IS893: Advanced Software Security

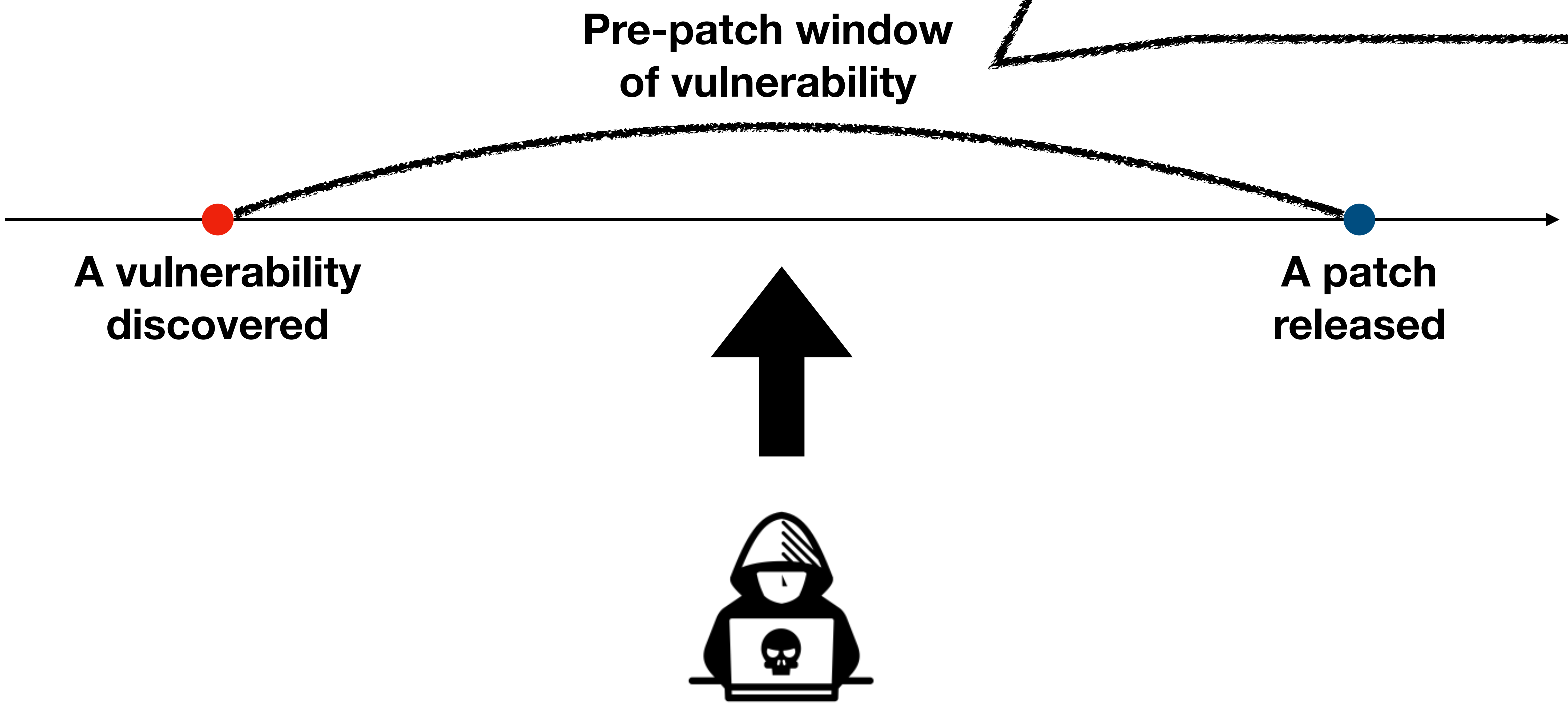
4. Cause Reduction

Kihong Heo

KAIST

Why Debugging Important?

For large and complicated SW,
this delay may become significant
(*1.5 months on average)



Pre-patch window
of vulnerability

A vulnerability
discovered

A patch
released

*Huang et al., Talos: Neutralizing Vulnerabilities with Security Workarounds for Rapid Response, *S&P* 2016

Cause Reduction

- Once a failing input is discovered, we must find out
  - why the failure occurred
  - how to fix it
- However, is the whole information relevant to the problem?
  - E.g., 1KLOC, long sequence of function calls, large inputs

> *"Often people who encounter a bug spend a lot of time investigating* ***which changes to the input*** *file will make the bug go away and* ***which changes*** *will not affect it."*
>
> — Richard Stallman, *Using and Porting GNU CC*

Simplified Test Case

- Ease of communication: succinctly express the problem
- Easier of debugging: result in smaller states and shorter executions
- Remove duplicates: one test case subsumes many minor variants

"When you have two competing theories which make exactly the same predictions, the one that is ***simpler is the better****."*

—Occam's razor

Crashing 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>
```

Printing the HTML file causes Mozilla’s Netscape to crash in 1999

Simplified Crashing Input

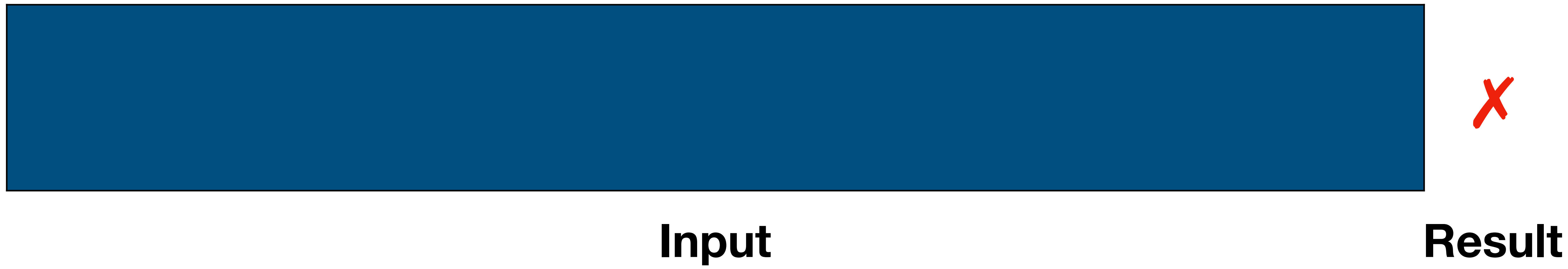
```
<SELECT>
```

How to automatically simplify the input?

Printing the HTML file still causes Mozilla's Netscape to crash in 1999

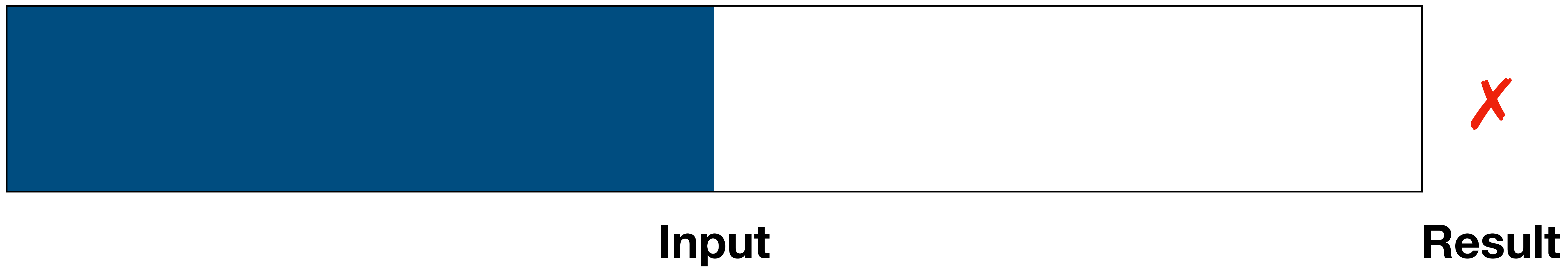
Naive Algorithm

- An algorithm to reduce failure-inducing inputs based on the binary search
    - Remove half the input and see if it still crashes
    - If so, recurse on the reduced half
    - If not, try the other half
- Arbitrary unit: line-level, character-level, etc



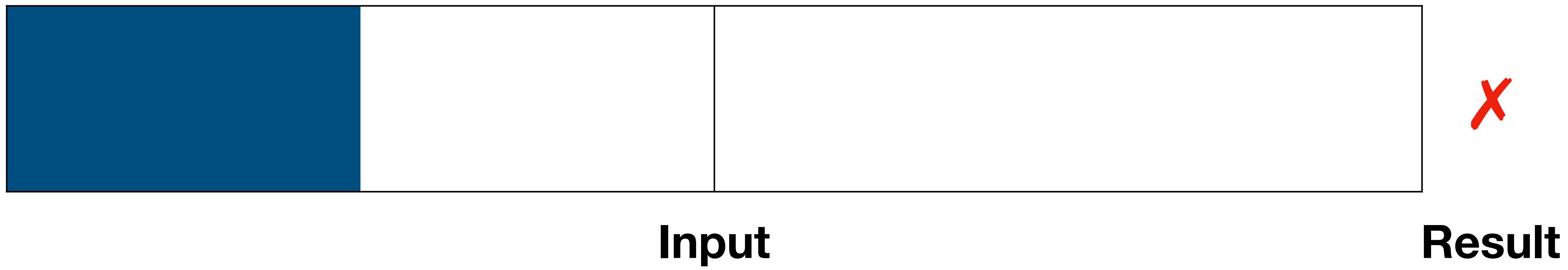
Naive Algorithm

- An algorithm to reduce failure-inducing inputs based on the binary search
  - Remove half the input and see if it still crashes
  - If so, recurse on the reduced half
  - If not, try the other half
- Arbitrary unit: line-level, character-level, etc



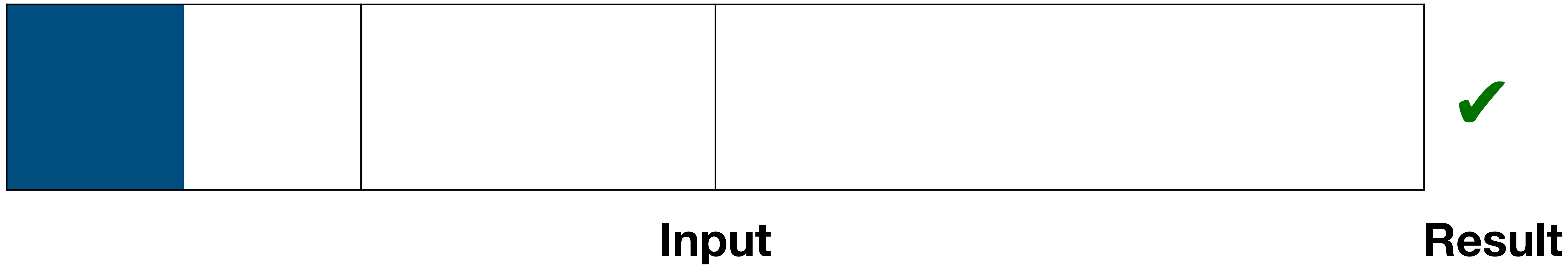
Naive Algorithm

- An algorithm to reduce failure-inducing inputs based on the binary search
  - Remove half the input and see if it still crashes
  - If so, recurse on the reduced half
  - If not, try the other half
- Arbitrary unit: line-level, character-level, etc



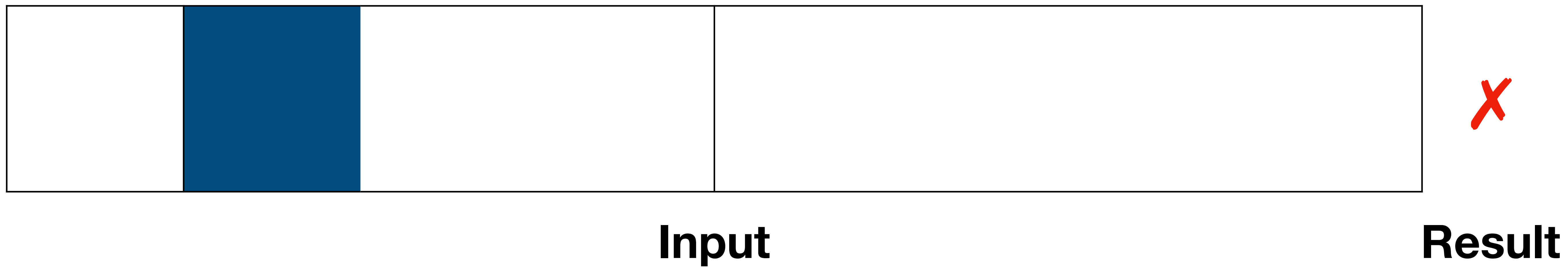
Naive Algorithm

- An algorithm to reduce failure-inducing inputs based on the binary search
  - Remove half the input and see if it still crashes
  - If so, recurse on the reduced half
  - If not, try the other half
- Arbitrary unit: line-level, character-level, etc



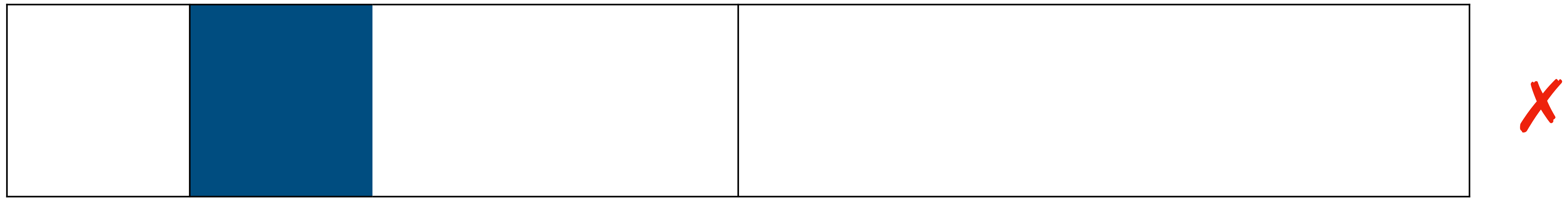
Naive Algorithm

- An algorithm to reduce failure-inducing inputs based on the binary search
  - Remove half the input and see if it still crashes
  - If so, recurse on the reduced half
  - If not, try the other half
- Arbitrary unit: line-level, character-level, etc



Naive Algorithm

- An algorithm to reduce failure-inducing inputs based on the binary search
  - Remove half the input and see if it still crashes
  - If so, recurse on the reduced half
  - If not, try the other half
- Arbitrary unit: line-level, character-level, etc



...

Example

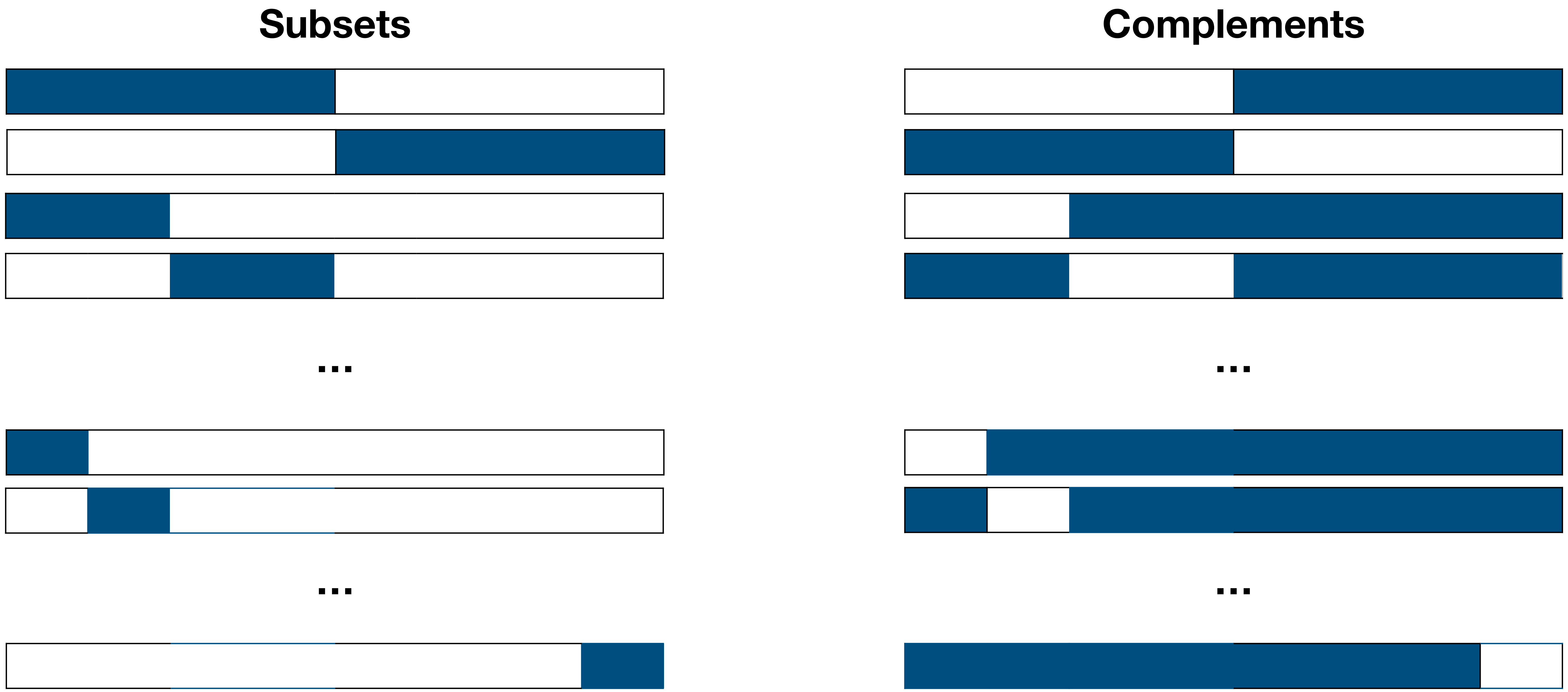
How to proceed?

```
1 <SELECT NAME="priority" MULTIPLE SIZE=7> ✗
2 <SELECT NAME="priority" MULTIPLE SIZE=7> ✔
3 <SELECT NAME="priority" MULTIPLE SIZE=7> ✔
4 <SELECT NAME="priority" MULTIPLE SIZE=7> ✔
```

Finer granularity!

Delta Debugging

- More generalized version of the naive algorithm
- Increase the granularity gradually until the limit



Example

How to proceed?

Finer granularity!

1 **<SELECT NAME="priority" MULTIPLE SIZE=7>** ✗

2 <SELECT NAME="priori**ty" MULTIPLE SIZE=7>** ✓

3 **<SELECT NAME="priori**ty" MULTIPLE SIZE=7> ✓

4 <SELECT NA**ME="priority" MULTIPLE SIZE=7>** ✓

5 **<SELECT NA**ME="priori**ty" MULTIPLE SIZE=7>** ✗

6 **<SELECT NA**ME="priority" MULTIP**LE SIZE=7>** ✗

7 **<SELECT NA**ME="priority" MULTIPLE SIZE=7> ✓

8 <SELE**CT NA**ME="priority" MULTIP**LE SIZE=7>** ✓

9 **<SELE**CT NAME="priority" MULTIP**LE SIZE=7>** ✓

10 **<SELECT NA**ME="priority" MULTIPLE SI**ZE=7>** ✗

11 **<SELECT NA**ME="priority" MULTIPLE SIZE=7> ✓

12 <S**ELECT NA**ME="priority" MULTIPLE SI**ZE=7>** ✓

13 **<S**EL**ECT NA**ME="priority" MULTIPLE SI**ZE=7>** ✓

14 **<SEL**EC**T NA**ME="priority" MULTIPLE SI**ZE=7>** ✓

15 **<SELEC**T **NA**ME="priority" MULTIPLE SI**ZE=7>** ✓

16 **<SELECT** NAME="priority" MULTIPLE SI**ZE=7>** ✗

17 **<SELECT** NAME="priority" MULTIPLE SIZE**=7>** ✗

18 **<SELECT** NAME="priority" MULTIPLE SIZE=7**>** ✗

19 <**SELECT** NAME="priority" MULTIPLE SIZE=7**>** ✓

20 **<**S**ELECT** NAME="priority" MULTIPLE SIZE=7**>** ✓

21 **<S**E**LECT** NAME="priority" MULTIPLE SIZE=7**>** ✓

22 **<SE**L**ECT** NAME="priority" MULTIPLE SIZE=7**>** ✓

23 **<SEL**E**CT** NAME="priority" MULTIPLE SIZE=7**>** ✓

24 **<SELE**C**T** NAME="priority" MULTIPLE SIZE=7**>** ✓

25 **<SELEC**T NAME="priority" MULTIPLE SIZE=7**>** ✓

26 **<SELECT** NAME="priority" MULTIPLE SIZE=7**>** ✗

Unit = 1

No more candidates

The original 896-line HTML input is reduced to the minimal failing test case <SELECT> after 57 tests

Minimality

- Goal:

> Given a failing test case c_F, find the smallest test case c s. t. $test(c) = \text{F}$

- A test case c is called the **global** minimum if

$$\forall c' \subseteq c_F . |c'| < |c| \implies test(c') \neq \text{F}$$

- Caveat: finding the global minimum may require exponential # of tests

1-minimality

- A test case $c \subseteq c_F$ is called a **local minimum** if

$$\forall c' \subset c. test(c') \neq \mathsf{F}$$

- A test case $c \subseteq c_F$ is called ***n*-minimal** if

$$\forall c' \subset c. |c| - |c'| \leq n \implies test(c') \neq \mathsf{F}$$

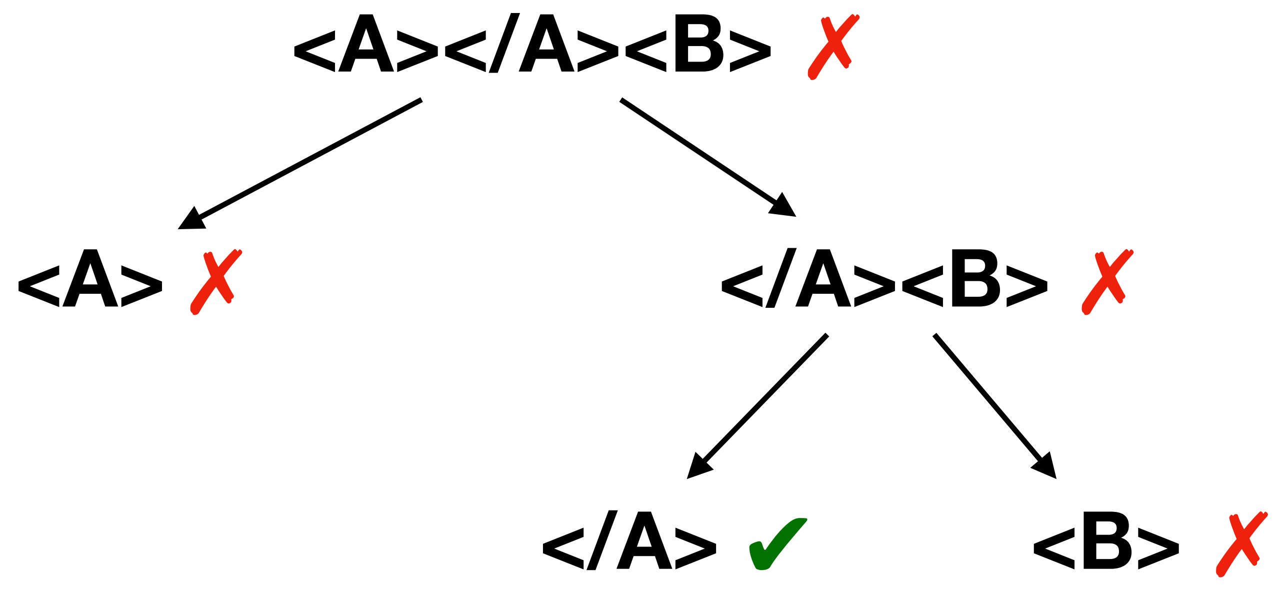
- The delta debugging finds a 1-minimal solution

“Removing any single part from the input causes the failure to go away”

Assumptions on Failures

- **Monotonicity**: the super input of a failure input always induces the failure
  - With a failure input, one cannot make it disappear by adding more contents
  - For example, property = failure if there exists an open but unclosed tag

This is an 1-minimal solution. But does this represent the actual cause?



<A></A><B> ✗

<A> ✗

</A><B> ✗

</A> ✓

<B> ✗

- **Determinism**: the same input always produce the same result

Minimization Algorithm

Let *test* and $c_{\text{✘}}$ be given such that $test(\emptyset) = \text{✔} \wedge test(c_{\text{✘}}) = \text{✘}$ hold.
The goal is to find $c'_{\text{✘}} = ddmin(c_{\text{✘}})$ such that $c'_{\text{✘}} \subseteq c_{\text{✘}}$, $test(c'_{\text{✘}}) = \text{✘}$, and $c'_{\text{✘}}$ is 1-minimal.
The *minimizing Delta Debugging algorithm* $ddmin(c)$ is

$$ddmin(c_{\text{✘}}) = ddmin_2(c_{\text{✘}}, 2) \quad \text{where}$$

$$ddmin_2(c'_{\text{✘}}, n) = \begin{cases} ddmin_2(\Delta_i, 2) & \text{if } \exists i \in \{1, \ldots, n\} \cdot test(\Delta_i) = \text{✘ (“reduce to subset”)} \\ ddmin_2\big(\nabla_i, \max(n-1, 2)\big) & \text{else if } \exists i \in \{1, \ldots, n\} \cdot test(\nabla_i) = \text{✘ (“reduce to complement”)} \\ ddmin_2\big(c'_{\text{✘}}, \min(|c'_{\text{✘}}|, 2n)\big) & \text{else if } n < |c'_{\text{✘}}| \text{ (“increase granularity”)} \\ c'_{\text{✘}} & \text{otherwise (“done”).} \end{cases}$$

where $\nabla_i = c'_{\text{✘}} - \Delta_i$, $c'_{\text{✘}} = \Delta_1 \cup \Delta_2 \cup \cdots \cup \Delta_n$, all Δ_i are pairwise disjoint, and $\forall \Delta_i \cdot |\Delta_i| \approx |c'_{\text{✘}}|/n$ holds.
The recursion invariant (and thus precondition) for $ddmin_2$ is $test(c'_{\text{✘}}) = \text{✘} \wedge n \leq |c'_{\text{✘}}|$.

Time Complexity

- In the worst case,
  - First, every test has an unresolved result until the maximum granularity: doubled number of subsets for each granularity

$$2 + 4 + 8 + \cdots + 2|c_F| = 2|c_F| + |c_F| + \frac{|c_F|}{2} + \frac{|c_F|}{4} + \cdots = 4|c_F|$$

  - Then, testing only the last complement results in a failure:

$$2(|c_F| - 1) + 2(|c_F| - 2) + \cdots + 2 = 2 + 4 + 6 + \cdots + 2(|c_F| - 1) = |c_F|(|c_F| - 1)$$

Polynomial time: $O(|c_F|^2)$

Example: GCC

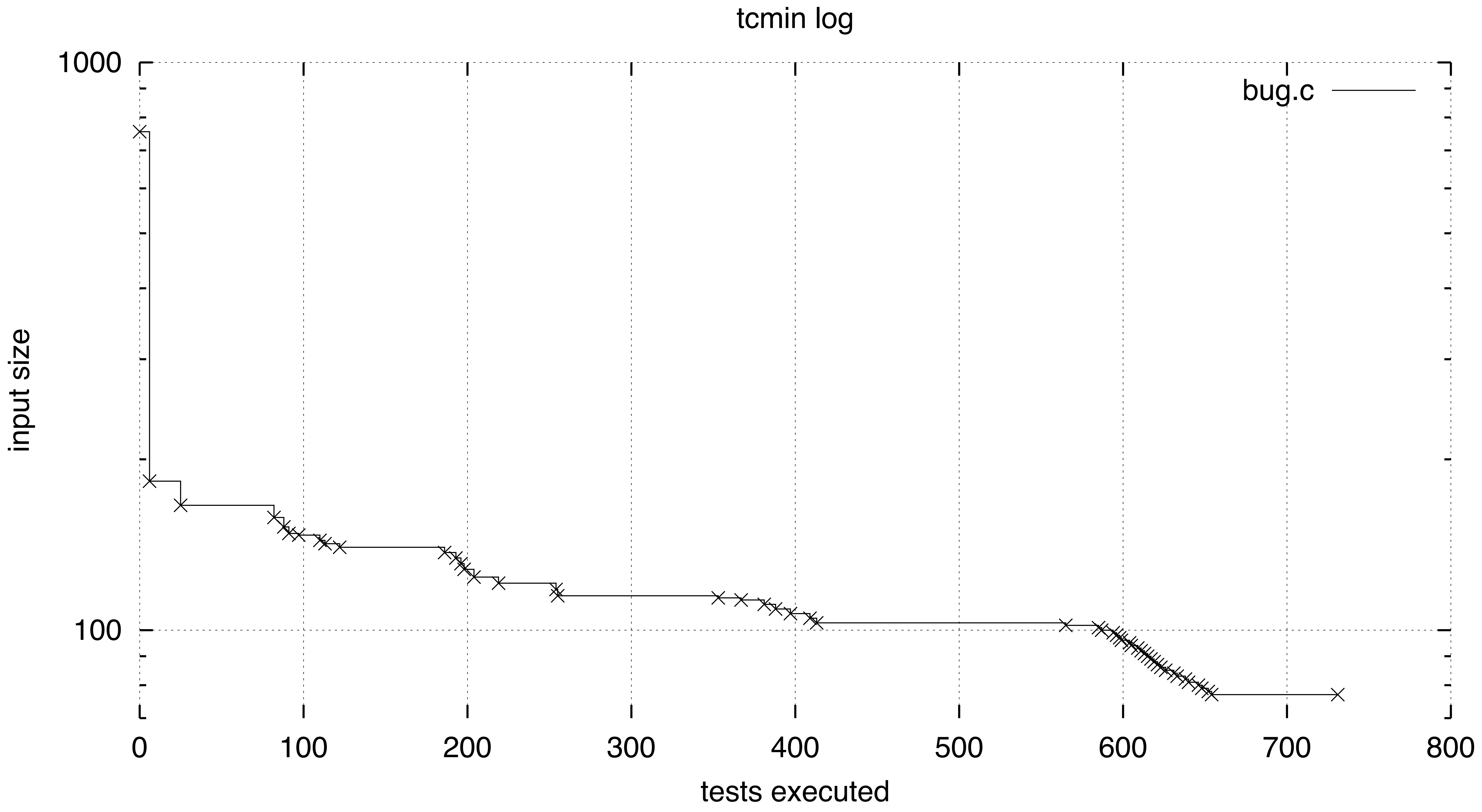
```
#define SIZE 20
double mult(double z[], int n) {
  int i , j ;
  i = 0;
  for (j = 0; j < n; j++) {
    i = i + j + 1;
    z[i] = z[i] *(z[0]+1.0); return z[n];
  }
  return z[n];
}

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 crashes GCC 2.95.2 with “-O”
- Delta debugging minimizes the input (755 chars) to the following file (77 chars)

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

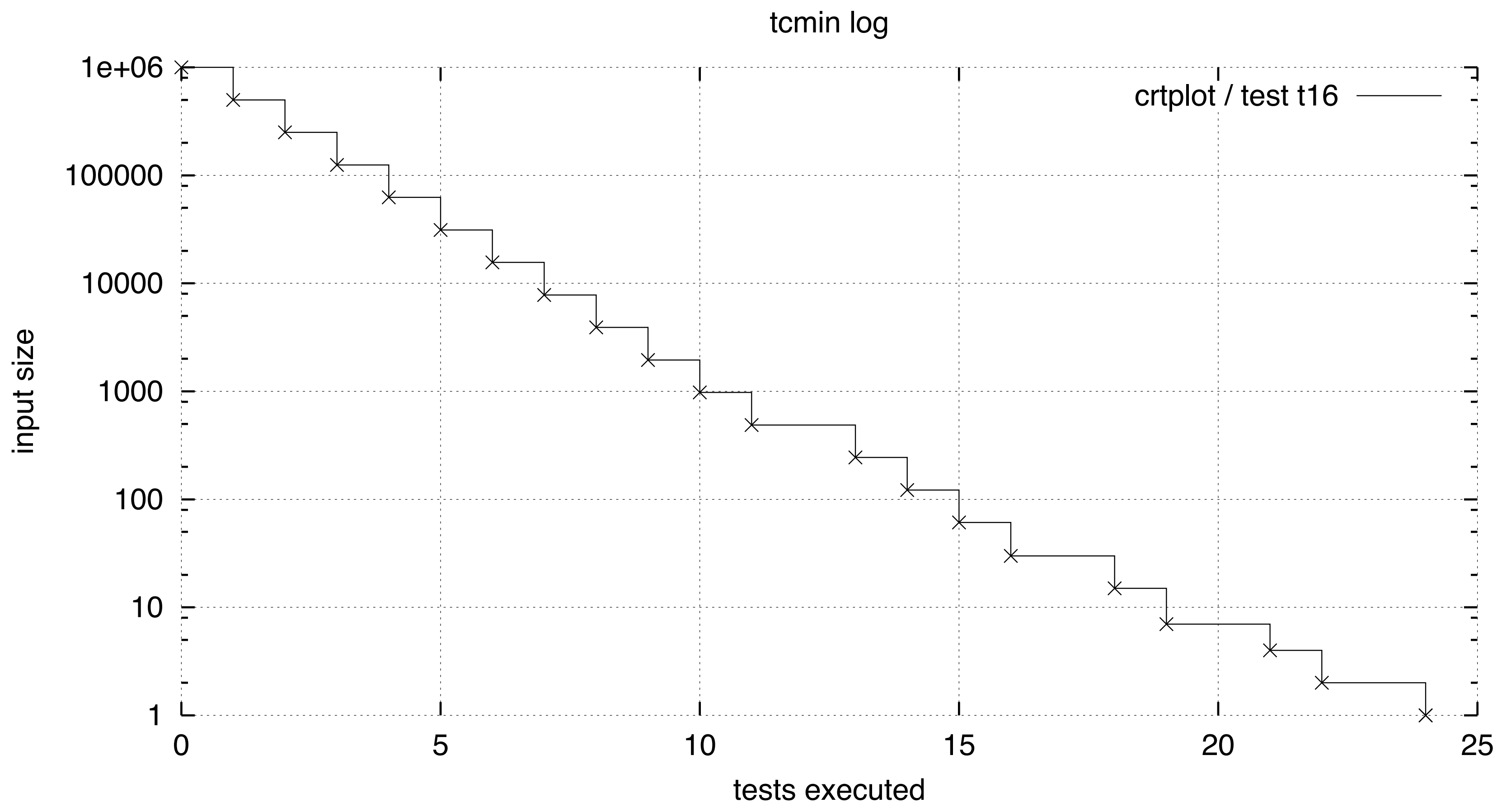
- This test case is 1-minimal:
  “removing any single char removes the failure”

Example: GCC


tcmin log
bug.c
input size
1000
100
tests executed
0
100
200
300
400
500
600
700
800

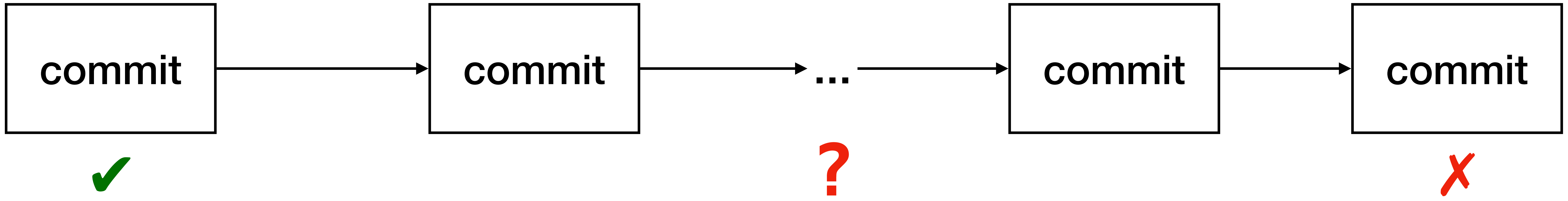
Example: Fuzzer

- Typically, failure-inducing inputs by fuzzers are large
  - E.g., a fuzz input comprising 10^6 characters that crashes CRTPLOT*
- Delta debugging is a great companion of fuzzing



*Barton Miller, et al., An Empirical Study of the Reliability of UNIX Utilities, *CACM*, 1990

Example: Git Bisect



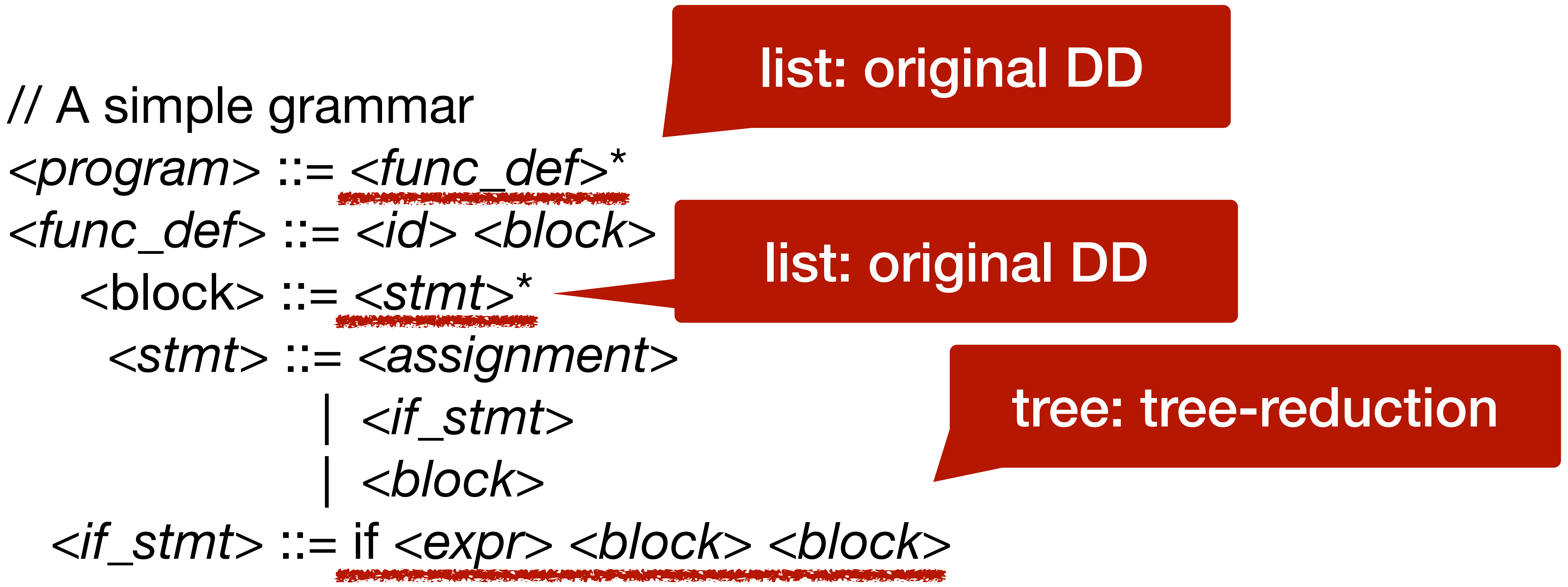
“Yesterday, my program worked. Today, it does not. Why?”

Improvement

- If the input is highly structured, DD becomes inefficient
    - E.g., data structures, source code
- Solution: hierarchical delta debugging
    - Reduce inputs with respect to the schema or grammar
    - Always produce syntactically correct candidates

Grammar-based Delta Debugging

- Delta debugging w.r.t a given **grammar** (i.e., rule out ill-formed programs)
- Idea: **Hierarchically** perform delta debugging + **tree-reduction** rules



```
// A simple grammar
<program> ::= <func_def>*
<func_def> ::= <id> <block>
    <block> ::= <stmt>*
     <stmt> ::= <assignment>
              | <if_stmt>
              | <block>
  <if_stmt> ::= if <expr> <block> <block>
```

list: original DD

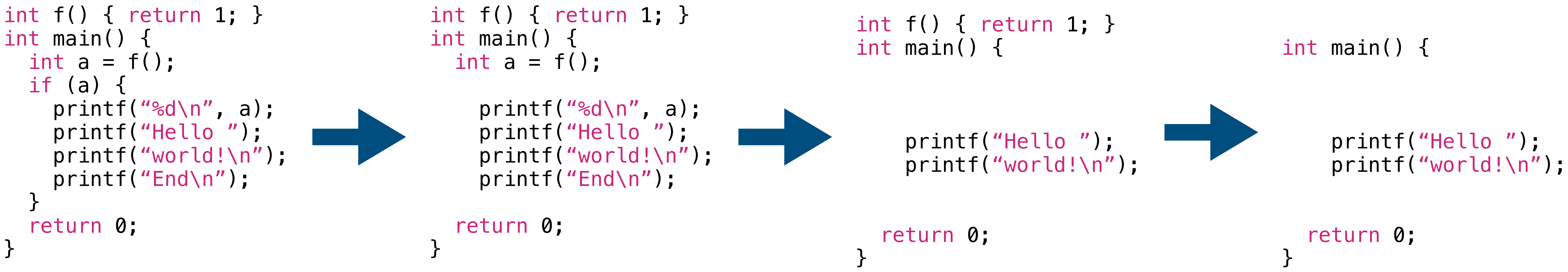
list: original DD

tree: tree-reduction

$$ddif(\texttt{if}\ E\ B_1\ B_2) = \begin{cases} B_1 & \text{if the replacement to } B_1 \text{ leads to success} \\ B_2 & \text{if the replacement to } B_2 \text{ leads to success} \\ \texttt{if}\ E\ B_1\ B_2 & \text{otherwise} \end{cases}$$

Example

- Property of interest: print a string including “Hello world!”



```
int f() { return 1; }
int main() {
  int a = f();
  if (a) {
    printf(“%d\n”, a);
    printf(“Hello ”);
    printf(“world!\n”);
    printf(“End\n”);
  }
  return 0;
}
```

```
int f() { return 1; }
int main() {
  int a = f();

    printf(“%d\n”, a);
    printf(“Hello ”);
    printf(“world!\n”);
    printf(“End\n”);

  return 0;
}
```

```
int f() { return 1; }
int main() {


    printf(“Hello ”);
    printf(“world!\n”);


  return 0;
}
```

```

int main() {


    printf(“Hello ”);
    printf(“world!\n”);


  return 0;
}
```

1. DD on the list of the functions
2. DD on the list of stmts of main
3. Reduction of the if-statement

4. DD on the list of stmts of the block

5. DD on the list of functions (again)
   (… until reaching a fixpoint)

Conclusion

- Cause reduction is necessary to understand a given SW error
- A well-known technique: Delta debugging
  - A simple and general approach to reduce failure-inducing inputs
  - Find a 1-minimal solution with quadratic time complexity
- Improvement: hierarchical delta debugging
  - Reduction w.r.t. the schema / grammar