4 5 6 7 }	FAIL

Continue the process

Interference Example

- Assume I = {1, 2, 3, 4, 5, 6, 7, 8}
  - The bug is due to the input elements 3, 5, 7 (NOTE: Delta Debugging doesn't know this!)

Configuration	Result
{1 2 3 4 5 6 7 8}	FAIL
{1 2 3 4 }	PASS
{ 5 6 7 8}	PASS
{1 2 3 4 5 6 }	PASS
{1 2 3 4 7 8}	PASS
{1 2 3 4 5 6 7 }	FAIL
{1 2 3 4 5 7 }	FAIL
{1 2 5 7 }	PASS
{ 3 4 5 7 }	FAIL
{ 3 5 7 }	FAIL

Continue the process

Handling Interference

- Review: The cute trick
  - Consider $(H_1 + H_2)$ -> FAIL but H_1 -> PASS and H_2 -> PASS
    - Find minimal M_2 in H_2 such that $(H_1 + M_2)$ -> FAIL
    - All elements in M_2 are necessary for FAIL (along with some elements from H_1)
  - Consider $(H_1 + M_2)$ -> FAIL
    - Find minimal M_1 in H_1 such that $(M_1 + M_2)$ -> FAIL
    - All elements in M_1 are necessary for FAIL (along with all elements from M_2)
  - Then all elements in $(M_1 + M_2)$ **are necessary** for FAIL
  - This is also minimal

QUIZ: Interference

- Mozilla Firefox Bug: crash occurs due to <select> in input <select size=5>, with each character being an input element. In how many iterations would delta debugging detect MIR?

(a) 1 – 5
(b) 5 – 10
(c) 10 – 15
(d) 15 – 20
(e) none of the above

QUIZ: Interference

- Mozilla Firefox Bug: crash occurs due to <select> in input <select size=5>, with each character being an input element. In how many iterations would delta debugging detect MIR?

Configuration	**Result**
<select size=5>	FAIL
<select	PASS
size=5>	PASS

Interference

(a) 1 – 5
(b) 5 – 10
(c) 10 – 15
(d) 15 – 20
(e) none of the above

QUIZ: Interference

- Mozilla Firefox Bug: crash occurs due to <select> in input <select size=5>, with each character being an input element. In how many iterations would delta debugging detect MIR?

	Configuration	**Result**
	<select size=5>	FAIL
1	<select	PASS
2	size=5>	PASS
3	<select siz	PASS
4	<select e=5>	FAIL
5	<select e=	PASS
6	<select 5>	FAIL
7	<select 5	PASS
8	<select >	FAIL

(a) 1 – 5
(b) 5 – 10
(c) 10 – 15
(d) 15 – 20
(e) none of the above

QUIZ: Interference

- Mozilla Firefox Bug: crash occurs due to <select> in input <select size=5>, with each character being an input element. In how many iterations would delta debugging detect MIR?

	Configuration	Result
	<select size=5>	FAIL
1	<select	PASS
2	size=5>	PASS
3	<select siz	PASS
4	<select e=5>	FAIL
5	<select e=	PASS
6	<select 5>	FAIL
7	<select 5	PASS
8	<select >	FAIL
9	<sel >	PASS
10	ect >	PASS
11	<selec >	PASS
12	<sel t >	PASS
13	<select >	FAIL
14	<sele t >	PASS
15	<sel ct >	PASS
16	<select >	FAIL
17	<se ct >	PASS
18	lect >	PASS
19	<sel ct >	PASS
20	<se ect >	PASS
21	<select >	FAIL
.		
.		
.		

(a) 1 – 5
(b) 5 – 10
(c) 10 – 15
(d) 15 – 20
(e) none of the above

Delta Debugging Algorithm

- Invariant: Input *I* along with some elements in $i_1,\ldots,i_n$ -> FAIL and *I* by itself does not cause FAIL
- Find: smallest subset of $\{i_1, \ldots, i_n\}$ that along with *I* will cause FAIL

$DD(I, \{i_1, \ldots, i_n\})$ =

if n == 1 return $\{i_1\}$

let $H_1 = I + \{i_1, \ldots, i_{n/2}\}$

let $H_2 = I + \{i_{n/2+1}, \ldots, i_n\}$

if H_1 -> FAIL

return $DD(I, \{i_1, \ldots, i_{n/2}\})$

else if H_2 -> FAIL

return $DD(I, \{i_{n/2+1}, \ldots, i_n\})$

else

let $M_2 = DD(H_1, \{i_{n/2+1}, \ldots, i_n\})$

Does invariant hold true?

Interference

Find MRI from second half using DD

Does the invariant hold true?

Delta Debugging Algorithm

- Invariant: Input *I* along with some elements in $i_1,\ldots,i_n$ -> FAIL and *I* by itself does not cause FAIL
- Find: smallest subset of $\{i_1, \ldots, i_n\}$ that along with *I* will cause FAIL

```
DD(I, {i_1, ..., i_n}) =
        if n == 1 return {i_1}
        let H_1 = I + {i_1, ..., i_{n/2}}
        let H_2 = I + {i_{n/2+1}, ..., i_n}
        if H_1 -> FAIL
          return DD(I, {i_1, ..., i_{n/2}})
      else if H_2 -> FAIL
         return DD(I, {i_{n/2+1}, ..., i_n})
      else
          let M_2 = DD(H_1, {i_{n/2+1}, ..., i_n})
          let M_1 = DD(M_2, {i_1, ..., i_{n/2}})
          return M_1 + M_2
```

Interference

Can you find the bug in this algorithm?

Delta Debugging Algorithm

- Invariant: Input *I* along with some elements in $i_1,\ldots,i_n$ -> FAIL and *I* by itself does not cause FAIL
- Find: smallest subset of $\{i_1, \ldots, i_n\}$ that along with *I* will cause FAIL

$DD(I, \{i_1, \ldots, i_n\})$ =

    if n == 1 return $\{i_1\}$
    let $H_1 = I + \{i_1, \ldots, i_{n/2}\}$
    let $H_2 = I + \{i_{n/2+1}, \ldots, i_n\}$
    if H_1 -> FAIL
        return $DD(I, \{i_1, \ldots, i_{n/2}\})$
    else if H_2 -> FAIL
        return $DD(I, \{i_{n/2+1}, \ldots, i_n\})$
    else
        let $M_2 = DD(H_1, \{i_{n/2+1}, \ldots, i_n\})$
        let $M_1 = DD(\cancel{M_2}\ H_2, \{i_1, \ldots, i_{n/2}\})$
        return $M_1 + M_2$

Interference

Can you find the bug in this algorithm?

Run-time Analysis

- Worst case:
  - We remove 1 element per iteration after trying every other element
  - Work is potentially n + (n-1) + (n-2) + …
  - $O(N^2)$
- Sub-diving sets until each set is of size 1 improves efficiency
- For single failure, converges in O(N log N)

Case Study: GNU C Compiler

- This program (bug.c) crashes GCC version 2.95.2 when optimizations are 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 R_p = running GCC on empty input
    - Test R_f = running GCC on bug.c

```
#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)
}
```

Case Study: GNU C Compiler

The Test Procedure provided to DD:

- Create an appropriate subset of bug.c
- Feed subset to GCC
- Return FAIL if GCC crashes, PASS otherwise

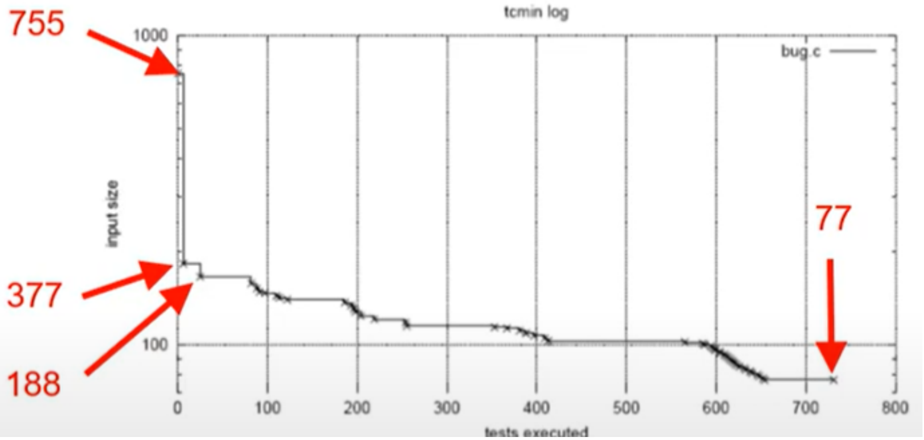
```
#define SIZE 20
double mult(double z[], int n) {
   int i, j;
   i = 0;
   for (j = 0; j < n; j++) {
                i = i + j + 1;
                z[i] = z[i] * (z[0] + 1.0);
   }
   return z[n];
}
void copy(double to[], double from[], int count) {
    int n = (count + 7) / 8;
    switch (count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

Case Study: GNU C Compiler

The Test Procedure provided to DD:

- Create an appropriate subset of bug.c
- Feed subset to GCC
- Return FAIL if GCC crashes, PASS otherwise

```
#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)
}
```

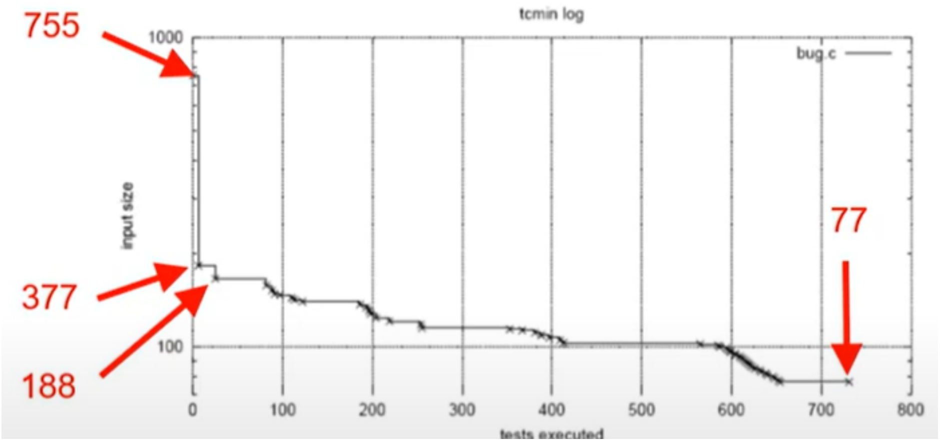


In the first two tests, size reduces from 755 to 188 characters

Case Study: GNU C Compiler

The Test Procedure provided to DD:

- Create an appropriate subset of bug.c
- Feed subset to GCC
- Return FAIL if GCC crashes, PASS otherwise




```
#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];
}
```

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

Takes several iterations to finally come up with a 77-character program that still causes GCC to crash

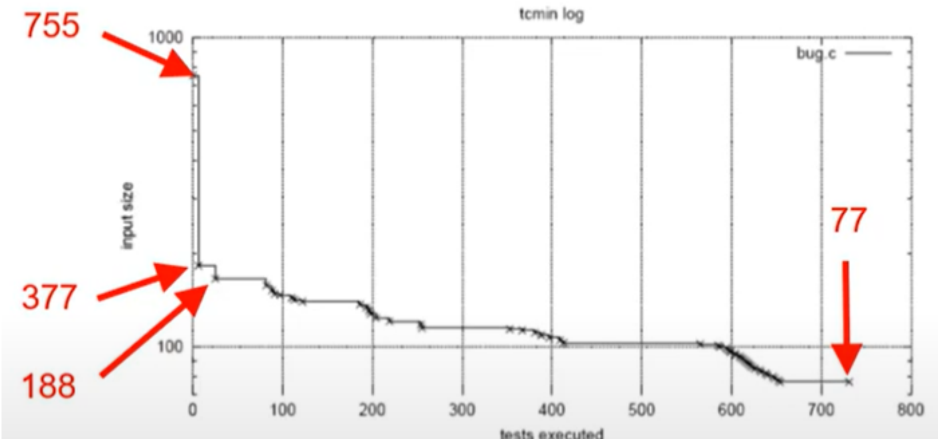
Case Study: GNU C Compiler

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

```
#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];
}
```

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

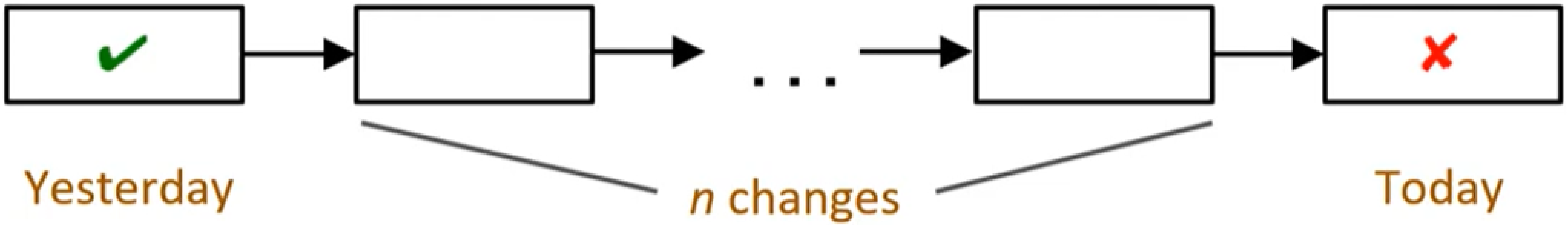
File bug report with a single line program and telling developers that it occurs when using optimization (-o) during compilation



Case Study: Minimizing Fuzzed Inputs

- Random Testing (a.k.a Fuzzing): feed program with randomly generated inputs 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 input that causes the same failure

Case Study: Isolating Failure-Inducing Changes



- Yesterday, my program was working. Today, it does not. Why?
  - The new release of GDB (v4.17) changed 178,000 lines
  - No longer integrated properly with DDD (data display debugger, a GUI of GDB)
  - How do we isolate the change from 178,000 lines that caused the failure?

Implement delta debugging algorithm with passing input being yesterday's code and failure input being today's code and the test procedure checking if the given subset of the code causes failure

QUIZ: Delta Debugging

Which of the following statements is True about delta debugging?

- ✗ It is fully automatic
- ✓ Finds 1-minimal instead of global minimum test case due to performance
  - 1-miminal: Removing any change from a set causes the failure to go away
  - Global minimum: Smallest set of changes that will make the program fail
- ✗ Finds the smallest failing subset of a failing input in polynomial time
- ✓ May find a different sizes subset of a failing input depending on the order in which it tests different input partitions.
- ✗ Is also effective at reducing non-deterministically failing inputs

What Have We Learnt?

- Delta Debugging is a technique, not a tool
- Bad news:
  - Must be re-implemented for each scenario and system to exploit knowledge about changes (line, character, commits, etc.)
- Good news
  - Relatively simple algorithm, big payoff
  - It is worth re-implementing

- You will see its use in your In-class exercise-2 and Homework-2!

Reminders

- Homework 1 is due next week! (May 1)
  - Use lecture notes and readings on software design patterns
  - Open-ended, so don’t wait until last minute!
- Paper presentations will start in two weeks (May 8)
  - Reading a research paper can take 5-6 hours or even a day so don’t wait until last minute as it will be overwhelming
  - You need to read ALL 6 papers scheduled to be presented during class
  - For the paper you selected, you will present it during class time
  - For ALL the 6 papers, you will submit their reviews