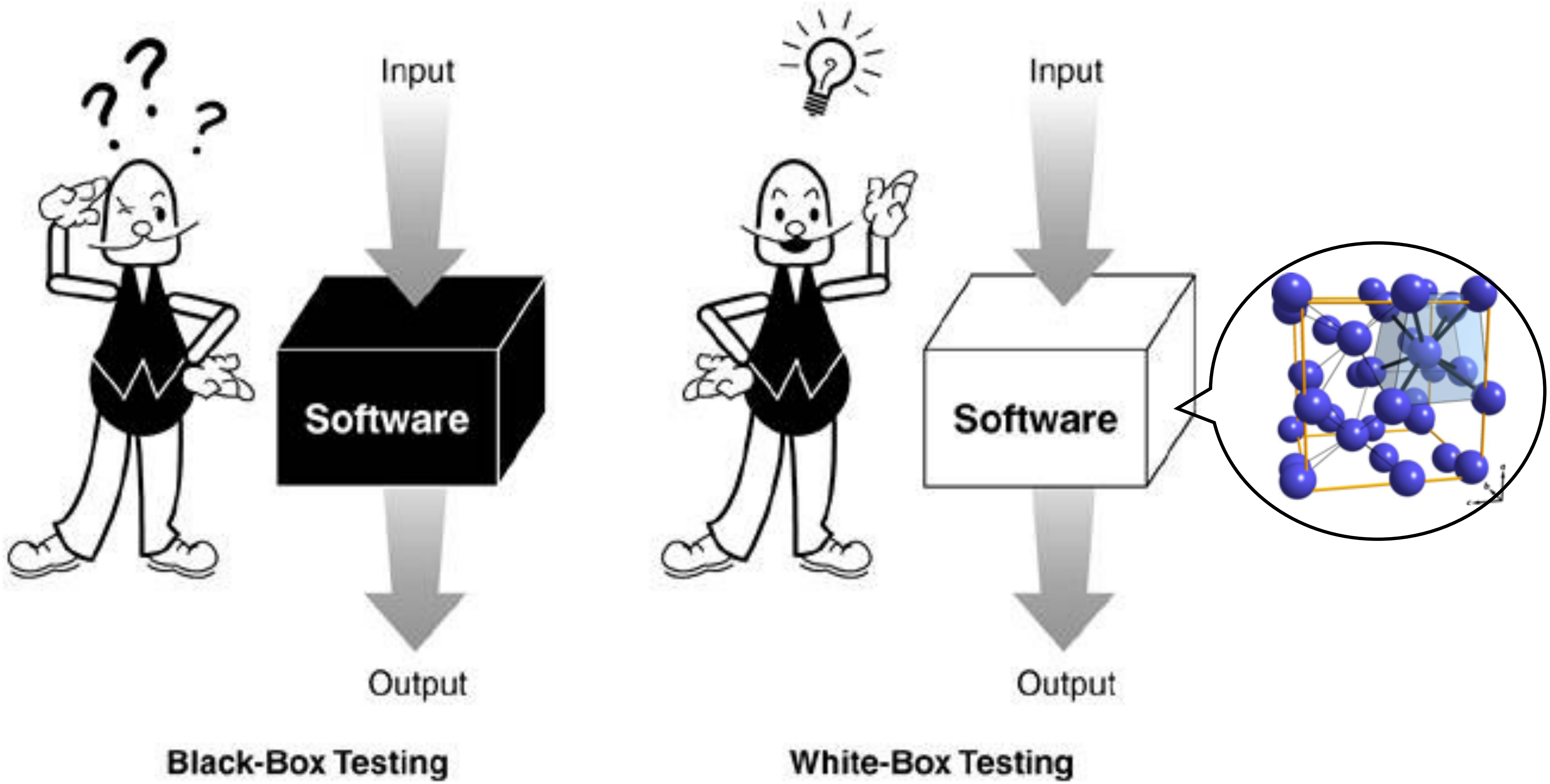
CE/CS/SE 3354
Software Engineering

Automated Test Generation

Testing Methodologies



Testing Methodologies

- Black-box (Functional) vs. White-box (Structural) testing
- Black-box testing: Generating test cases based on the functionality of the software
  - Internal structure of the program is hidden from the testing process
- White-box testing: Generating test cases based on the source-code structure of the program
  - Internal structure of the program is taken into account

Black-Box Testing (Functional Testing)

- Black-box testing:
  - Identify the main functions of software under test
  - Create test data which will check whether these functions are performed by the software
  - No consideration is given how the program performs these functions, program is treated as a black-box
- A systematic approach to functional testing: requirements based testing
  - Driving test cases automatically from a formal specification of the functional requirements

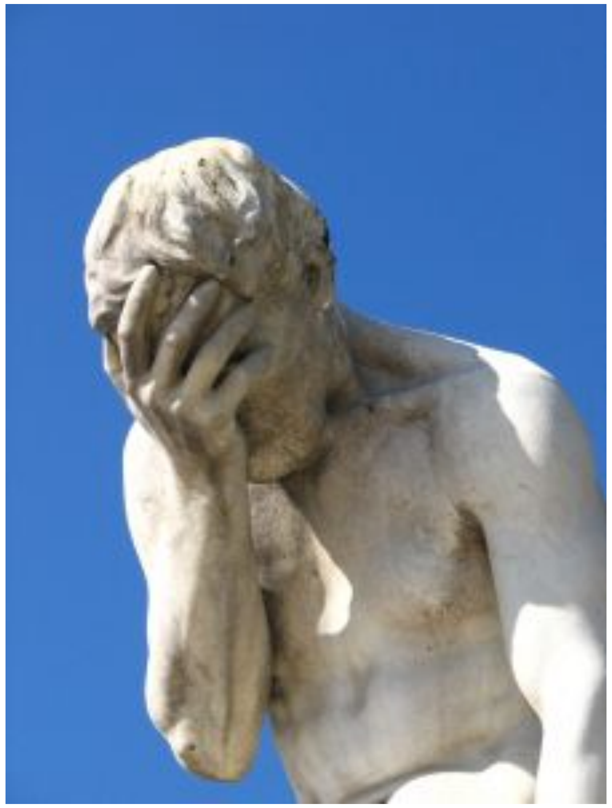
Exhaustive Testing is Hard

```
int max(int x, int y)
{
  if (x > y)
    return x;
  else
    return x;
}
```

- Number of possible test cases (assuming 32 bit integers)
  - $2^{32} * 2^{32} = 2^{64}$

Exhaustive Testing

- Assume that the input for the max procedure was an integer array of size n
  - Number of possible test cases: $(2^{32})^n=2^{32n}$
- Assume that the size of the input array is not bounded
  - Number of test cases: ∞
- The point is, naive exhaustive testing is pretty hopeless

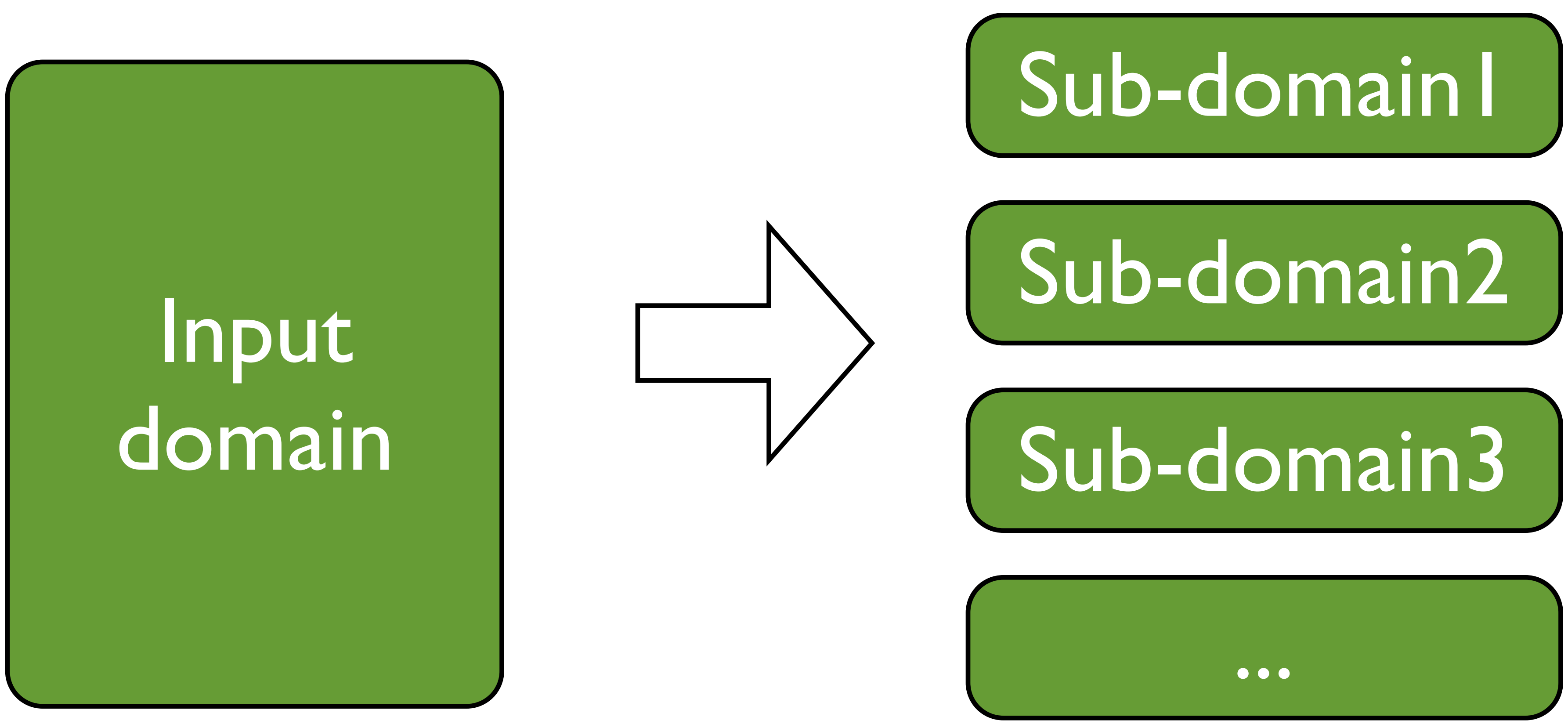


This Class

- Automated test generation
  - Black-box test generation
    - Exhaustive testing
    - Domain testing
    - Random testing and adaptive random testing
- Debugging
  - Fault localization
  - Delta debugging

Domain Testing

- Partition the input domain to equivalence classes
- For some requirements specifications it is possible to define equivalence classes in the input domain
- Choose one test case per equivalence class to test



Domain Testing: Example

- A factorial function specification:

If the input value n is less than 0 then error message must be printed. If 0<=n < 20, then the exact value n! must be printed. If 20<=n<=200, then an approximate value of n! must be printed in floating point format using some approximate numerical method. Finally, if n > 200, the input can be rejected by printing an error message.

- Possible equivalence classes: D1 = {n<0}, D2 = {0<=n<20}, D3 = {20<=n<=200}, D4 = {n > 200}
- Choose one test case per equivalence class to test

Equivalence Classes

- If the equivalence classes are disjoint, then they define a partition of the input domain
- If the equivalence classes are not disjoint, then we can try to minimize the number of test cases while choosing representatives from different classes
- Example: D1 = {x is even}, D2 = {x is odd}, D3 = {x<0}, D4={x>=0}
- Test set {x=48, x= –23} covers all the equivalence classes

Equivalence Classes

- If the equivalence classes are disjoint, then they define a partition of the input domain
- If the equivalence classes are not disjoint, then we can try to minimize the number of test cases while choosing representatives from different classes
- Example: D1 = {x is even}, D2 = {x is odd}, D3 = {x<0},

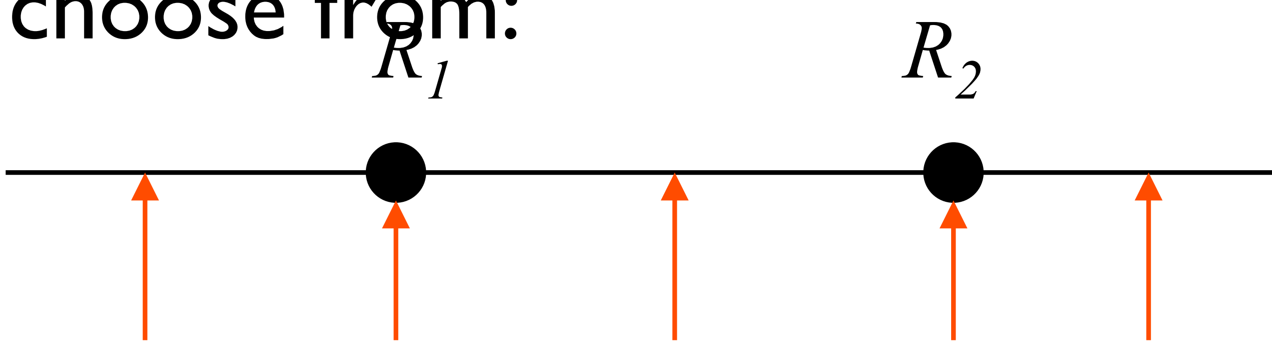
23}

On one extreme we can make each equivalence class have only one element which turns into *exhaustive testing*

The other extreme is choosing the whole input domain D as an equivalence class which would mean that we will use only one test case

Testing Boundary Conditions

- For each range [R1, R2] to test, choose from:
  - Values less than R1
  - Values equal to R1
  - Values > R1 but < R2
  - Values equal to R2
  - Values greater than R2



- For equality check select 2 values
  - 1) equal, 2) not equal
- For sets, lists select 2 cases
  - 1) empty, 2) not empty
- …

Testing Boundary Conditions

- For the factorial example, ranges for variable n are:
  - $[-\infty, 0], [0,20], [20,200], [200, \infty]$
- A possible test set:
  - {n = -5, n=0, n=11, n=20, n= 25, n=200, n= 3000}
- If we know the maximum and minimum values that n can take, we can also add those n=MIN, n=MAX to the test set

This Class

- Automated test generation
  - Black-box test generation

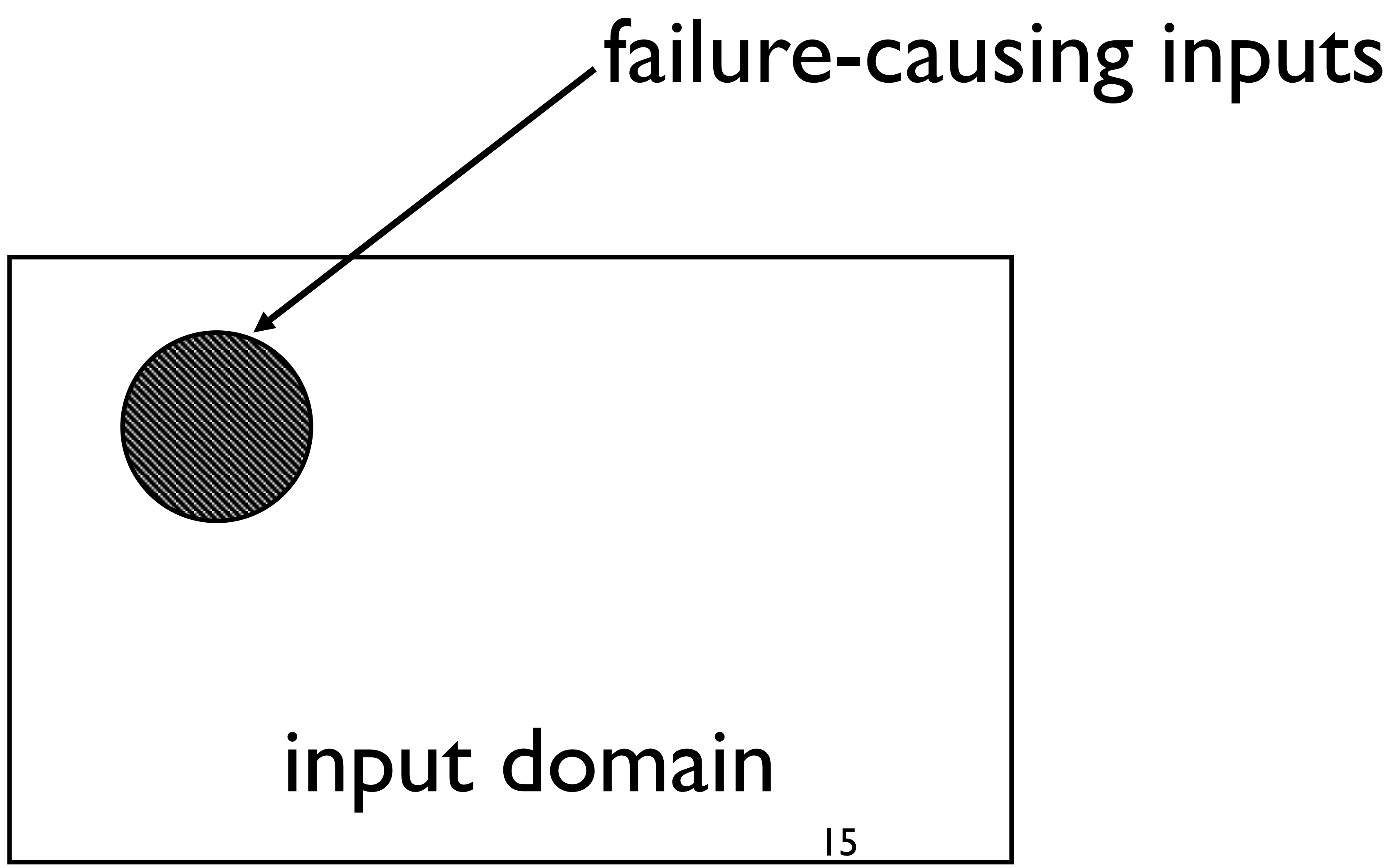
    Exhaustive testing

    Domain testing

    Random testing and adaptive random testing
- Debugging
  - Fault localization
  - Delta debugging

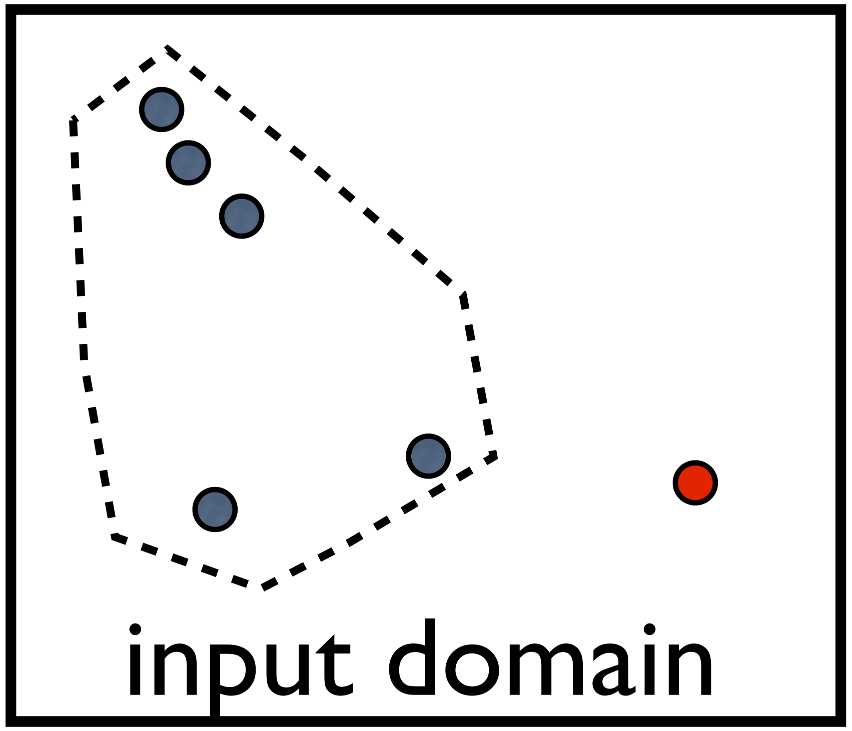
Basic Concepts

- Input domain – set of all possible inputs
- Failure-causing inputs – inputs that exhibit failure



Random Testing

- Random Testing
  - selects tests from the entire input domain randomly and independently



- Advantages:
  - Intuitively simple
  - Allows statistical estimation of the software's reliability
- Disadvantages:
  - No guide towards failure-causing inputs

How to improve random testing?

- Any common information or characteristics to all faulty programs?

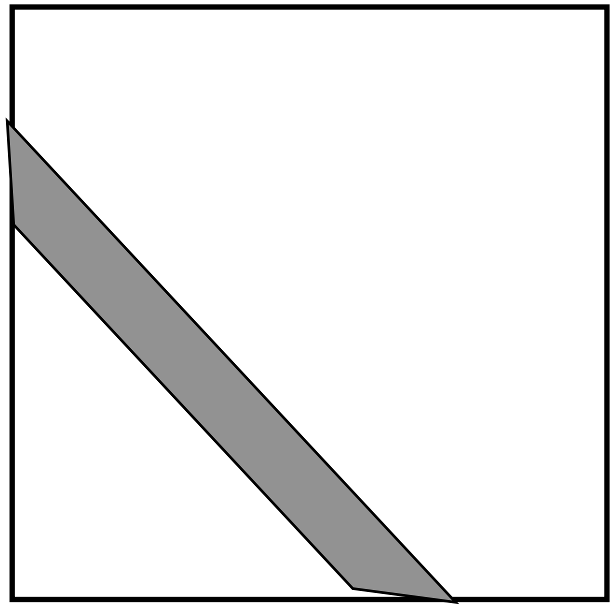
Failure-causing inputs

Patterns of Failure-Causing Inputs

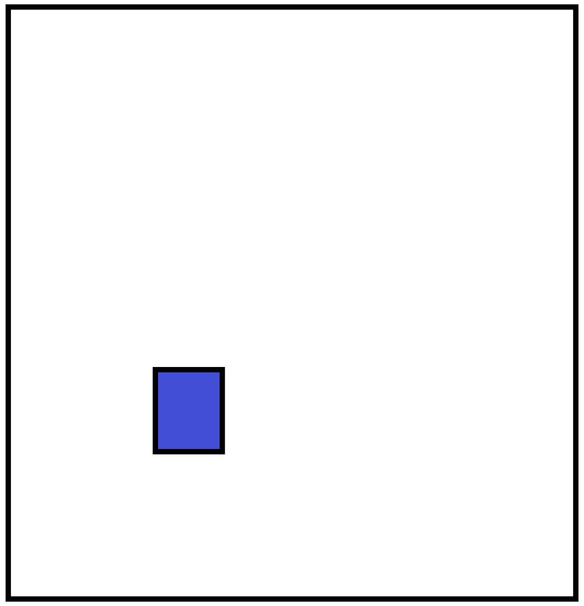
- Strip Pattern
- Block Pattern
- Point Pattern

Types of Failure Patterns

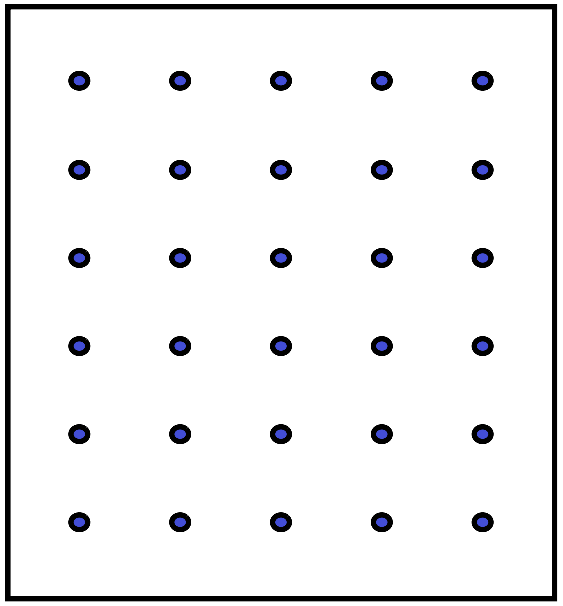
Strip Pattern



Block Pattern

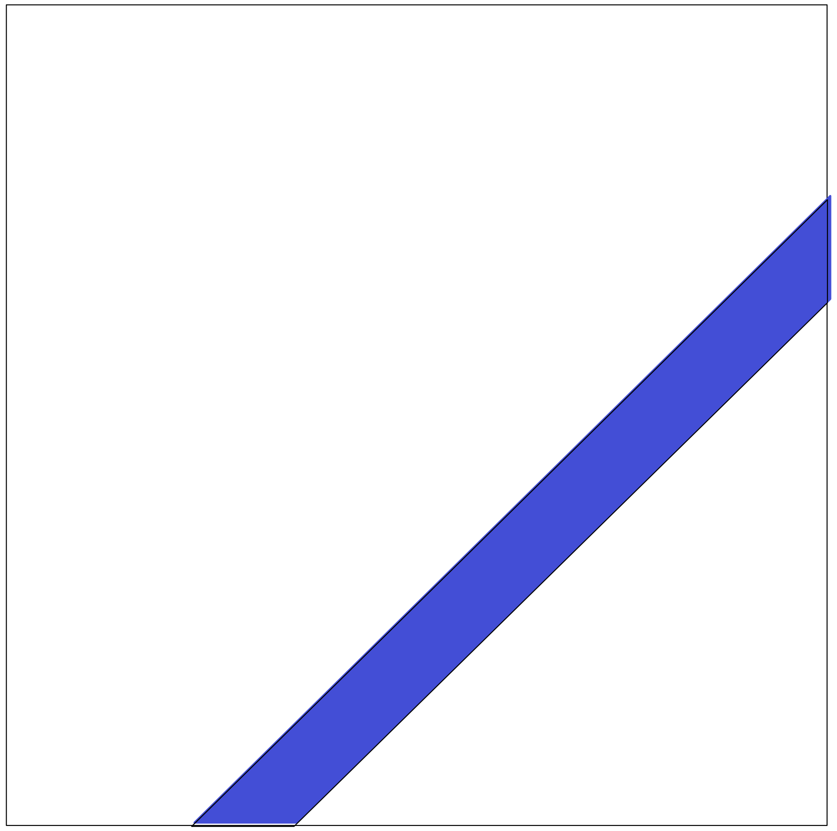


Point Pattern



Strip Pattern

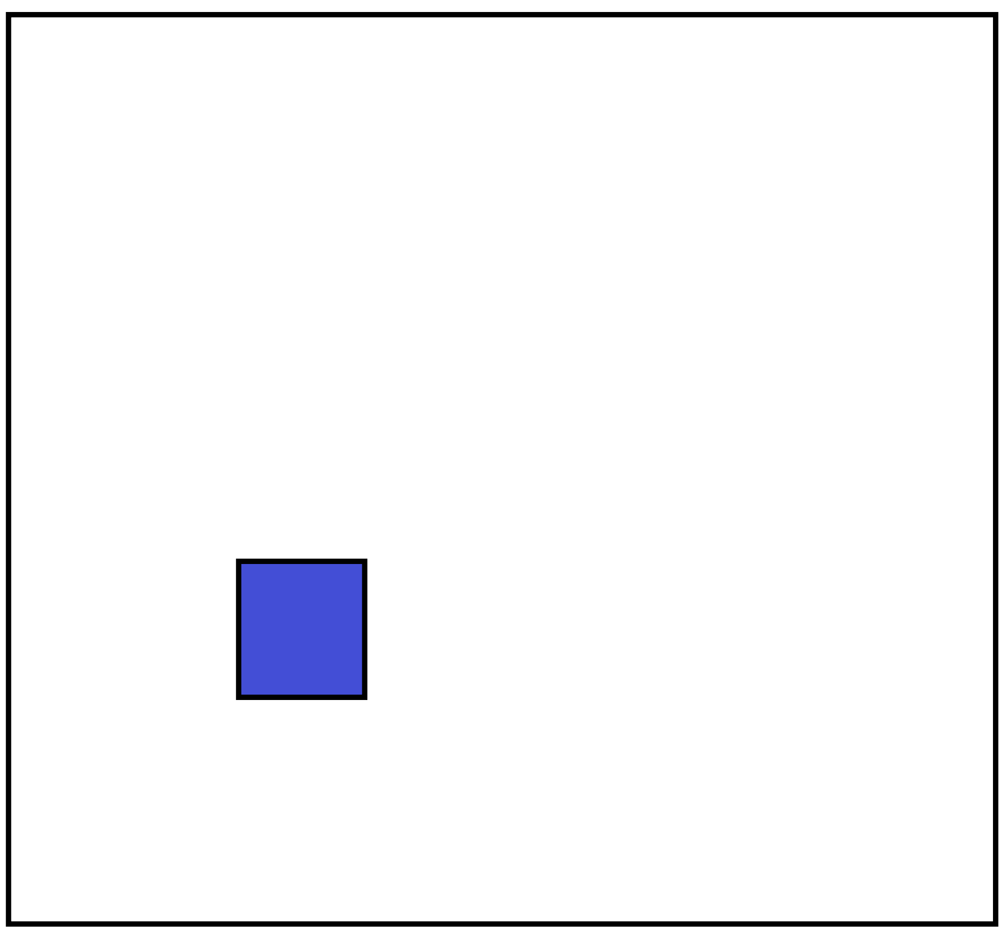
Two Dimensional Input Domain



```
If (2*x – y > 10)
/* the correct statement is
 If (2*x – y > 20) */
    z = x/2 *y;
else
    z := x*y;
```

Block Pattern

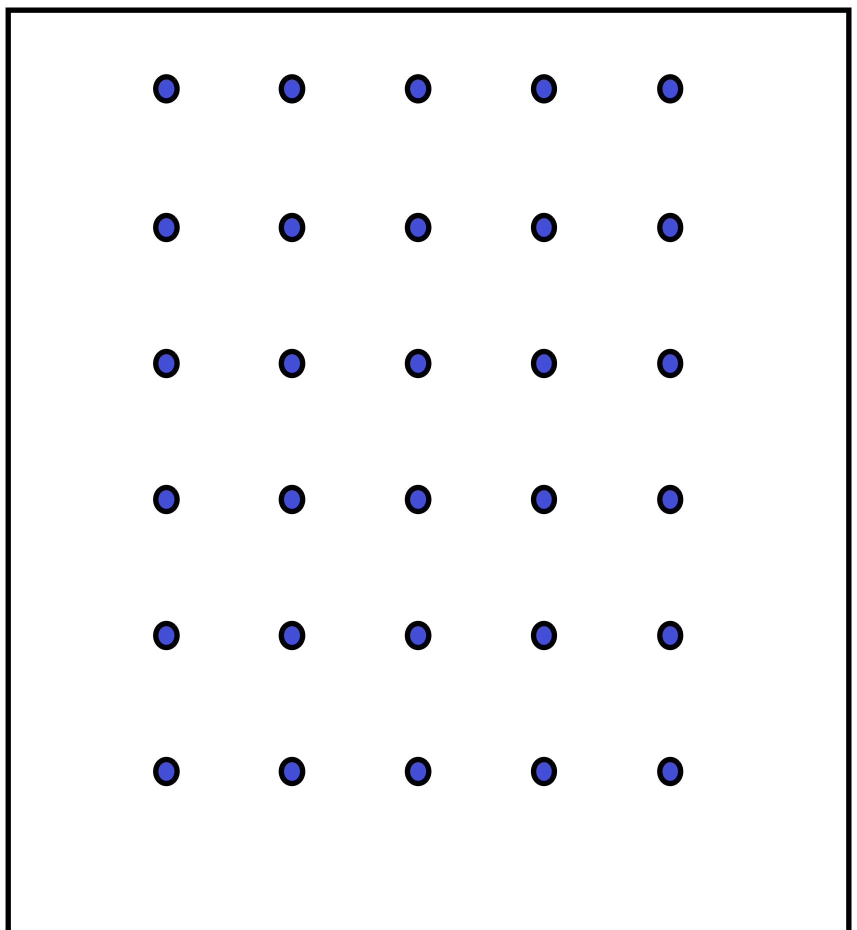
Two Dimensional Input Domain



```
If (x >= 4 and x <=6)
   and
   (y >= 4 and y <= 6)
    z := x + y;
/* the correct statement is
   z := x - y; */
else
    z := 100;
```

Point Pattern

Two Dimensional
input domain



```
If ((x mod 10) = 0) and
   ((y mod 10) = 0) and
   ( x > 2) then
    z:= f(x, y);
/* should be
   z:=  g(x,y); */
else
    z := f(x, y);
```

Which pattern occurs more frequently?

block and strip patterns

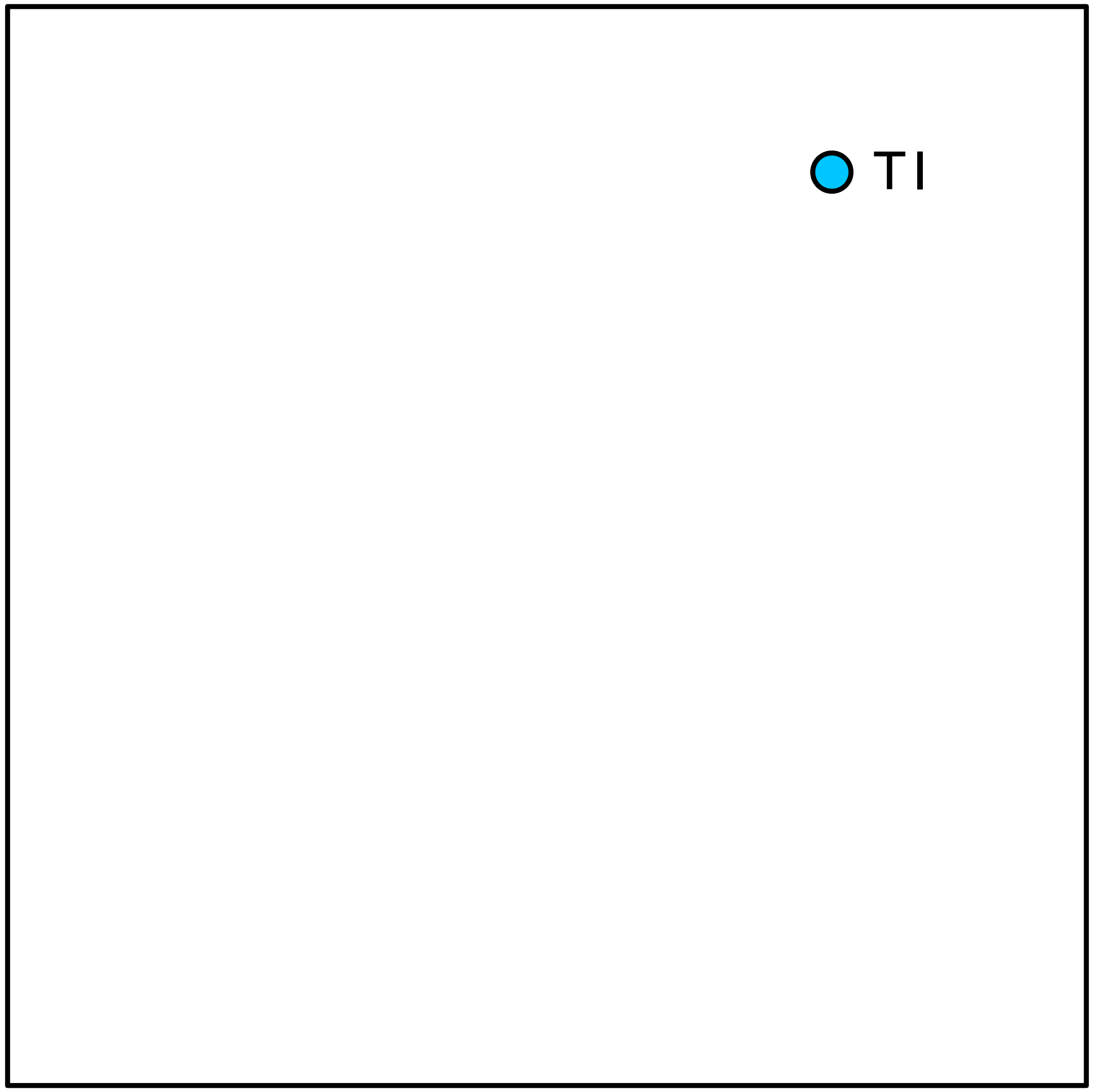
- For non-point failure patterns
  - Even spread random test cases will enhance the fault detection capabilities

How to achieve "even spread"?

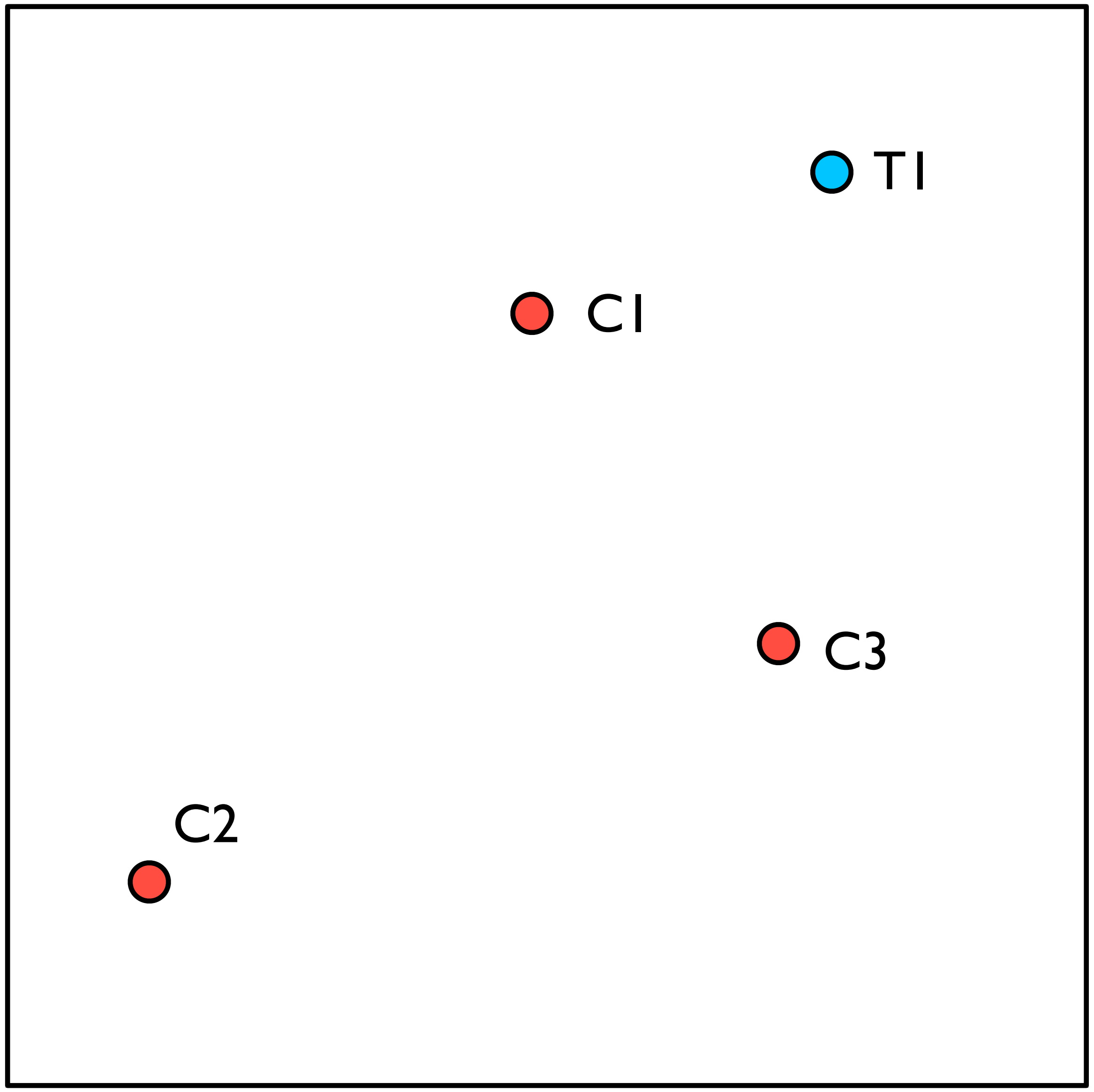
Adaptive Random Testing (ART) considering

1) notion of distance
2) notion of exclusion
3) notion of partitioning
4) …

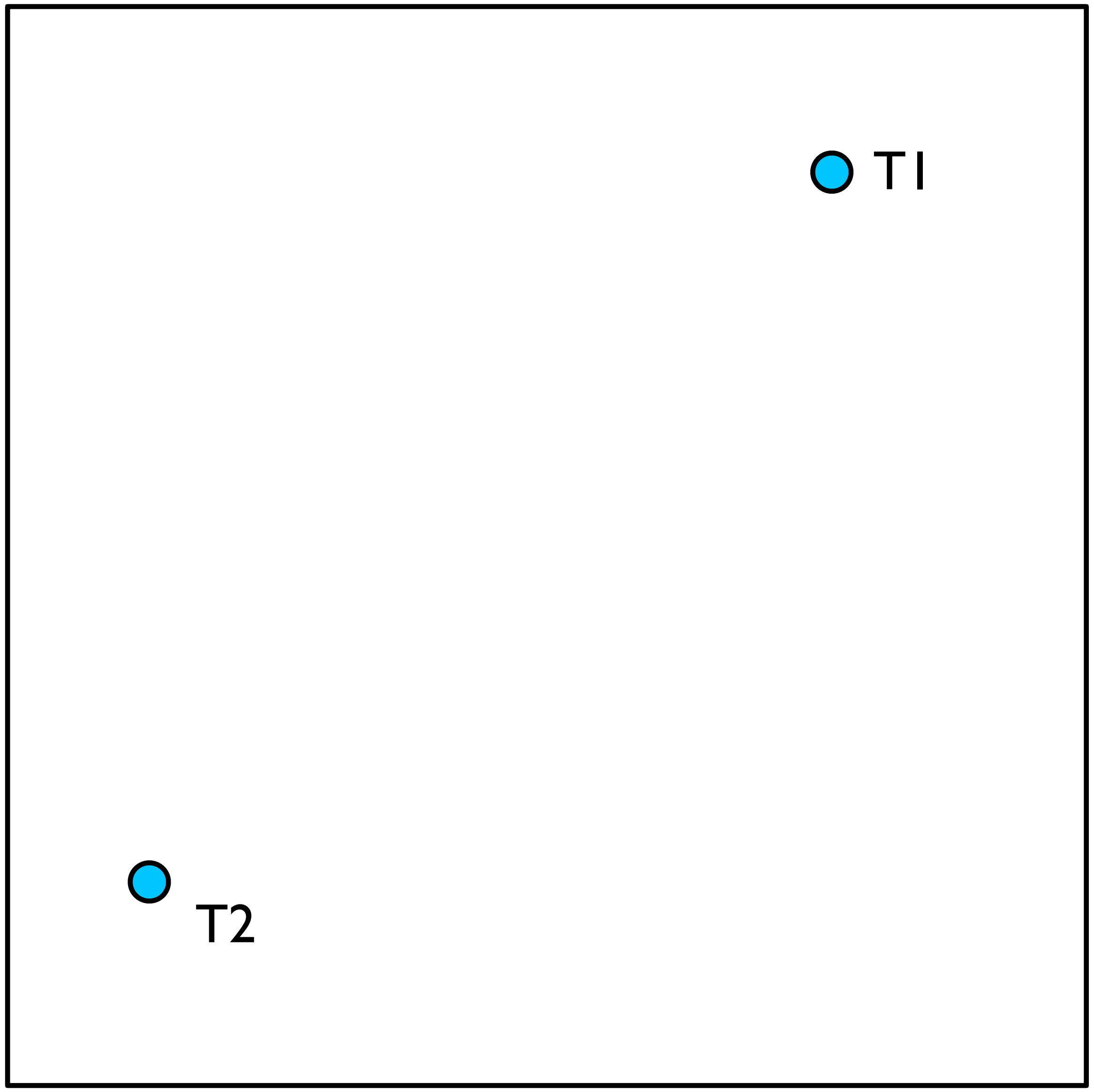
ART by distance



ART by distance



ART by distance



Distance Definition

- Distance between two integer arrays
  - Euclidean Distance

$$D_e(p,q) = D_e(q,p) = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + \ldots + (p_n - q_n)^2} = \sqrt{\sum_{i=1}^{n} (p_i - q_i)^2}$$

This Class

- Automated test generation
  - Black-box test generation

    Exhaustive testing

    Domain testing

    Random testing and adaptive random testing
- Debugging
  - Fault localization
  - Delta debugging

CE/CS/SE 3354
Software Engineering

Dynamic Bug Detection

What is fault localization?

```
int mid(int x,int y,int z) {
    int m;
    m = z;
    if (y < z) {
      if (x < y)
        m = y;
      else if (x < z)
        m = y;
    } else {
      if (x > y)
        m = y;
      else if (x > z)
        m = x; }
    return m;
}
```

Faulty!

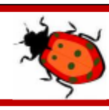
What is fault localization?

```
int mid(int x,int y,int z) {
    int m;
    m = z;
    if (y < z) {
      if (x < y)
        m = y;
      else if (x < z)
        m = y;//should be m=x
    } else {
      if (x > y)
        m = y;
      else if (x > z)
        m = x; }
    return m;
}
```

Fault Localization is the process of automatically narrowing or guiding the search for faulty code to help a developer debug more quickly.

How to do fault localization?

Tests

Statements	3,3,5	1,2,3	3,2,1	5,5,5	5,3,4	2,1,3
`int m;`						
`m = z;`						
`if (y < z) {`						
`if (x < y)`						
`m = y;`						
`else if (x < z)`						
`m =` 						
`} else {`						
`if (x > y)`						
`m = y;`						
`else if (x > z)`						
`m = x; }`						
`return m;`						
	Pass	Pass	Pass	Pass	Pass	Fail

Uses dynamic Information:

- Statements executed by each test
- The pass/fail outcome of each test

Performs statistical analysis:

- Statements mainly executed by failed tests are more suspicious

A representative technique: Tarantula

Statements	3,3,5	1,2,3	3,2,1	5,5,5	5,3,4	2,1,3	Susp
`int m;`							0.5
`m = z;`							0.5
`if (y < z) {`							0.5
`if (x < y)`							0.63
`m = y;`							0
`else if (x < z)`							0.71
`m =` 							0.83
`} else {`							0
							0
							0
							0
							0
`return m;`							0.5
	Pass	Pass	Pass	Pass	Pass	Fail	

$$\frac{1/1}{1/1+5/5}$$

$$\frac{1/1}{1/1+1/5}$$

Tarantula :

$$Suspicious(s) = \frac{\frac{fail(s)}{totalfail}}{\frac{fail(s)}{totalfail} + \frac{pass(s)}{totalpass}}$$

More Formulae

- Tarantula

$$Suspicious(s) = \frac{fail(s)/totalfail}{fail(s)/totalfail + pass(s)/totalpass}$$

- SBI

$$Suspicious(s) = \frac{fail(s)}{fail(s) + pass(s)}$$

- Jaccard

$$Suspicious(s) = \frac{fail(s)}{allfailed + pass(s)}$$

- Ochiai

$$Suspicious(s) = \frac{fail(s)}{\sqrt{allfailed * (pass(s) + fail(s))}}$$

This Class

- Automated test generation
  - Black-box test generation
    - Exhaustive testing
    - Domain testing
    - Random testing and adaptive random testing
- Debugging
  - Fault localization
  - Delta debugging

Debugging

- Sometimes the inputs is too complex…
  - Quite common in real world (compiler, office, browser, database, OS, …)
  - Locate the relevant inputs

Consider Mozilla Firefox

- Taking html pages as inputs
- A large number of bugs are related to loading certain html pages
  - Corner cases in html syntax
  - Incompatibility between browsers
  - Corner cases in Javascripts, css, …
  - Error handling for incorrect html, Javascript, css, …

How do we go from this

```
<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">HPUX<
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<
```

To this…

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

Motivation

- Turning bug reports with real web pages to minimized test cases
- The minimized test case should still be able to reveal the bug
- Benefit of simplification
  - Easy to communicate
  - Remove duplicates
  - Easy debugging

    Involve less potentially buggy code

    Shorter execution time

Delta Debugging

- The problem definition
  - A program exhibit an error for an input
  - The input is a set of elements
  - E.g., a sequence of API calls, a text file, a serialized object, …
  - Problem:

    Find a smaller subset of the input that still cause the failure

A generic algorithm

- How do people handle this problem?

- Binary search
  - Cut the input to halves
  - Try to reproduce the bug
  - Iterate

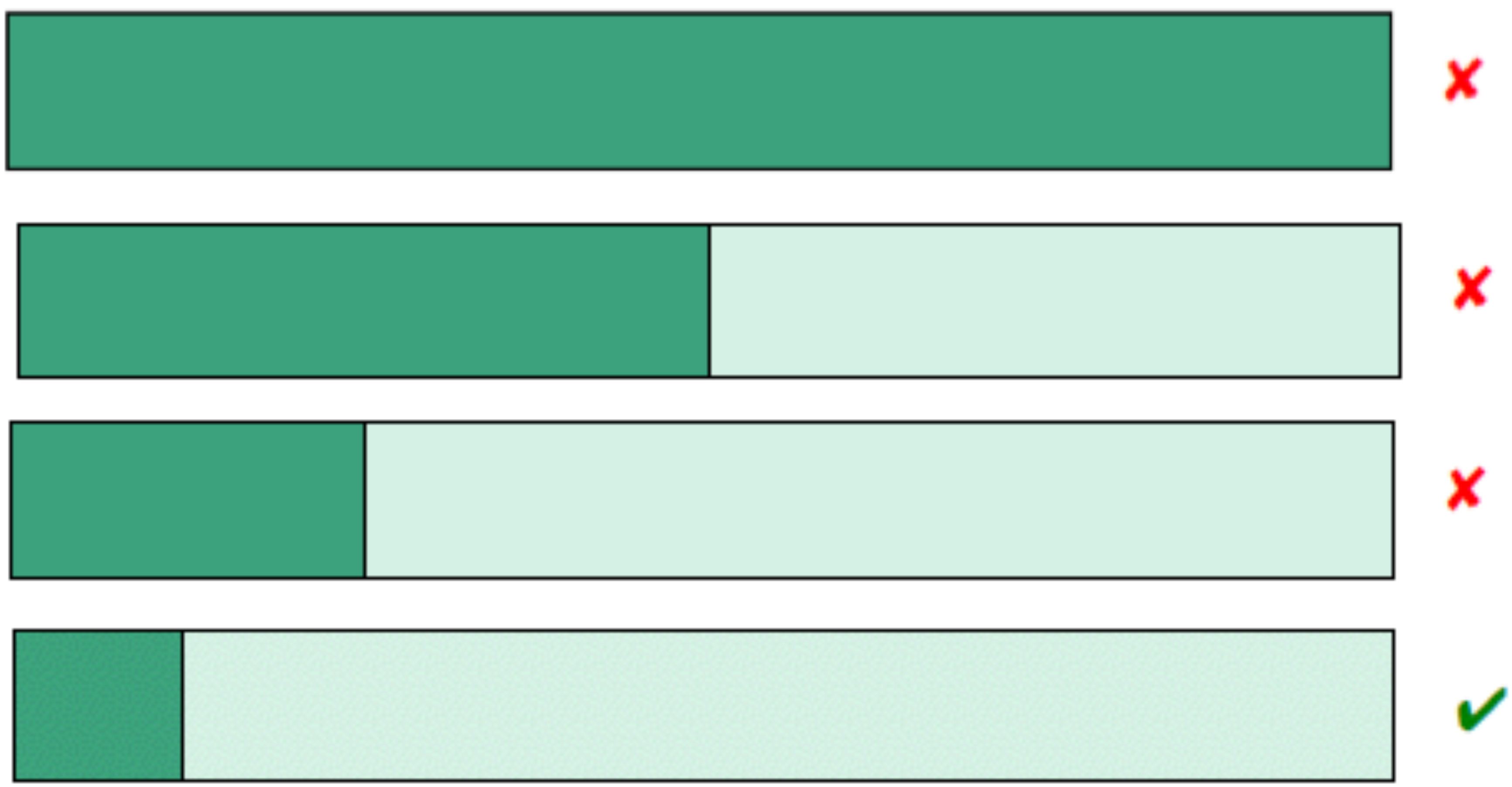
Delta Debugging Version 1

- The set of elements in the bug-revealing input is **I**
- Assumptions
  - Each subset of **I** is a valid input:

    Each Subset of **I** -> success / fail
  - A single input element **E** causes the failure
  - **E** will cause the failure in any cases (combined with any other elements) (Monotonic)

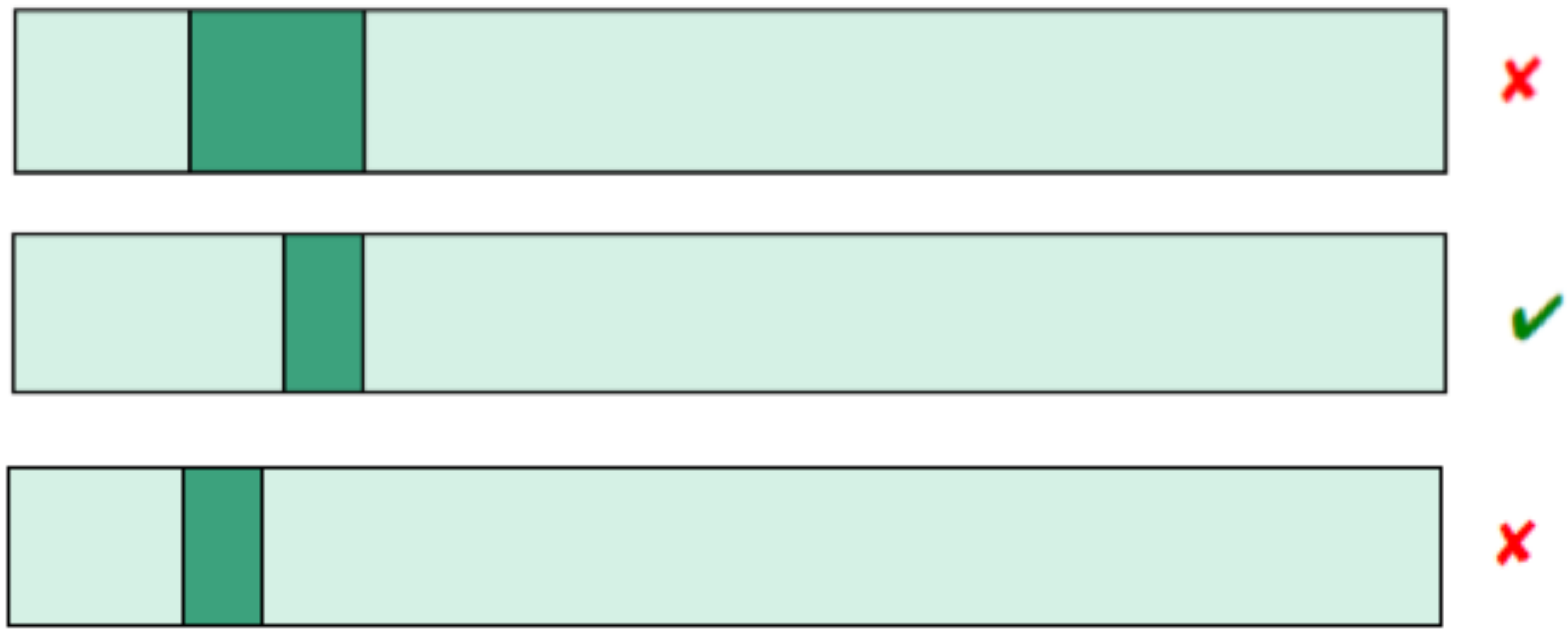
Solution is simple

- Go with the binary search process
- Throw away half of the input elements, if the rest input elements still cause the failure



Solution is simple

- Go with the binary search process
- Throw away half of the input elements, if the rest input elements still cause the failure



A single element: we are done!

Example

Assume I = { 1, 2, 3, 4, 5, 6, 7, 8}

- The bug is due to input element 7

Configuration	Result
1 2 3 4	✓
5 6 7 8	✗
5 6	✓
7 8	✗
7	✗

Delta Debugging Version 1

- This is just binary search: easy to automate
- The assumptions do not always hold
- Let's look at the assumptions:
  - (I1 U I2) = ✗
  - => I1 = ✗ and I2 = ✓
  - or I1 = ✓ and I2 = ✗

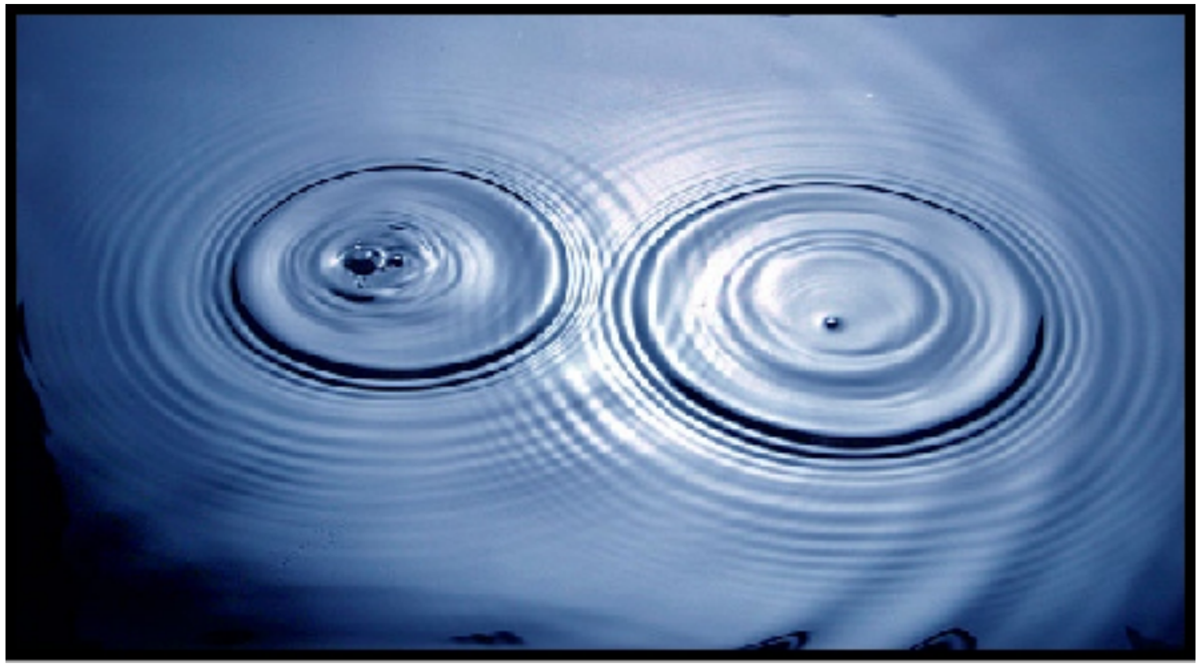
- It is interesting to see if this is not the case

Case I: multiple failing branches

- What happened if I1 = X and I2 = X ?
- A subset of I1 fails and also a subset of I2 fails
- We can simply continue to search both I1 and I2
  - And we find two fail-causing elements
  - They may be due to the same bug or not

Case II: Interference

- What happened if I1 = ✔ and I2 = ✔ ?
- This means that a subset of I1 and a subset of I2 cause the failure when they combined
- This is called interference



Handling Interference

- The cute trick
  - Consider I1 = ✔ and I2 = ✔
  - But I1 U I2 = ✗
  - An element D1 in I1 and an element D2 in I2 cause the failure together
  - We do binary search in I2 with I1
  - Split I2 to P1 and P2, try I1 U P1 and I1 U P2
  - Continue until you find D2, so that I1 U D2 cause the failure
  - Then we do binary search in I1 with D2 until find D1

Example I: Handle interference

Consider 8 input elements, of which 3 and 7 cause the failure when they applied together

Configuration								Result
1	2	3	4					✔
				5	6	7	8	✔ Interference!
1	2	3	4	5	6			✔
1	2	3	4			7	8	✘
1	2	3	4			7		✘
1	2					7		✔
		3	4			7		✘
		3				7		✘

Example II: Handle multiple interference

Consider 8 input elements, of which 3, 5 and 7 cause the failure when they applied together

Configuration	Result
1 2 3 4	✓
5 6 7 8	✓ Interference!
1 2 3 4 5 6	✓
1 2 3 4 7 8	✓ Second Interference! What to do?
1 2 3 4 5 6 7	✗ Go on with I1 U P1!
1 2 3 4 5 7	✗
1 2 5 7	✓
3 4 5 7	✗
3 5 7	✗

Delta Debugging Version 2

- The set of elements in the bug-revealing input is I
- New Assumptions
  - Each subset of I is a valid input
  - A subset of input elements E causes the failure
  - E will cause the failure in any cases (combined with any other elements)

Delta Debugging Version 2

- Algorithm
  - Split I to I1 and I2
  - Case I: I1 = ✗ and I2 = ✓
    Try I1
  - Case II: I1 = ✓ and I2 = ✗
    Try I2
  - Case III: I1 = ✗ and I2 = ✗
    Try both I1 and I2
  - Case IV: I1 = ✓ and I2 = ✓
    Handle interference for I1 and I2

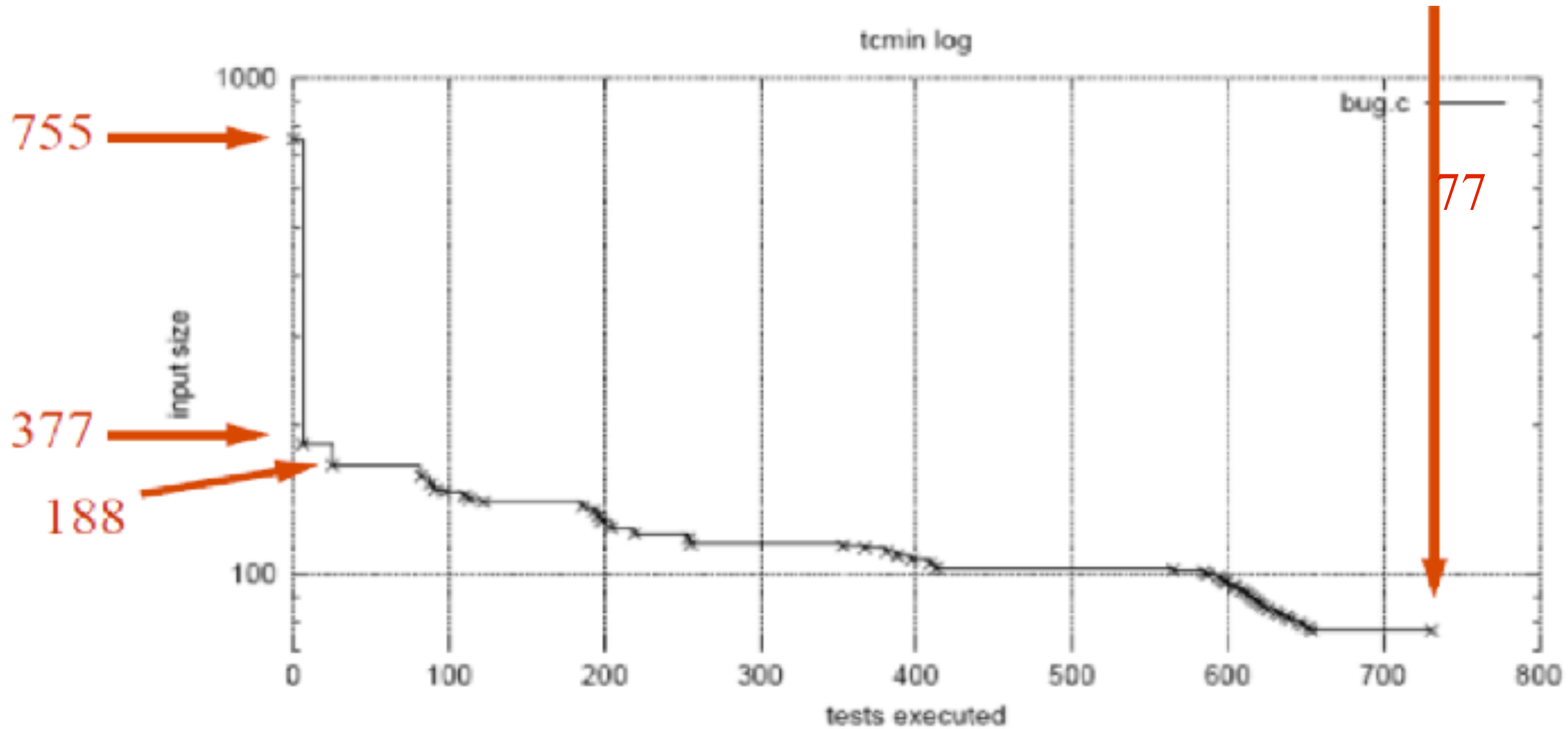
Real example: GNU Compiler

- This input program (bug.c) causes gcc 2.59.2 to crash when all optimizations are enabled
- Minimize it to debug gcc
- Consider each character as an element

```
#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);
}
```

Real example: GNU Compiler

- Delta debugging in action



GCC compiler example

- The minimized code:

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

- The code is minimal
  - No single character can be removed
  - Even every space is removed
  - The function name has been changed from mult to a single t
  - Gcc is executed for 700+ times

Summary of dynamic debugging

- Tools can help us to narrow the scope to consider
  - Fault localization

    Reduce the code to be considered
  - Delta debugging

    Reduce the inputs to be considered
- Paper reading
  - **Fault localization:** James A. Jones, Mary Jean Harrold, John T. Stasko: **Visualization of test information to assist fault localization.** ICSE 2002: 467-477
  - **Delta Debugging:** Andreas Zeller: **Yesterday, My Program Worked. Today, It Does Not. Why?** ESEC / SIGSOFT FSE 1999: