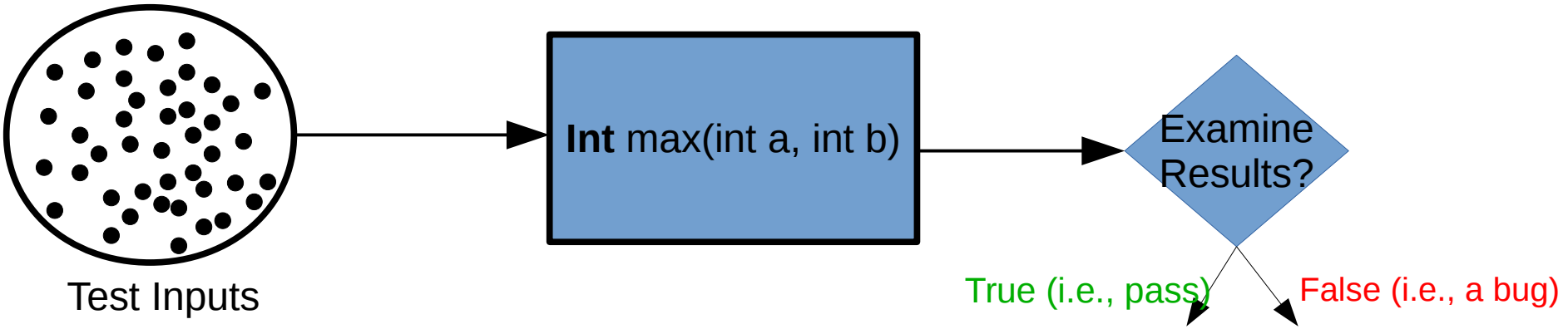
What is So Hard about Testing?

Example: Lets consider a program under test (**PUT**) that takes two integer and returns the maximum value.

```
int max(int a, int b)
// effects: a > b => returns a
// a < b => returns b
// a = b => returns a
```

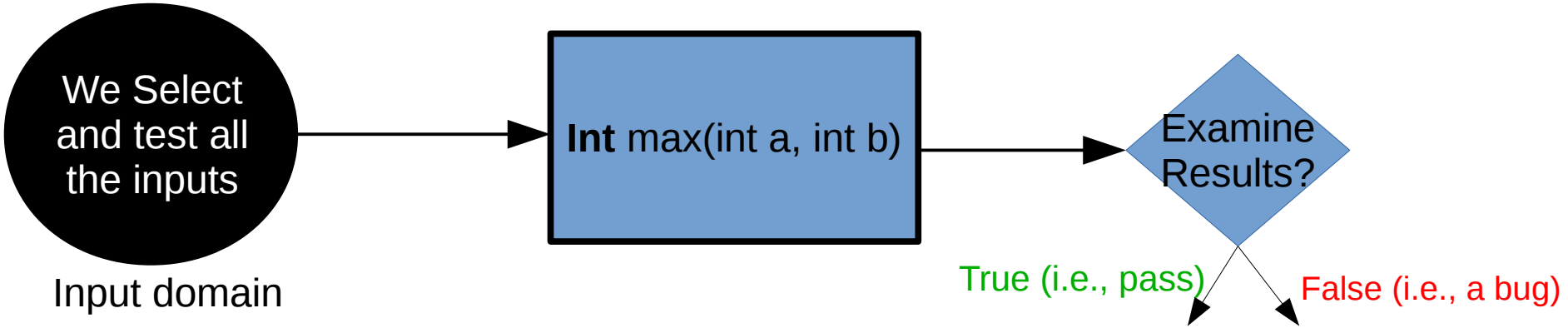


The question is: How can we select a set of inputs from input domain (i.e., test inputs) for our **PUT** that after we run them, we have enough confidence that the PUT is implemented correctly?

Exhaustive Testing

First approach: we select all the inputs (i.e., do **exhaustive testing**)

- "**Just try it and see if it works for int max(int a, int b) ...**"



- If **a** and b are 32bit integers, we'll have a total number of combinations, which is $2^{32} \times 2^{32} = 2^{64} \simeq 10^{19}$
- So we need to run 10^{19} test cases to cover the whole input domain for the **PUT** int max(int a, int b)
- Exhaustive testing would require hundreds of years to cover all possible inputs.

  Sounds totally impractical – and this is a trivial small problem

Random Testing

- Second approah: choose our test inputs randomly (i.e., do random testing)

  "**Just try it and see if it works for int max(int a, int b) ...**"

int max(int a, int b)
// *effects*: **a > b** => returns a
// **a < b** => returns b
// **a = b** => returns a

- So we can randomly choose 3 test cases

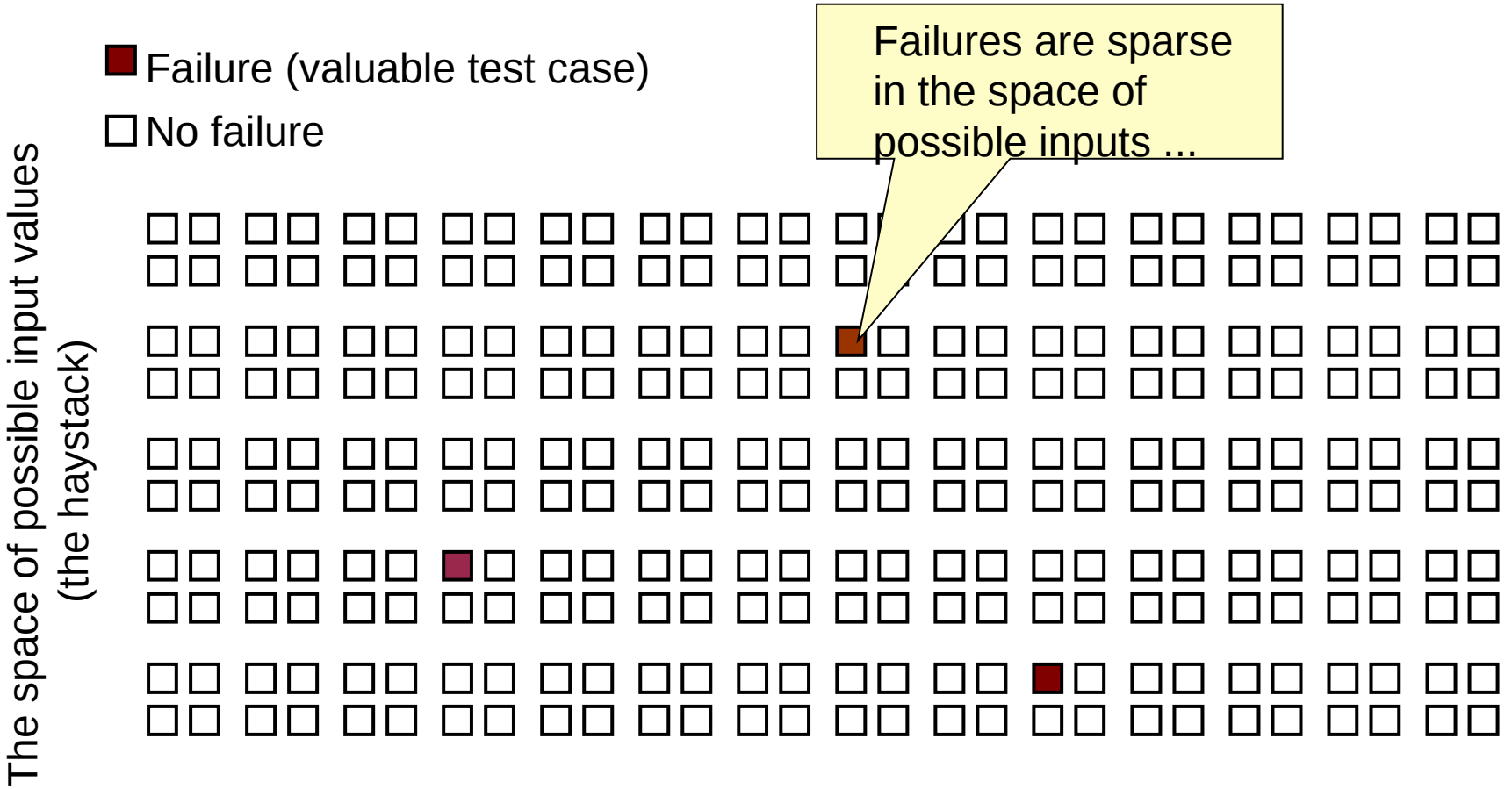
int(1,2) expected results 2, int(2,1) expected results 2, int(1,1) expected results 1.

- Key problem: what are some values or ranges of a and b might be worth testing.
- How about.. a = INT_MAX (i.e., +2,147,483,648) and b= INT_MIN(i.e., −2,147,483,648)
- How about a=0 and b=-1, etc.
- **Why not random**, failing values are sparse in the input domain space. It is very unlikely by randomly picking inputs, we'll pick the failing values

— needles in a very big haystack

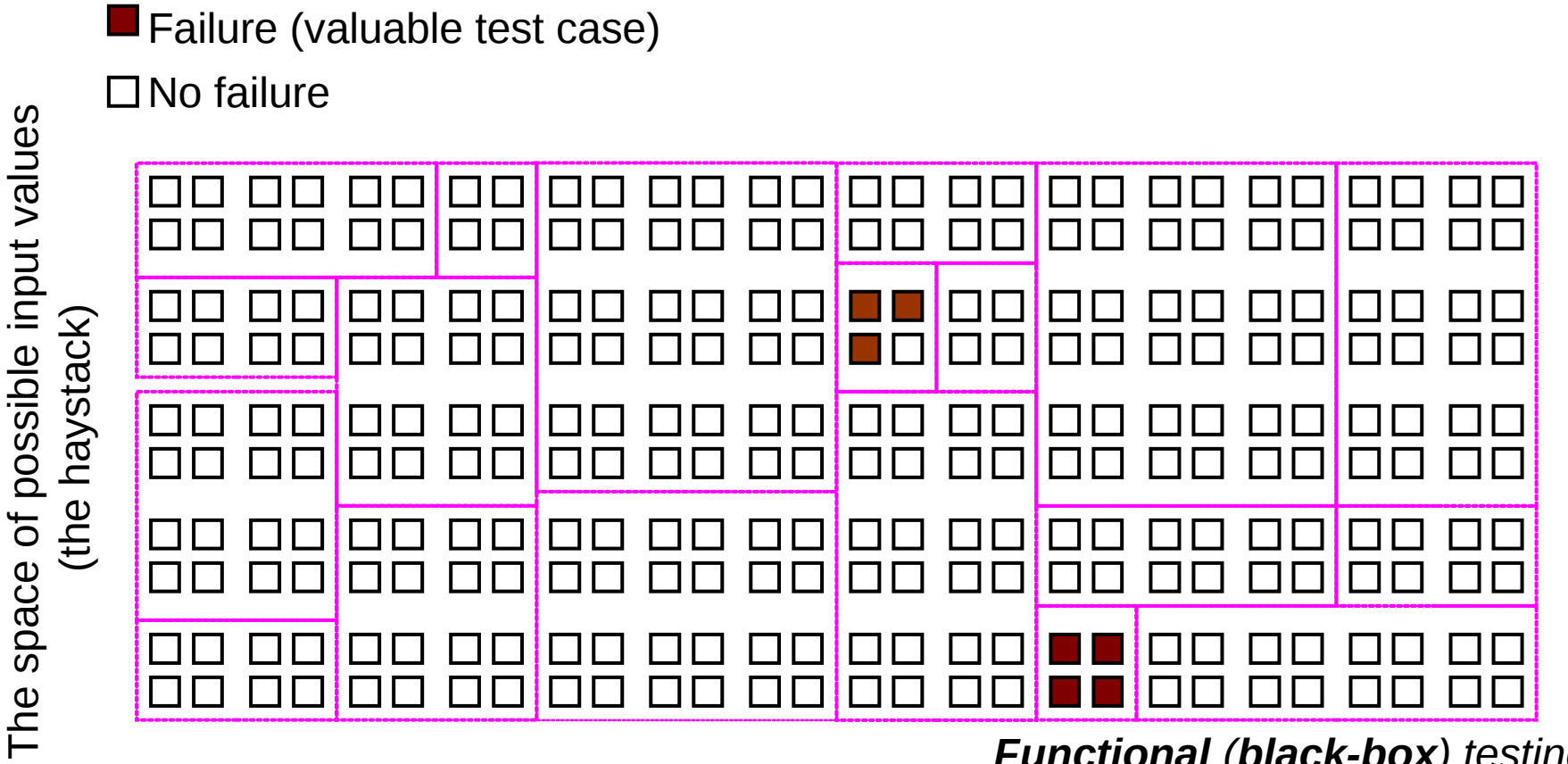
Oregon State University

Why not RANDOM?



Oregon State University

Equivalence Partitioning & Boundary Value Analysis



***Functional** (**black-box**) testing is one way of drawing pink lines to isolate regions with likely failures*

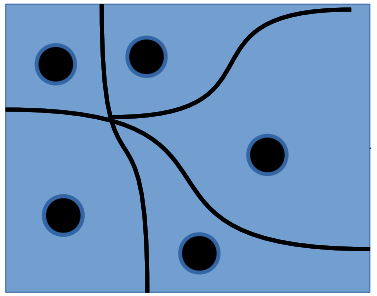
Oregon State University

Black Box Testing

- **Choosing tests** based on partitioning the input domain.

  Identify sets (partitions) with same behavior

  Try one input from each set



- "**Just try it and see if it works for int max(int a, int b) //which is a simple program to find maximum of two numbers**"
- Some possible partitions:

  a > 0 b>0

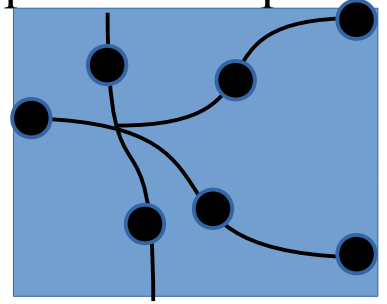
  a <0 **b** <0

  a = 0 **b** =0

- **Boundary Testing**: create tests at the edges or extreme ends of partitions input values

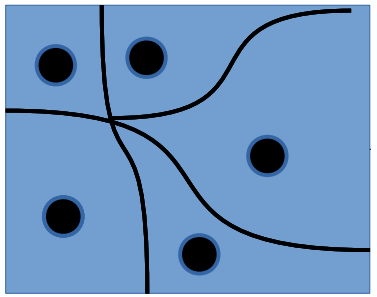
  a=INT_MIN **b**=INT_MIN (boundary values)

  a=INT_MAX **b**=INT_MAX (boundary values)



Oregon State University

Partition the Input Space & Boundary Testing



- Some possible inputs for **int max(int a, int b)**

a > 0	b>0	
a <0	**b** <0	
a = 0	**b** =0	(boundary values)
a=INT_MIN	**b**=INT_MIN	(boundary values)
a=INT_MAX	**b**=INT_MAX	(boundary values)

- Some Test Cases

max(1, 1)	=>	1
max(1, -1)	=>	1
max(1, 0)	=>	1
max(1, INT_MIN)	=>	1
max(1, INT_MAX)	=>	INT_MAX
….		

Oregon State University

Structural (White-Box) Testing

- Choosing test inputs to cover code.
- White Box Testing requires two basic steps

  1- Understand the source code

  - 2- Create test cases and execute

- The **goals**

  - Ensure test cases (test suites) cover (executes) all the program

  - Measure quality of test cases (test suites) with % coverage

- **Varieties** of coverage

  - For example, Statement coverage, Branch Coverage, Path Coverage

- What is full Coverage?

```
int max(int a, int b){
  int m=a;
  if(a>=b)
     m=a;
  return m;
}
```

To achieve 100% statement coverage of this code segment just one test case is required with a ≥ b (e.g., (5,3) => 5)

- It covers every statement/line
- It misses the bug!

statement coverage is not enough!

Note: here we are doing structural (white box) test, since we are **choosing our input values** in order ensure statement/line coverage

References

Pezze + Young, “Software Testing and Analysis”, Chapter 10 & 11
Patton, Ron. "Software Testing." (2000). Chapter 4 & 5
Sommerville, I., Software Engineering, Sixth Edition, Addison-Wesley, 2001 Chapter 20

Oregon State
University