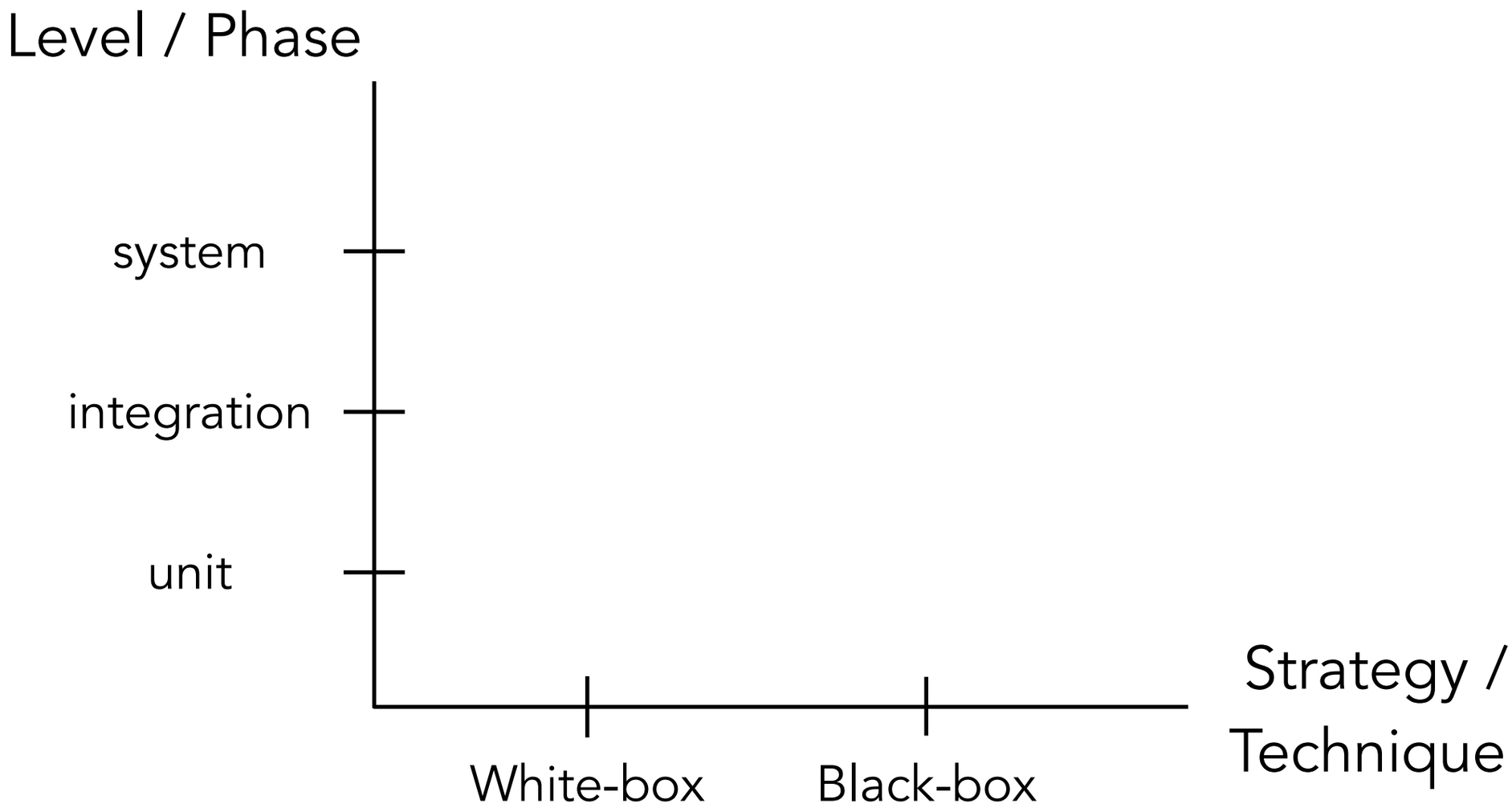
Software Testing, Verification and Validation

October 12, 2022
Week #4 — Lecture #3

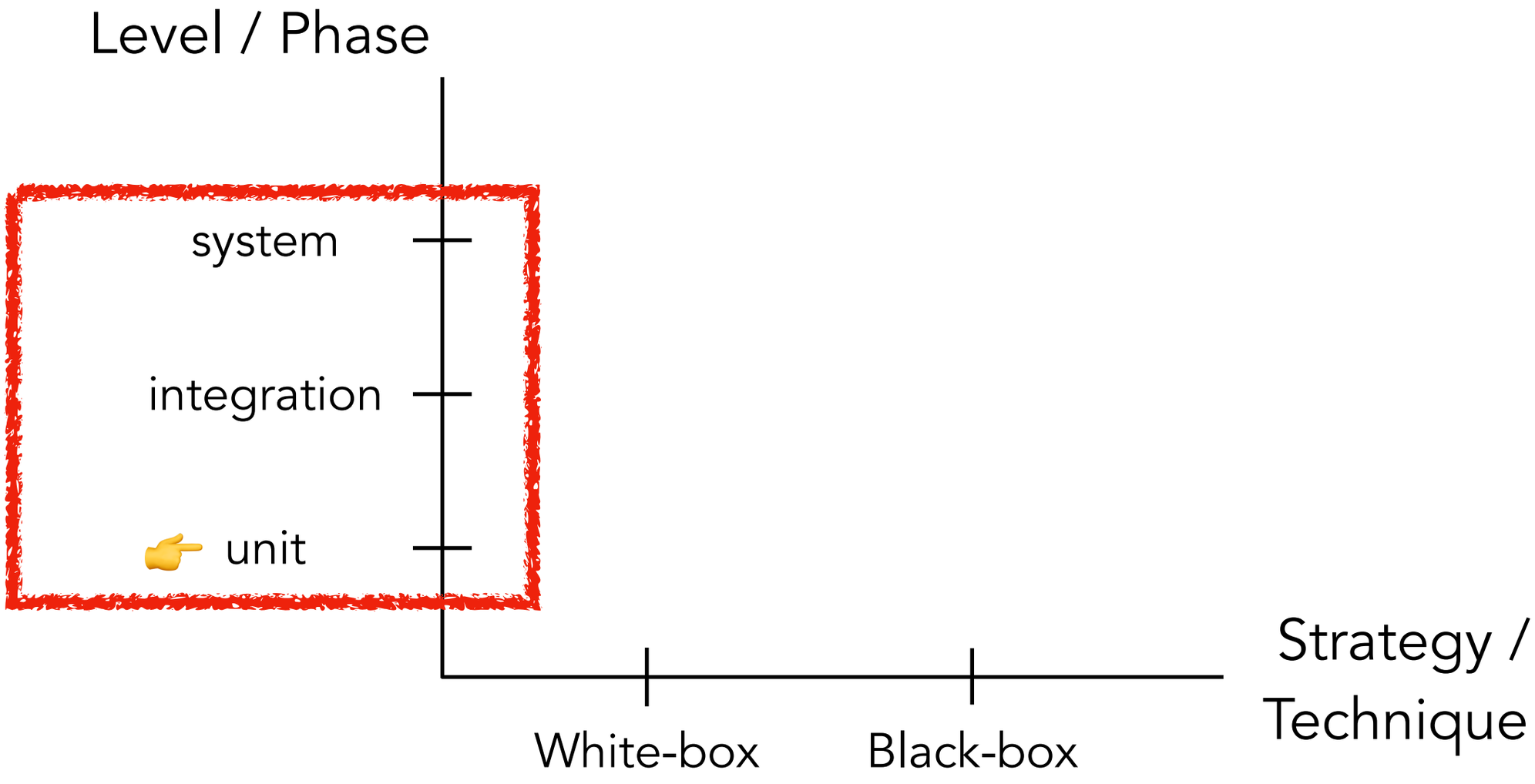
U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Last lecture, we discussed the main Levels/Phases of testing (unit, integration, system) and we also introduced two testing strategies (black-box and white-box). We further discussed in detail two black-box techniques: equivalence and category partition.

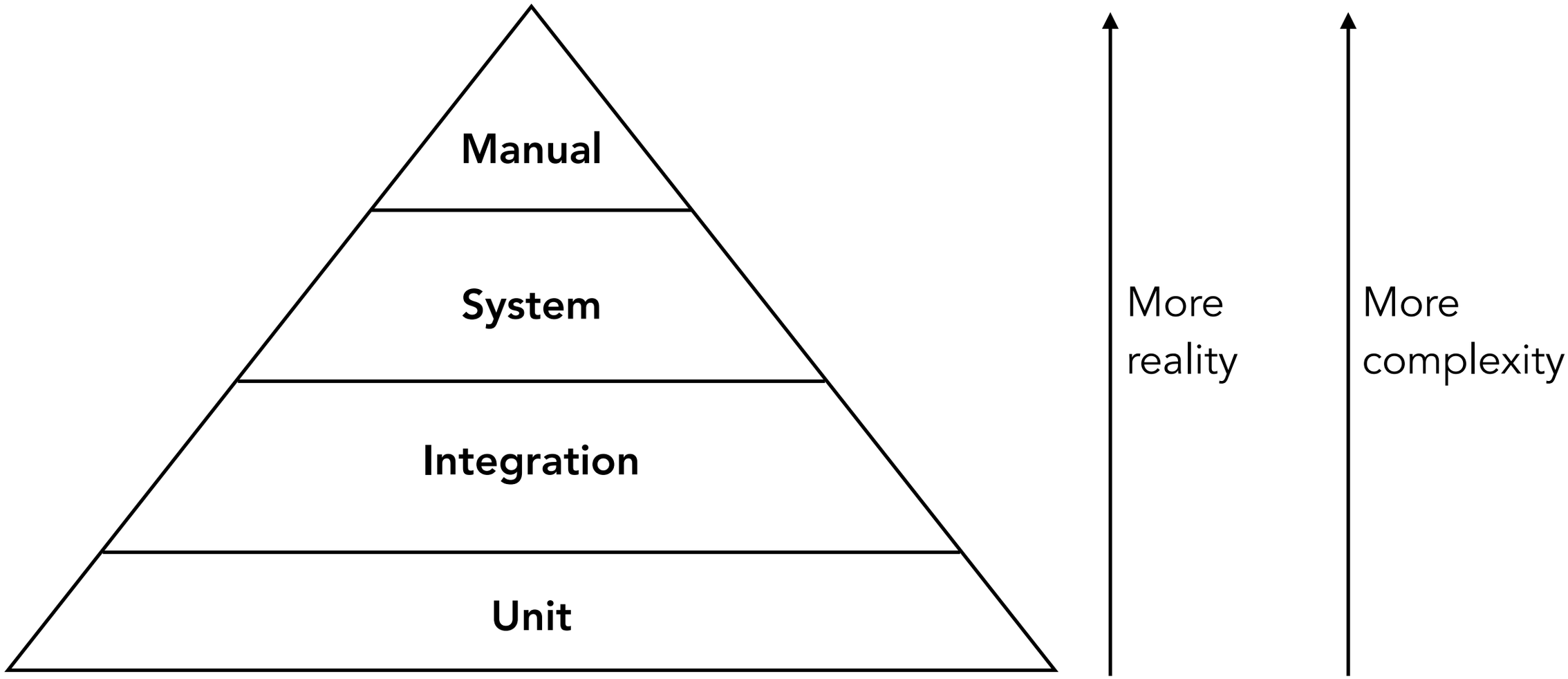
Test types


Level / Phase
system
integration
unit
White-box
Black-box
Strategy /
Technique

Test types

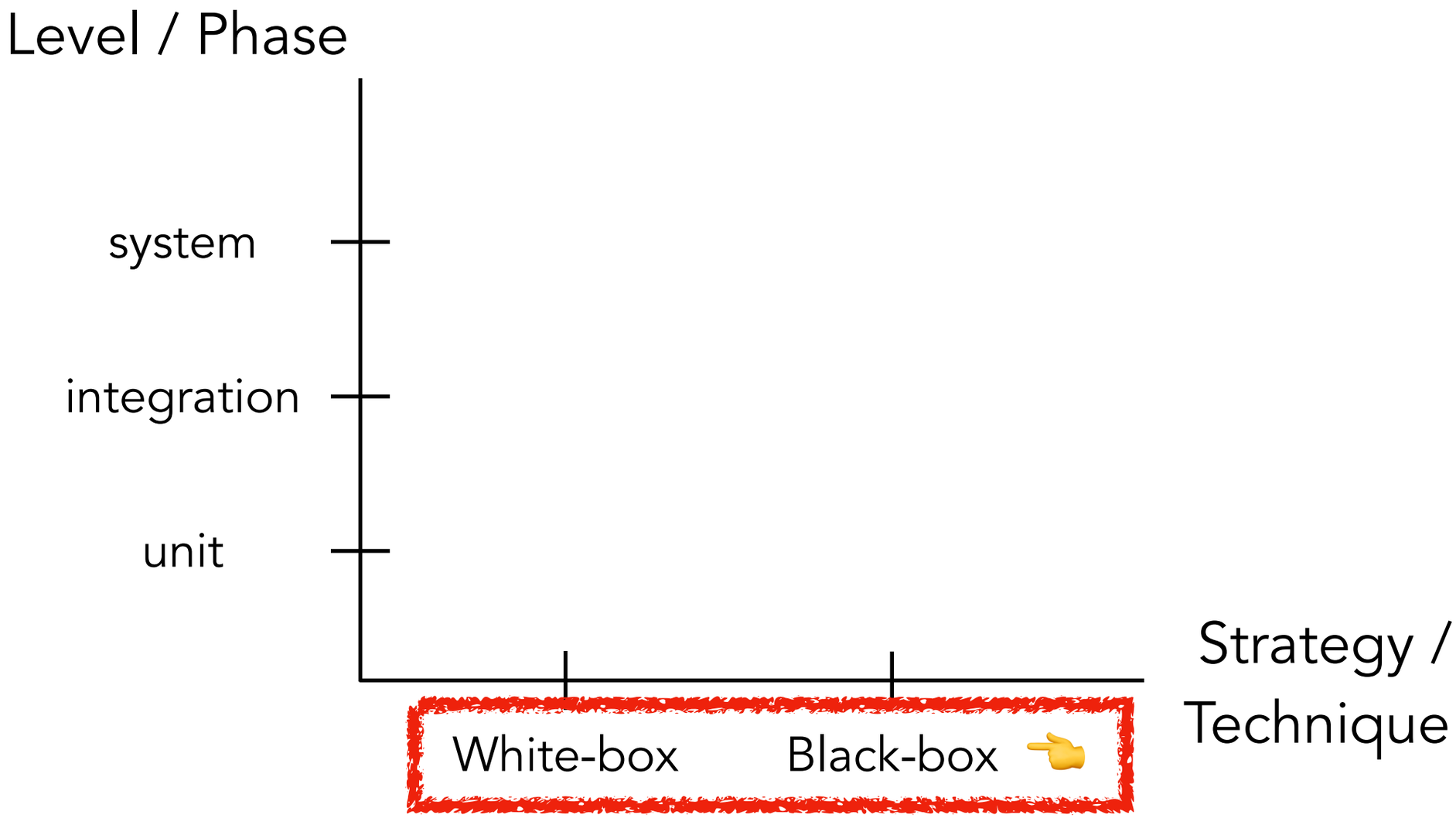

Level / Phase
system
integration
👉 unit
White-box
Black-box
Strategy / Technique

Testing Pyramid

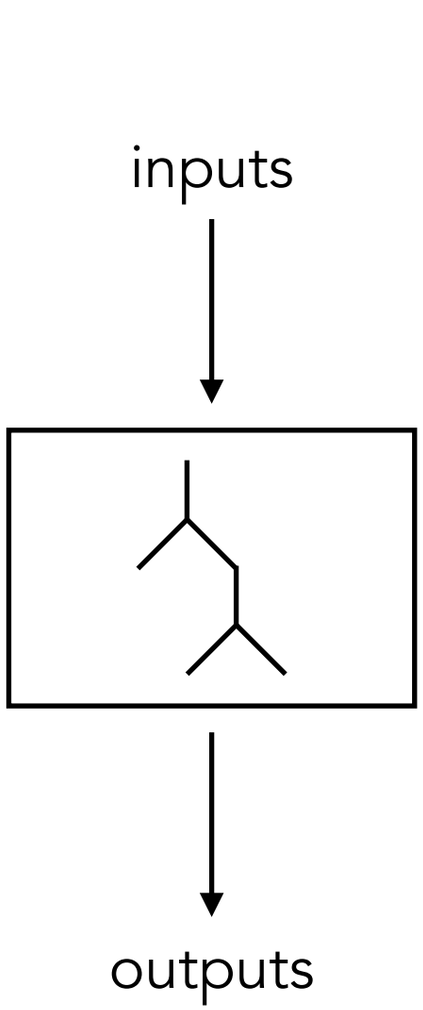


by Martin Fowler, https://martinfowler.com/bliki/TestPyramid.html

Test types


Level / Phase
system
integration
unit
Strategy / Technique
White-box
Black-box 👈

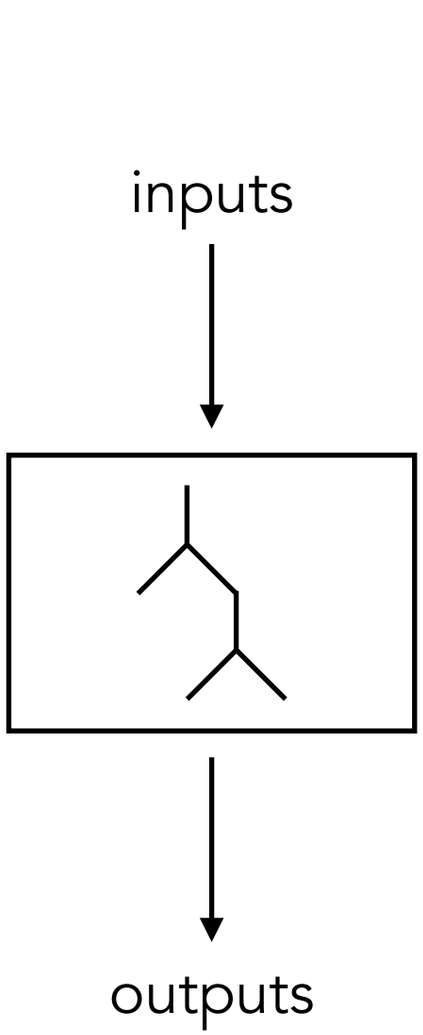
White-box testing



Knowledge sources

Control flow graphs
Data flow graphs
Cyclomatic complexity
...

White-box testing



Techniques

Control flow testing/coverage

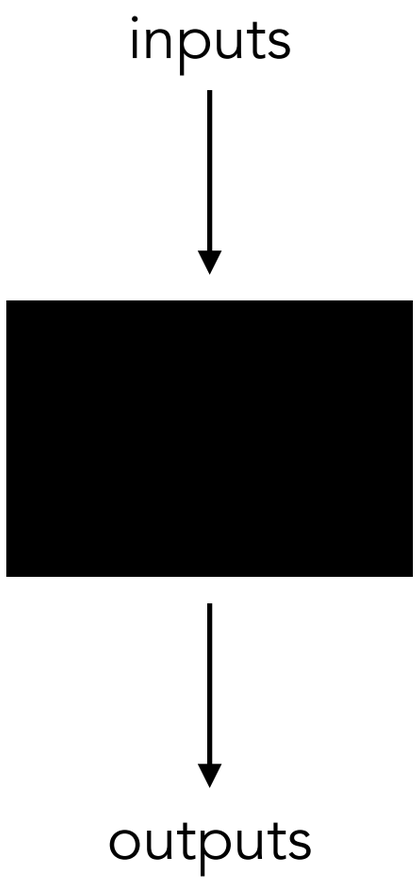
Data flow testing/coverage

Class testing/coverage

Mutation testing

…

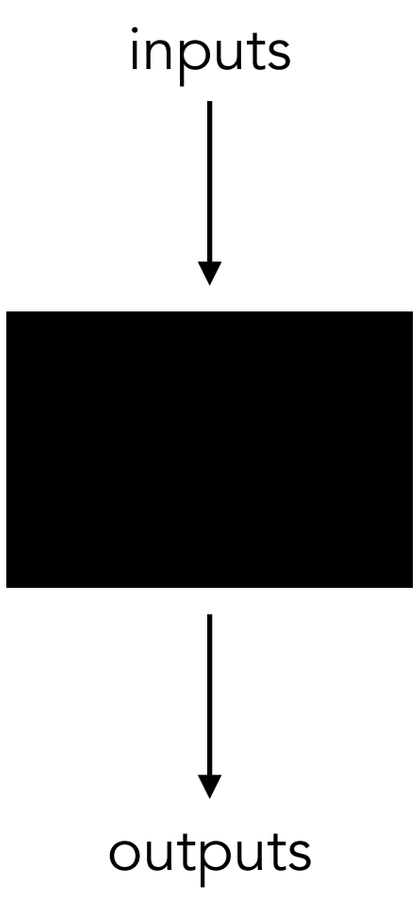
Black-box testing



Knowledge sources

Requirements document
Specifications
User manual
Models
Domain knowledge
Intuition
Experience
…

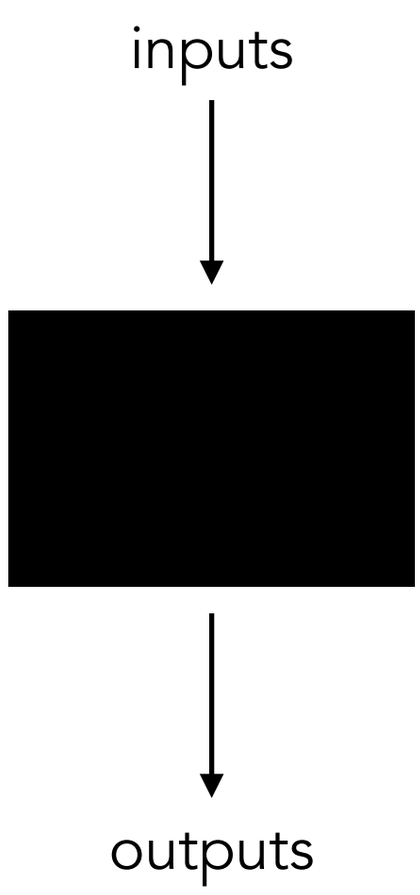
Black-box testing



Techniques

- ✅ Equivalence class partitioning
- ✅ Category partition
- Boundary value analysis
- Cause effect graphing
- Error guessing
- Random testing
- State-transition testing
- Scenario-based testing
- …

Black-box testing



Techniques

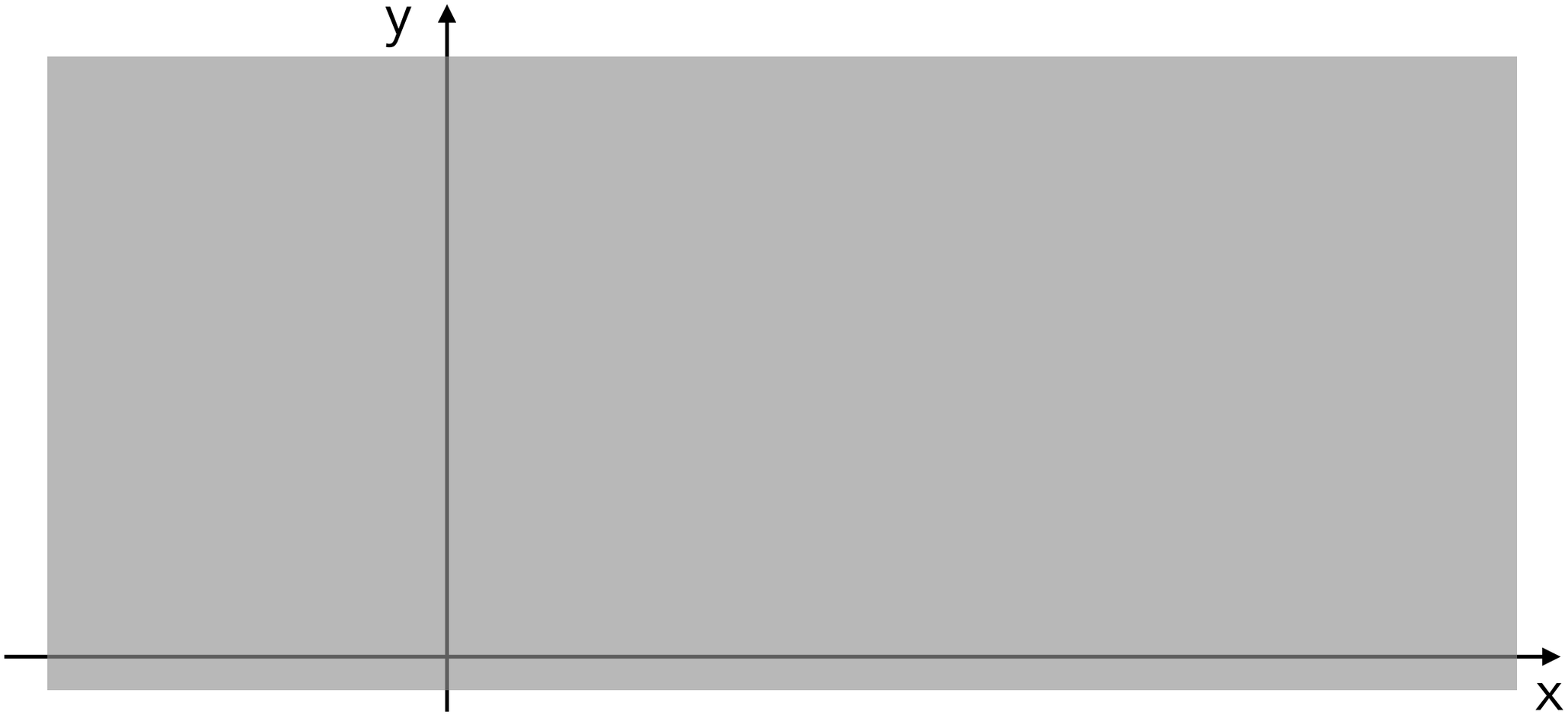
- ✅ Equivalence class partitioning
- ✅ Category partition
- 👉 Boundary value analysis
- Cause effect graphing
- Error guessing
- 👉 Random testing
- State-transition testing
- Scenario-based testing
- …

Partition example *(recap)*

Consider an application that requires two integer inputs x and y.

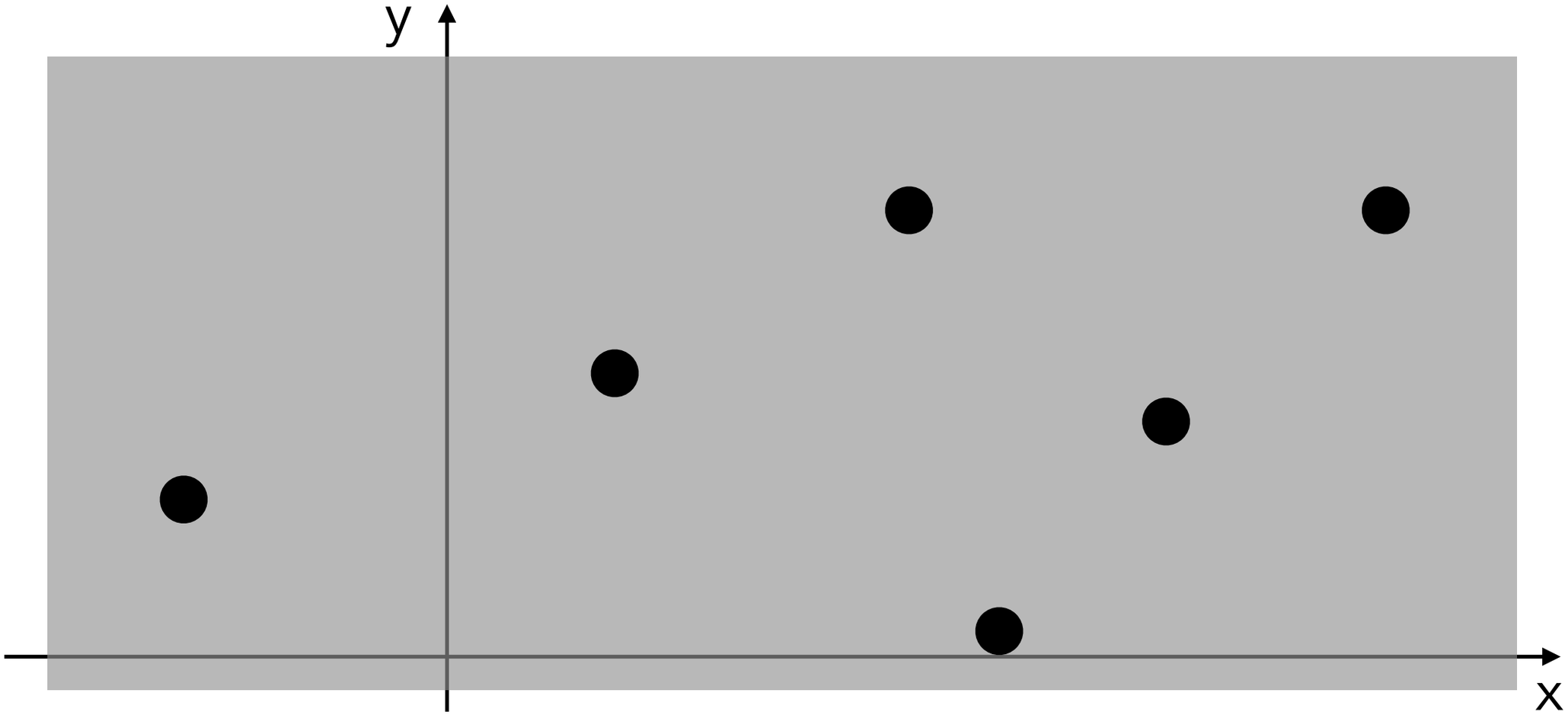
Partition example *(recap)*

Consider an application that requires two integer inputs x and y.



Partition example *(recap)*

Consider an application that requires two integer inputs x and y.

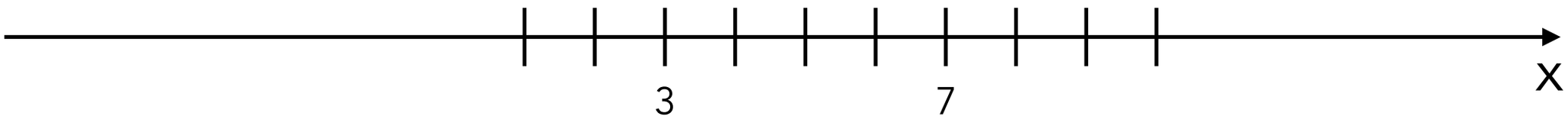


Partition example *(recap)*

Consider an application that requires two integer inputs x and y. Each of these inputs is expected to lie in the following ranges: x > 3 and x <= 7 and y >= 5 and y <= 9.

Partition example *(recap)*

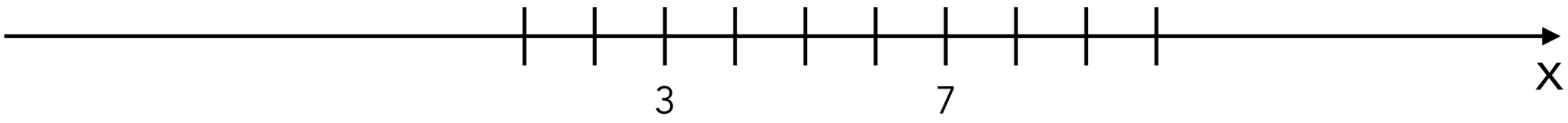
Consider an application that requires two integer inputs x and y. Each of these inputs is expected to lie in the following ranges: x > 3 and x <= 7 and y >= 5 and y <= 9.



Partition example *(recap)*

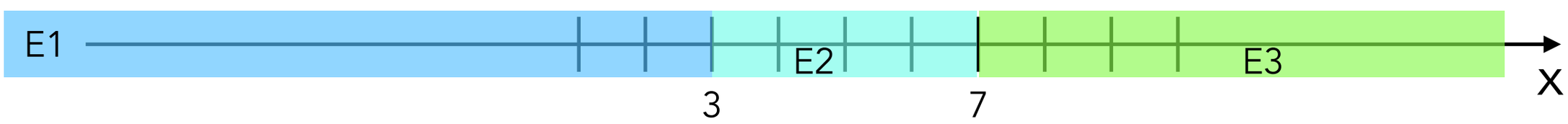
Consider an application that requires two integer inputs x and y. Each of these inputs is expected to lie in the following ranges: x > 3 and x <= 7 and y >= 5 and y <= 9.

How many categories are there for x?
Please describe them.



Partition example *(recap)*

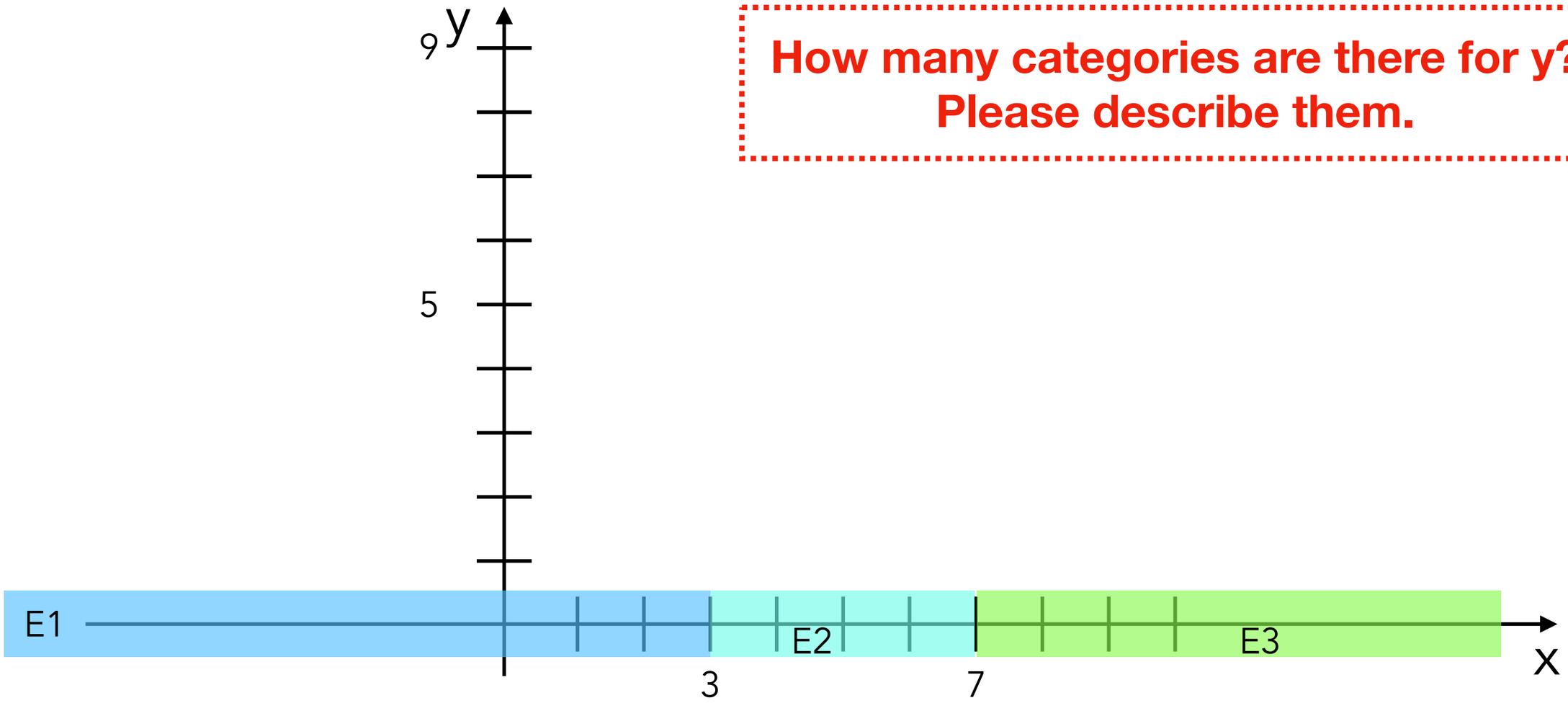
Consider an application that requires two integer inputs x and y. Each of these inputs is expected to lie in the following ranges: x > 3 and x <= 7 and y >= 5 and y <= 9.



Partition example *(recap)*

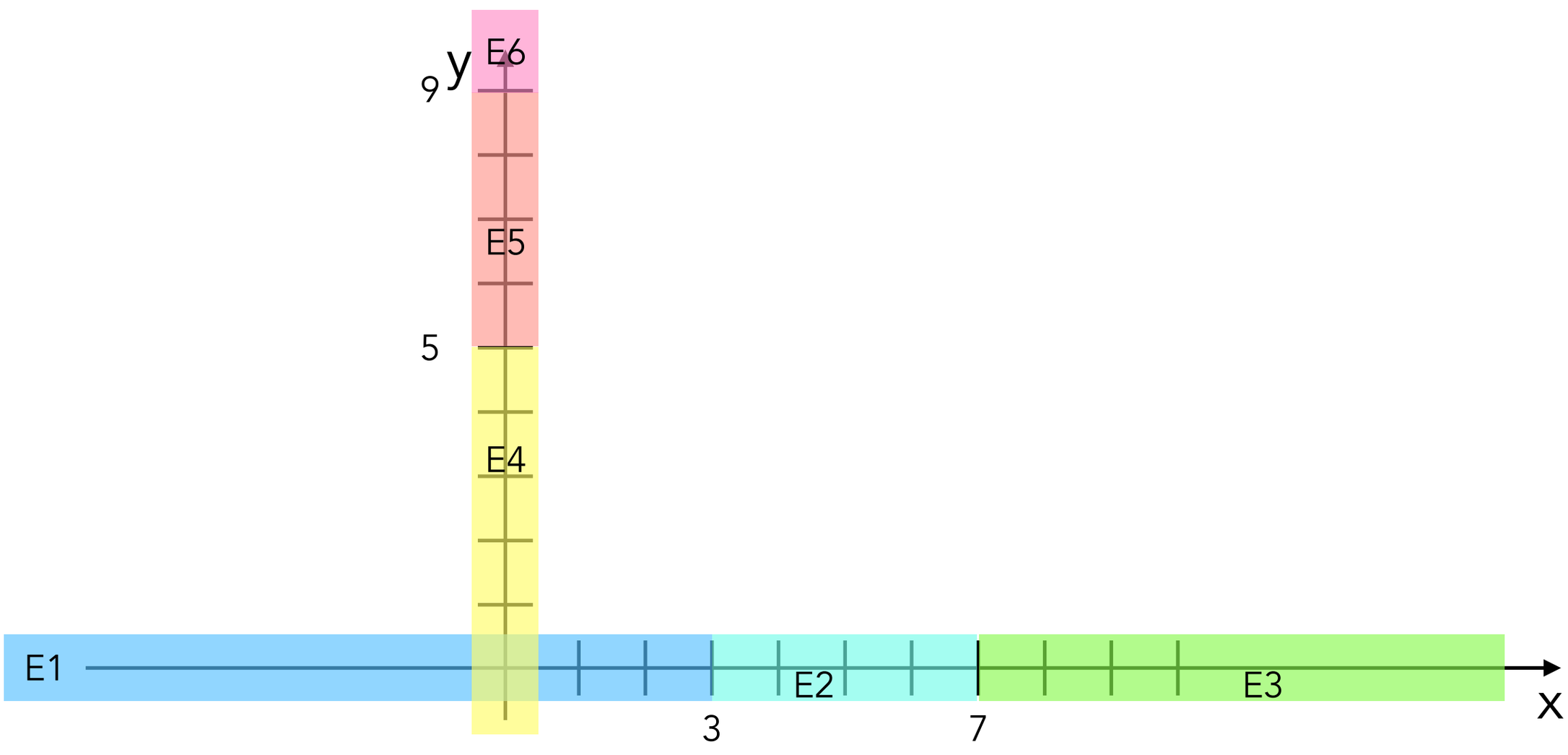
Consider an application that requires two integer inputs x and y. Each of these inputs is expected to lie in the following ranges: x > 3 and x <= 7 and y >= 5 and y <= 9.

How many categories are there for y? Please describe them.



Partition example *(recap)*

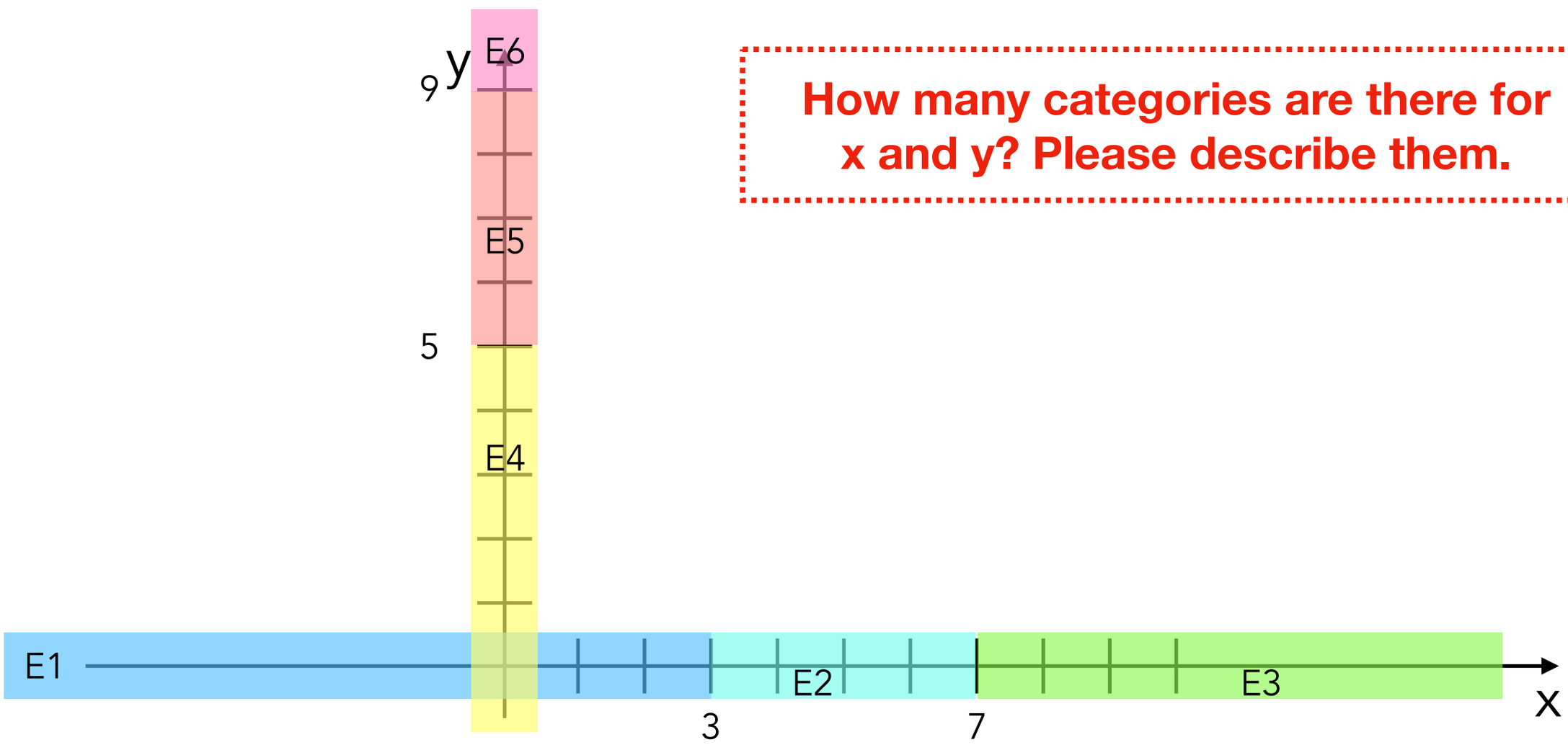
Consider an application that requires two integer inputs x and y. Each of these inputs is expected to lie in the following ranges: x > 3 and x <= 7 and y >= 5 and y <= 9.



Partition example *(recap)*

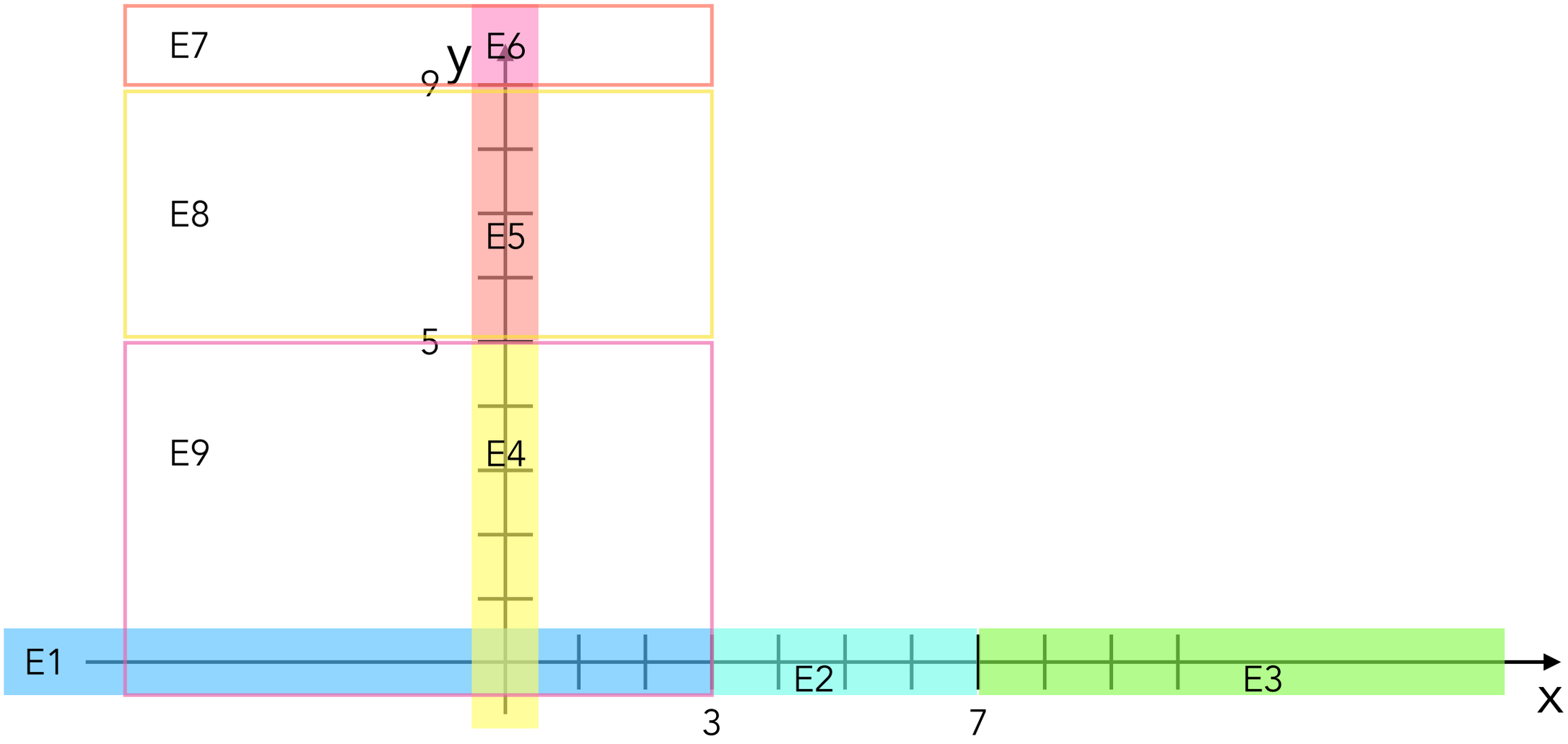
Consider an application that requires two integer inputs x and y. Each of these inputs is expected to lie in the following ranges: x > 3 and x <= 7 and y >= 5 and y <= 9.

How many categories are there for x and y? Please describe them.



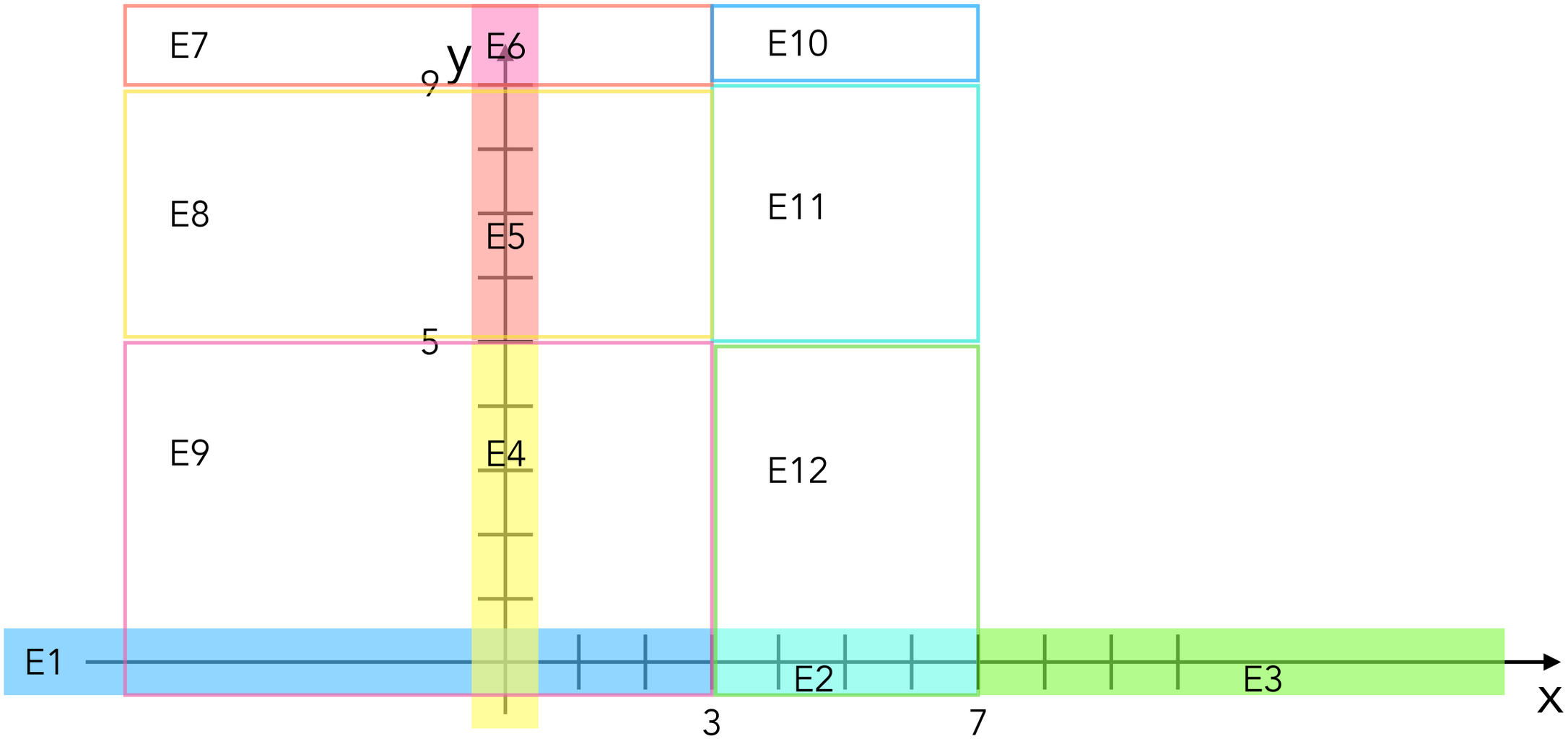
Partition example *(recap)*

Consider an application that requires two integer inputs x and y. Each of these inputs is expected to lie in the following ranges: x > 3 and x <= 7 and y >= 5 and y <= 9.



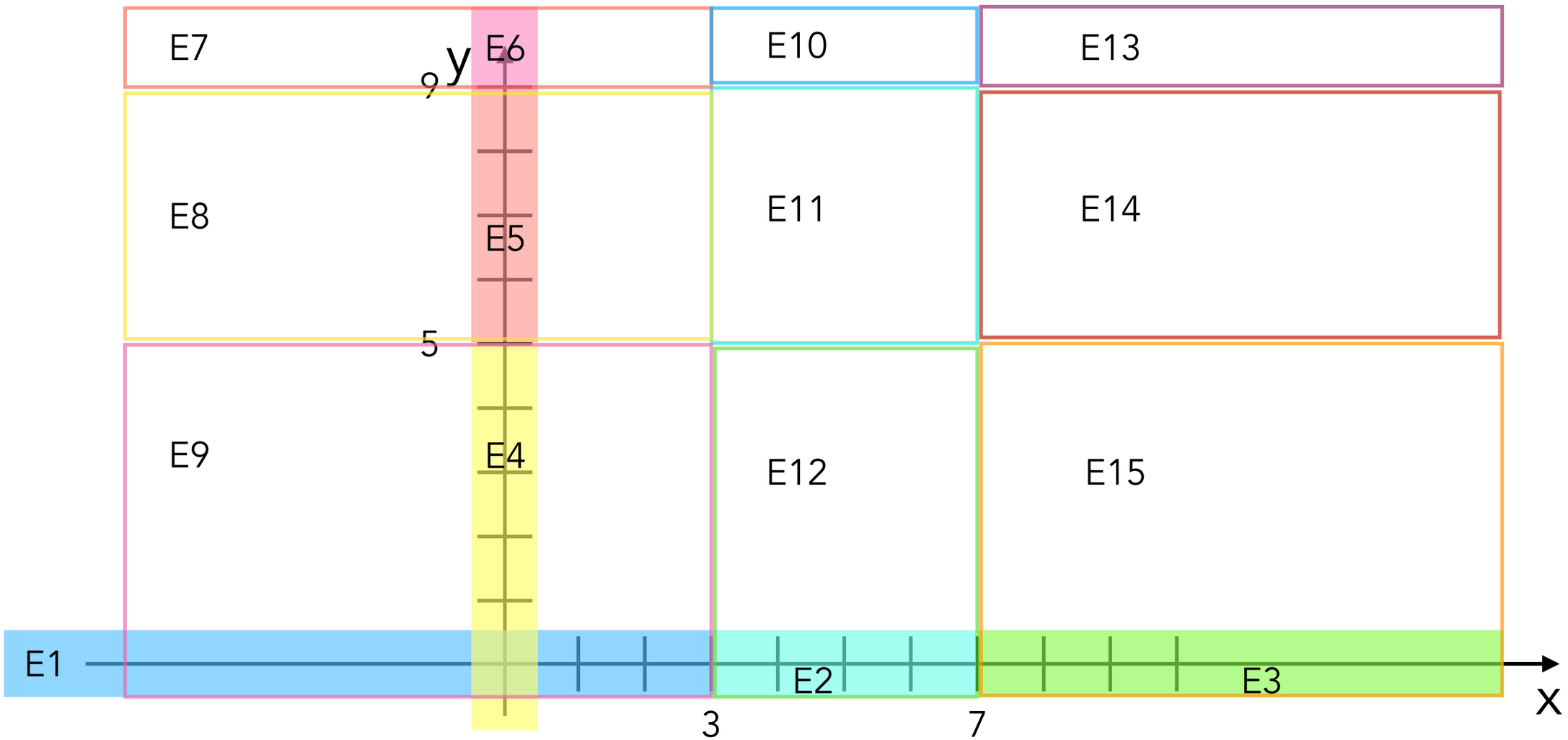
Partition example *(recap)*

Consider an application that requires two integer inputs x and y. Each of these inputs is expected to lie in the following ranges: x > 3 and x <= 7 and y >= 5 and y <= 9.



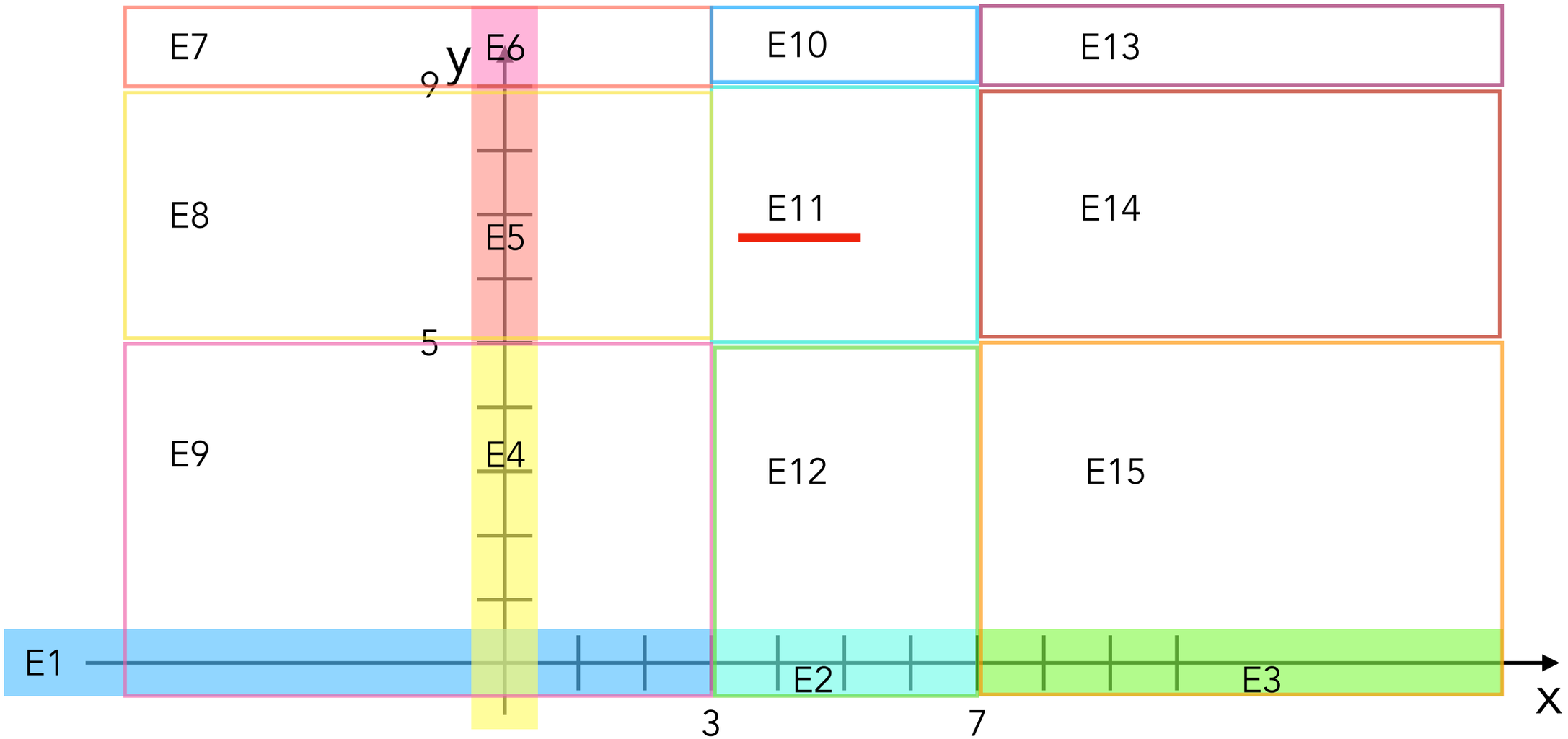
Partition example *(recap)*

Consider an application that requires two integer inputs x and y. Each of these inputs is expected to lie in the following ranges: x > 3 and x <= 7 and y >= 5 and y <= 9.



Partition example *(recap)*

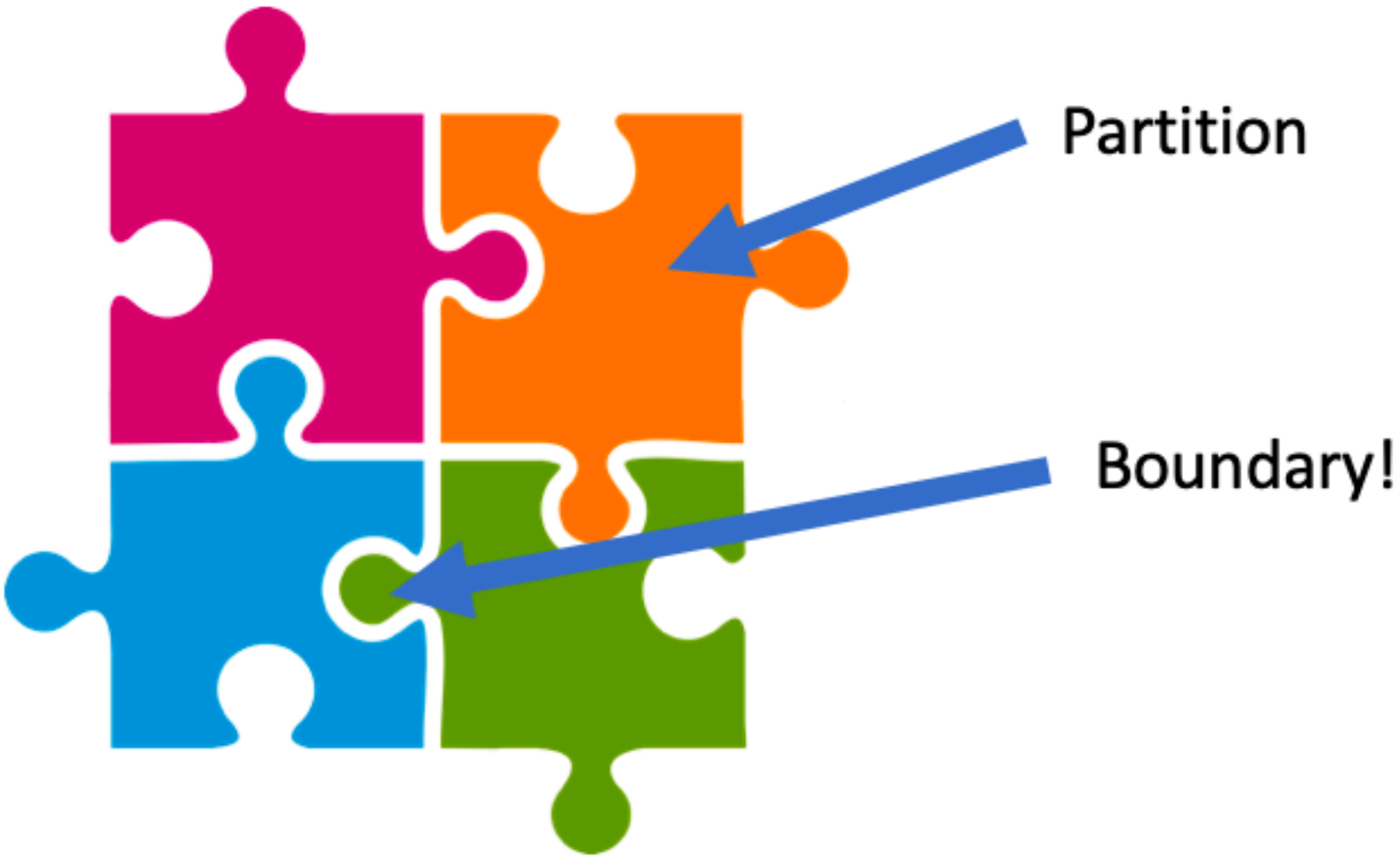
Consider an application that requires two integer inputs x and y. Each of these inputs is expected to lie in the following ranges: x > 3 and x <= 7 and y >= 5 and y <= 9.



Boundary Value Analysis

Boundary testing is the process of testing between extreme ends or boundaries between partitions of the input values.

Off-by-one mistakes are a common cause for *bugs* in software systems. As developers, we have all made mistakes such as using a "greater than" operator (>) where it had to be a "greater than or equal to" operator (>=). Interestingly, programs with such a *bug* tend to work well for most of the provided inputs. They fail, however, when the input is "near the boundary of condition".

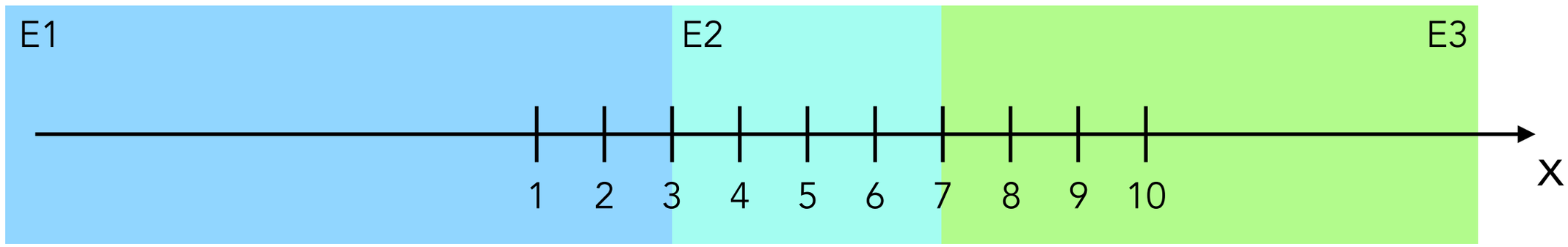

Partition
Boundary!

Boundary Value Analysis

When we devise classes, these have "close boundaries" with the other classes. In other words, if we keep performing small changes to an input that belongs to some partition (e.g., by adding +1 to it), at some point this input will belong to another class. The precise point where the input changes from one class to another is what we call a **boundary**. And this is precisely what boundary testing is about: to make the program behave correctly when inputs are near a boundary. More formally, we can find such boundaries by finding a pair of consecutive input values *[p_1,p_2]*, where *p1* belongs to partition *A*, and *p2* belongs to partition *B*.

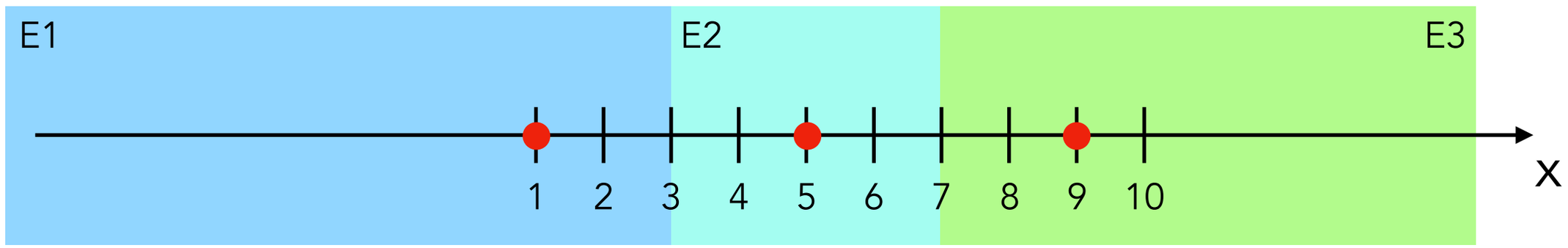
Boundary Value Analysis,
(but partitions first)

Consider an application that requires two integer inputs x and y. Each of these inputs is expected to lie in the following ranges: x > 3 and x <= 7 and y >= 5 and y <= 9.



Boundary Value Analysis,
(but partitions first)

Consider an application that requires two integer inputs x and y. Each of these inputs is expected to lie in the following ranges: x > 3 and x <= 7 and y >= 5 and y <= 9.



3 test cases, one per partition

test 1: x = 1 (E1), test 2: x = 5 (E2), test 3: x = 9 (E3)

Boundary Value Analysis

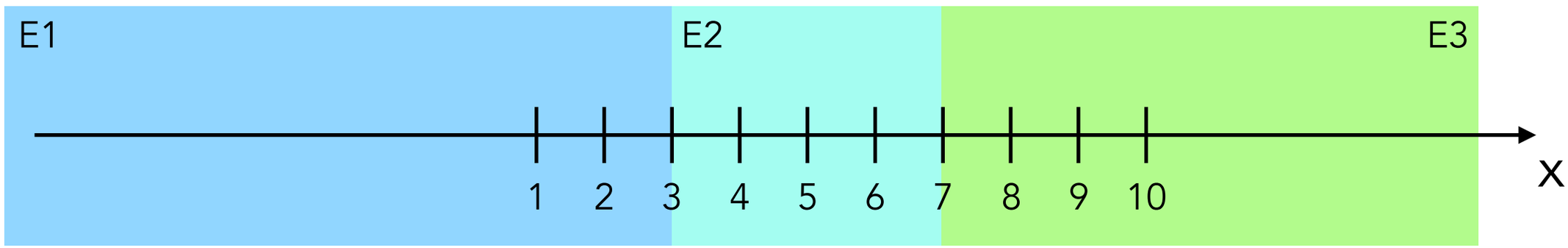
However, a tester who is aware of boundaries also devises test cases that explore the boundaries of the domain.

Before we see how to analyze boundaries, let us define some terminology:

- **On-point**: The value that is exactly on the boundary. This is the value we see in the condition itself.
- **Off-point**: The value that is closest to the boundary and that flips the condition's result. If the on-point makes the condition evaluate to true, the off point makes it evaluate to false and vice versa. Note that when dealing with equalities or inequalities (e.g. x == 6 or x != 6), there are two off-points; one in each direction.
- **In-points**: All values that make the condition evaluate to true.
- **Out-points**: All values that make the condition evaluate to false.

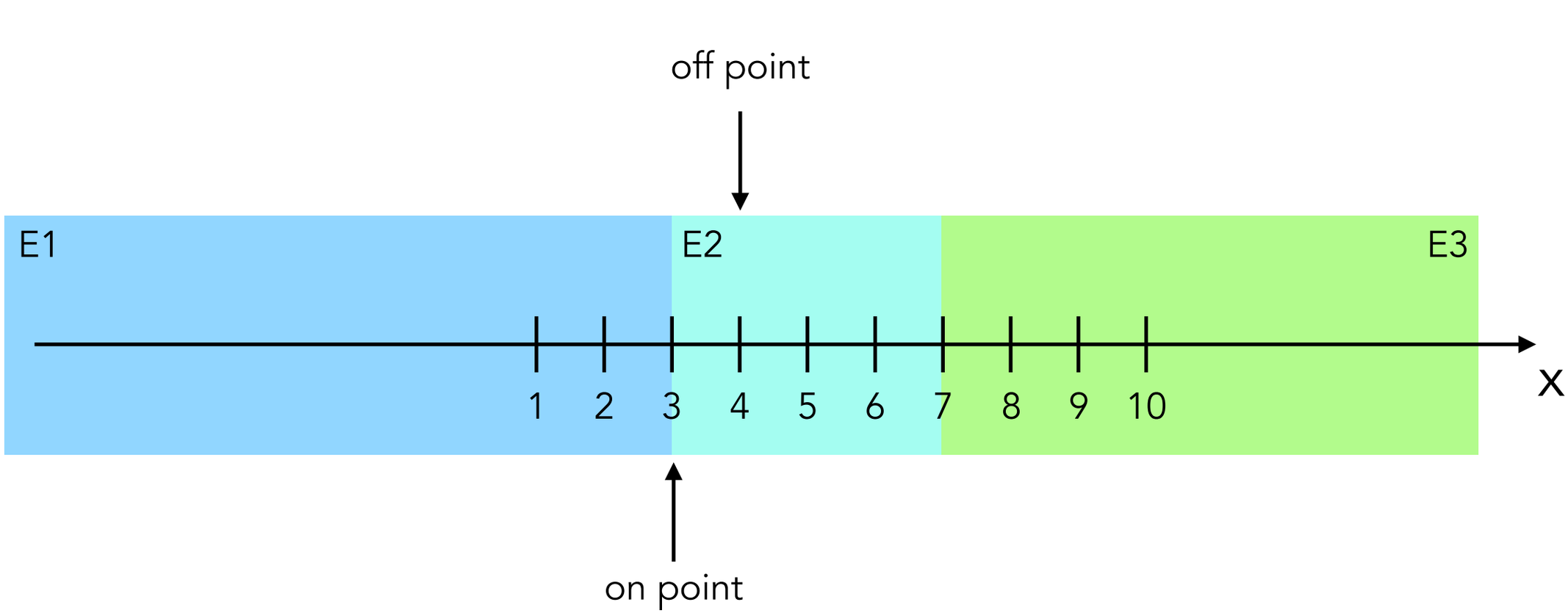
Boundary Value Analysis I

However, a tester who is aware of boundaries also devises test cases that explore the boundaries of the domain. Let us explore them: x > 3 and x <= 7.



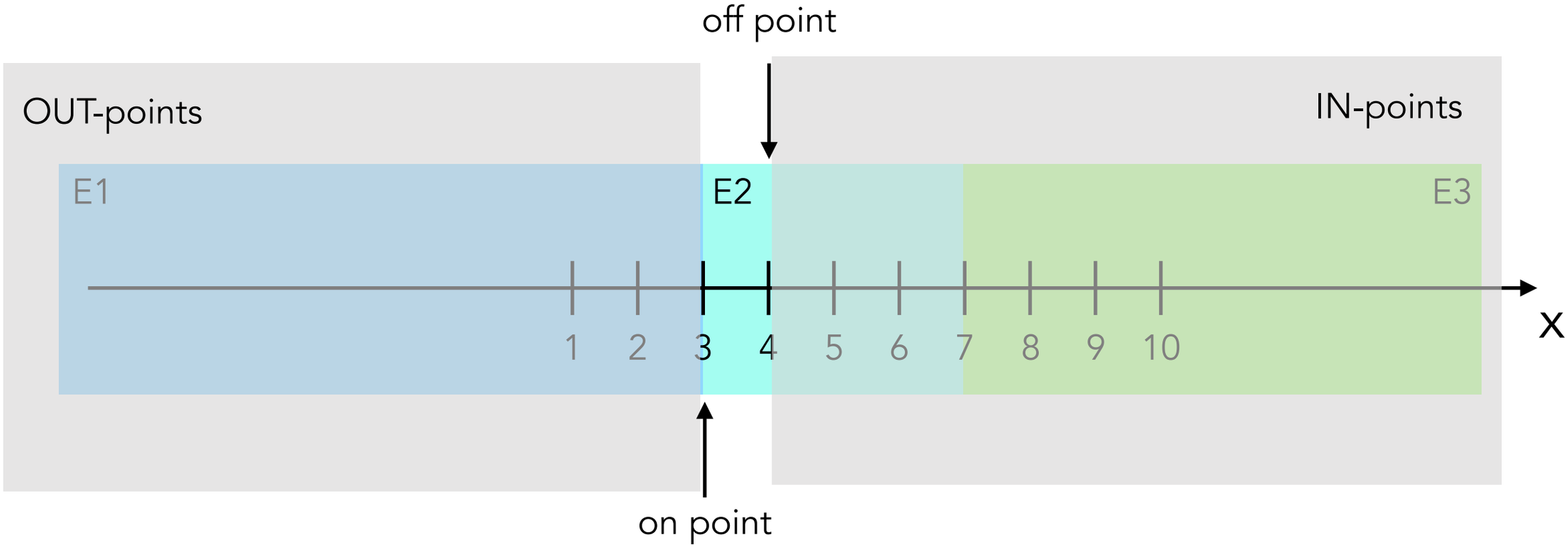
Boundary Value Analysis I

However, a tester who is aware of boundaries also devises test cases that explore the boundaries of the domain. Let us explore them: x > 3 and x <= 7.



Boundary Value Analysis I

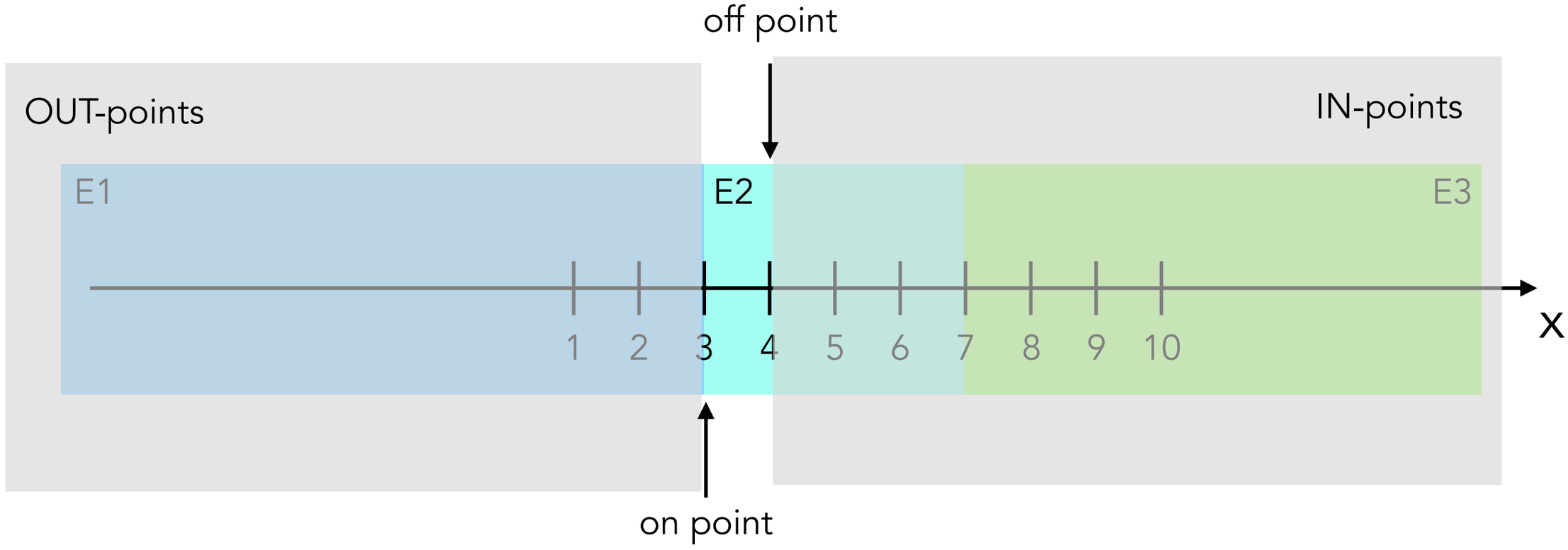
However, a tester who is aware of boundaries also devises test cases that explore the boundaries of the domain. Let us explore them: x > 3 and x <= 7.



Boundary Value Analysis I

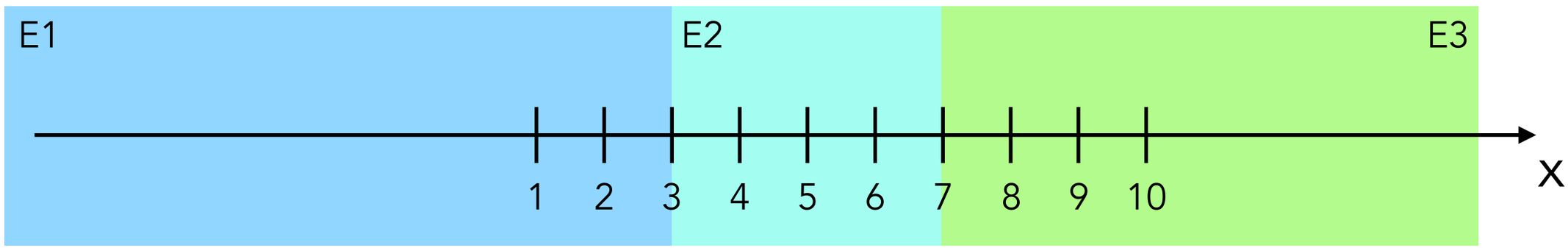
2 test cases

test 4: x = 3 (category E1), test 5: x = 4 (category E2)



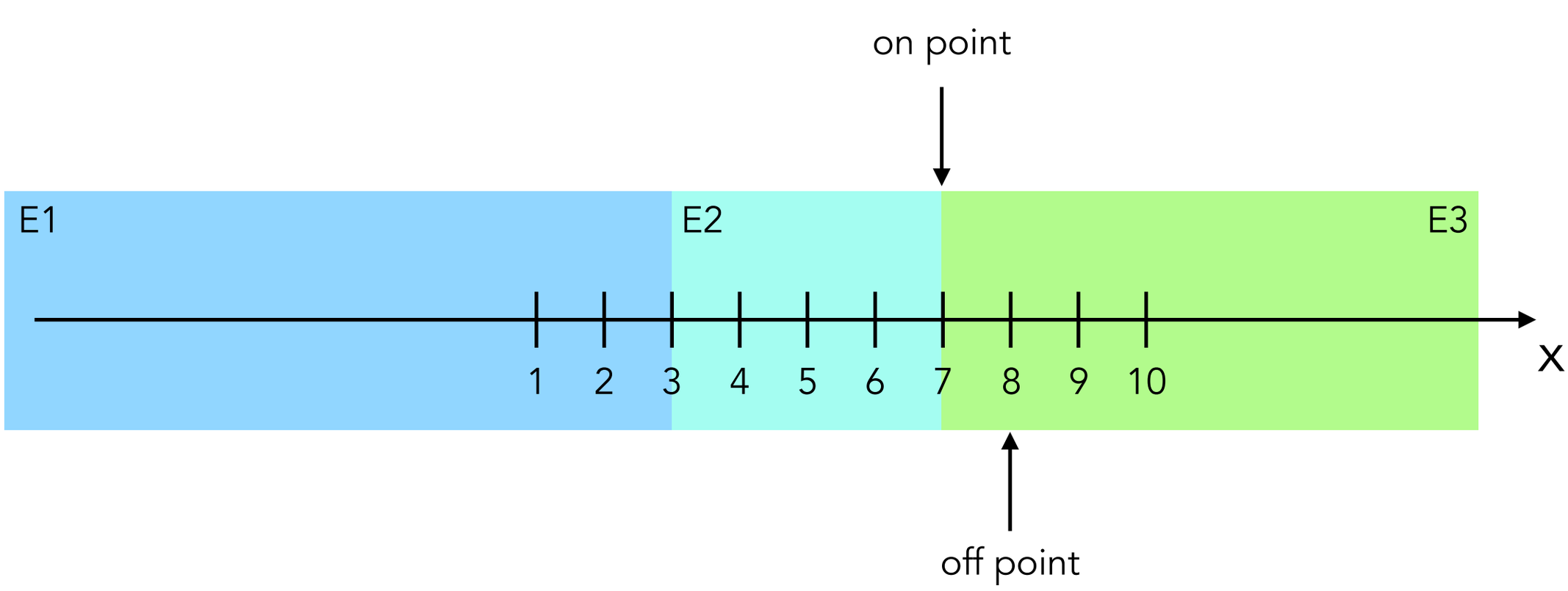
Boundary Value Analysis II

However, a tester who is aware of boundaries also devises test cases that explore the boundaries of the domain. Let us explore them: x > 3 and x <= 7.



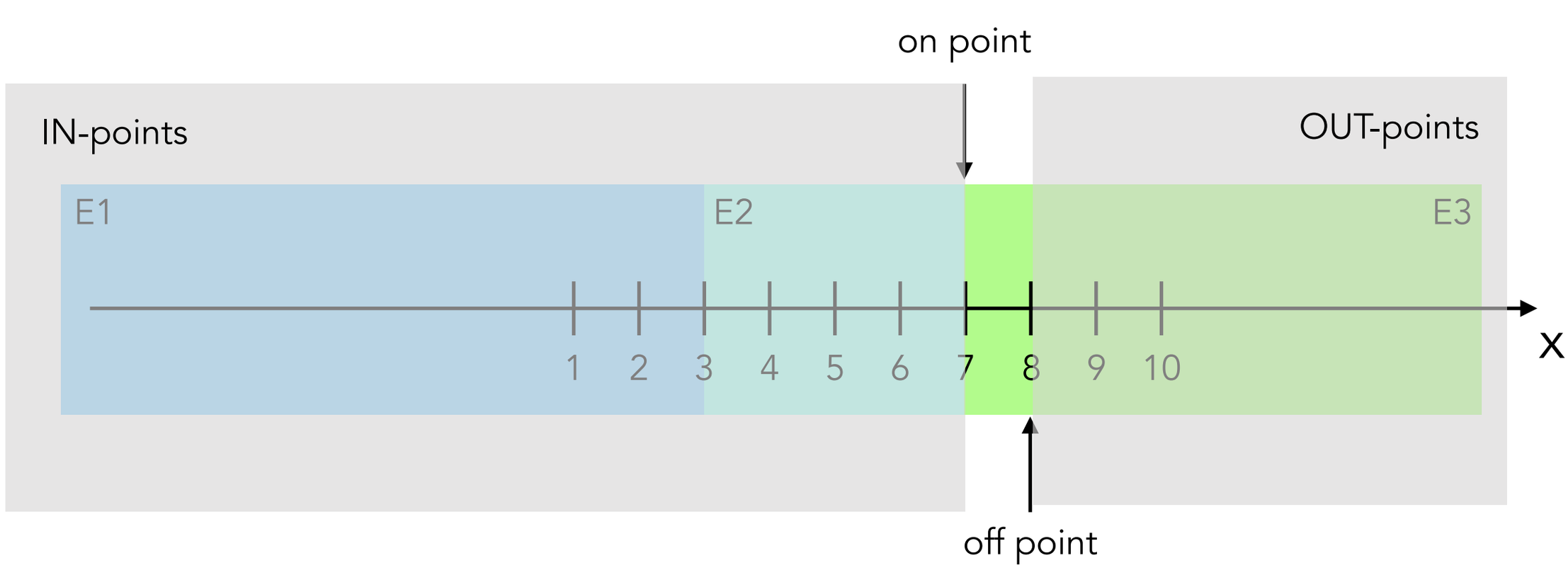
Boundary Value Analysis II

However, a tester who is aware of boundaries also devises test cases that explore the boundaries of the domain. Let us explore them: x > 3 and x <= 7.



Boundary Value Analysis II

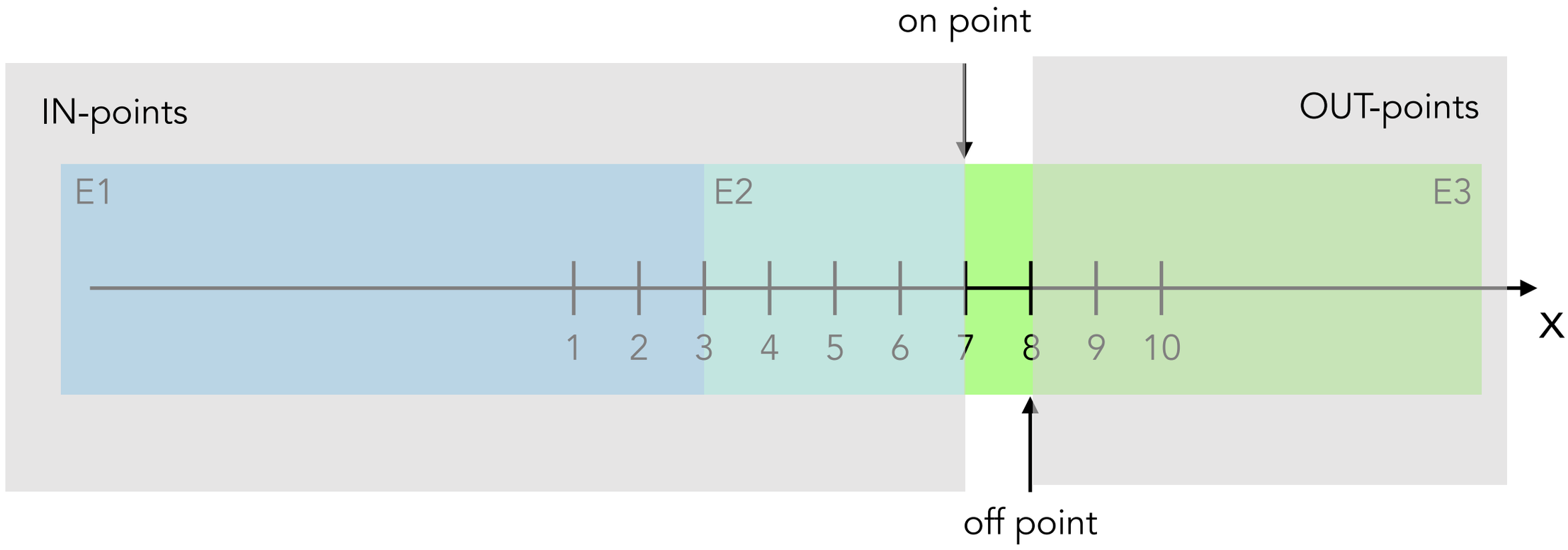
However, a tester who is aware of boundaries also devises test cases that explore the boundaries of the domain. Let us explore them: x > 3 and x <= 7.



Boundary Value Analysis II

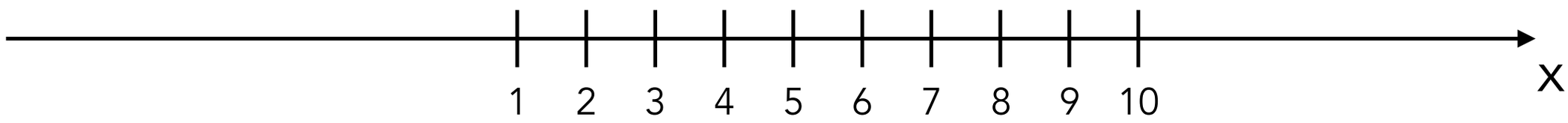
2 test cases

test 6: x = 7 (category E2), test 7: x = 8 (category E3)



Boundary Value Analysis

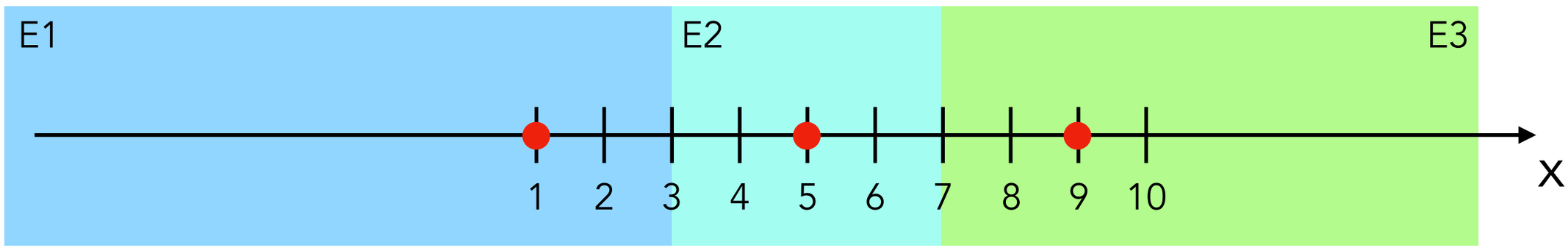
boundaries of the domain: x > 3 and x <= 7.



Boundary Value Analysis

3 test cases, one per partition

test 1: x = 1 (E1) test 2: x = 5 (E2) test 3: x = 9 (E3)



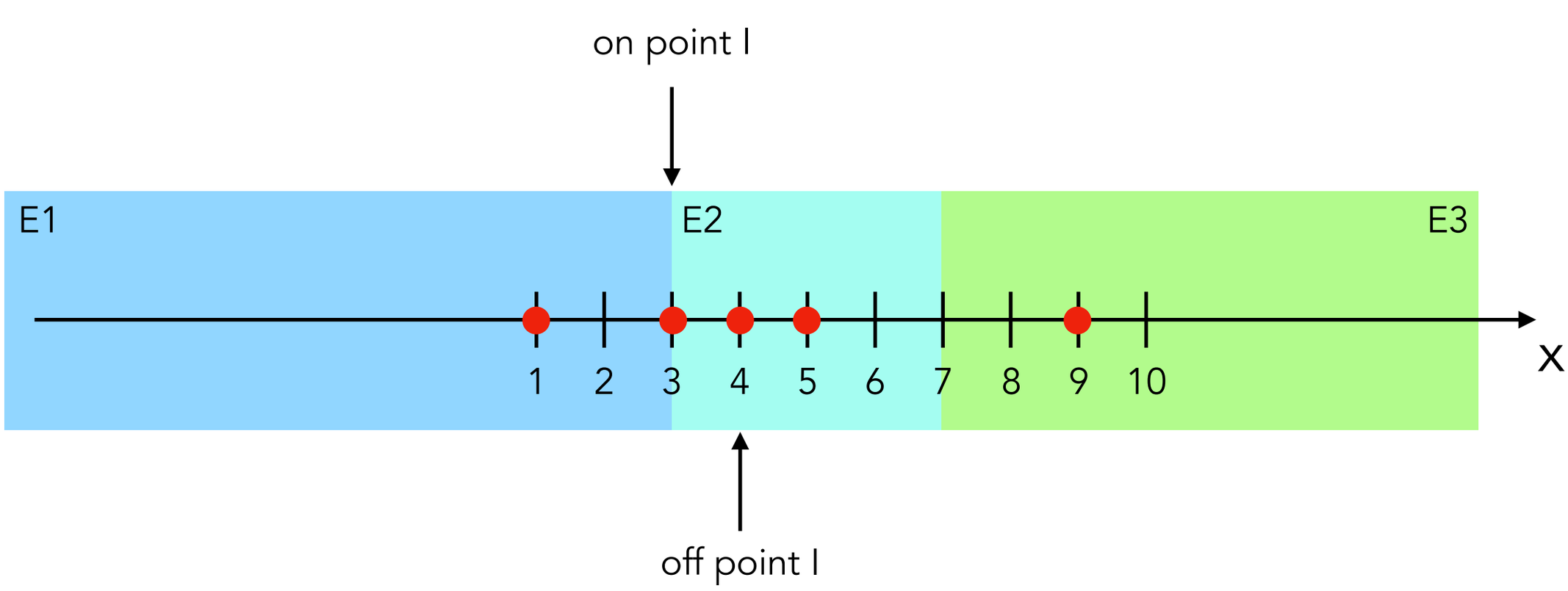
Boundary Value Analysis

3 test cases, one per partition

test 1: x = 1 (E1) test 2: x = 5 (E2) test 3: x = 9 (E3)

2 test cases (boundary I)

test 4: x = 3 (category E1) test 5: x = 4 (category E2)



Boundary Value Analysis

3 test cases, one per partition

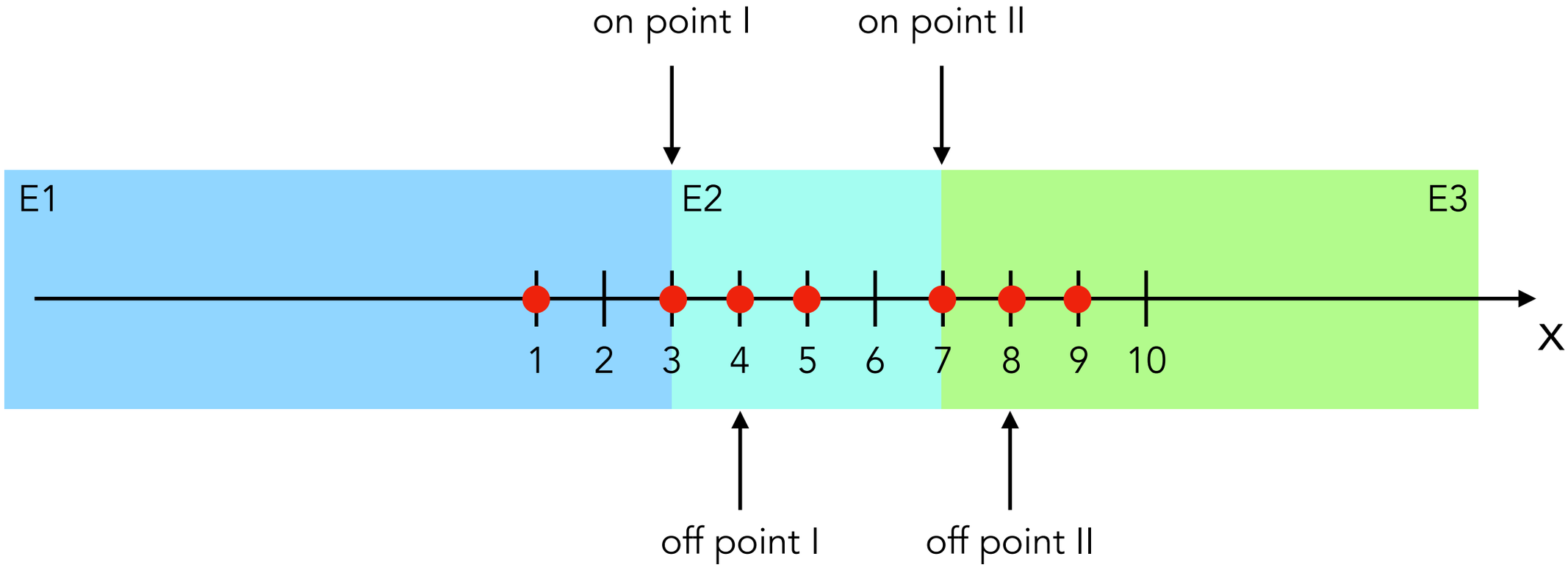
test 1: x = 1 (E1) test 2: x = 5 (E2) test 3: x = 9 (E3)

2 test cases (boundary I)

test 4: x = 3 (category E1) test 5: x = 4 (category E2)

2 test cases (boundary II)

test 6: x = 7 (category E2) test 7: x = 8 (category E3)



Partitions + Boundaries

In practice, testers combine equivalent class analysis and boundary testing, which is called *domain testing*. The following strategy is commonly suggested when applying *domain testing*:

1. Read the requirements.
2. Identify the input and output variables in play, together with their types, and their ranges.
3. Identify the dependencies (or independence) among input variables, and how input variables influence the output variable.
4. Perform equivalent class analysis (valid and invalid classes).
5. Explore the boundaries of these classes.
6. Think of a strategy to derive test cases, focusing on minimizing the costs while maximizing diversity and perhaps fault detection capability.
7. Generate a set of test cases that should be executed against the system under test.

Random testing

Input values / Actions are randomly generated.

Advantages:

- Good for finding system crashes.
- No effort in generating test cases.
- Independent of updates.
- Increase confidence on the software when running several hours without finding errors.
- “Easy” to implement.

Disadvantages:

- Not good for finding other kinds of errors besides system crashes.
- Difficult to reproduce the errors (repeat test cases / sequence of inputs).
- Unpredictable.
- May not cover special cases that are discovered by more advance techniques.

Random testing

- **Monkey testing**: a technique where the user tests a computer program or a system by providing random inputs/actions and checking the behavior, or seeing whether the program or system will crash.
Note: monkey testing is included in Android Studio.

- **Fuzz testing**: an automated software testing technique that involves providing invalid, unexpected, or random data as inputs to a computer program.

References

- Ilene Burnstein. “Practical software testing a process-oriented approach” (chapter 4). 2003.
- Jorgensen, Paul C. “Software Testing: A Craftsman’s Approach” 4th edition (chapter 5 and 6). 2014.
- T. J. Ostrand, M J Balcer. “The category-partition method for specifying and generating fuctional tests”, 1988 (https://doi.org/10.1145/62959.62964).
- Jeng, B., & Weyuker, E. J. (1994). A simplified domain-testing strategy. ACM Transactions on Software Engineering and Methodology (TOSEM).
- Kaner, Cem, Sowmya Padmanabhan, and Douglas Hoffman. The Domain Testing Workbook. Context Driven Press, 2013.
- Kaner, Cem. What Is a Good Test Case?, 2003. http://testingeducation.org/BBST/testdesign/Kaner_GoodTestCase.pdf