CMPT 473
Software Testing, Reliability and Security

Input Space Partitioning

Nick Sumner

Recall

- Testing involves running software and comparing observed behavior against expected behavior
  - Select an input, look at the output

Recall

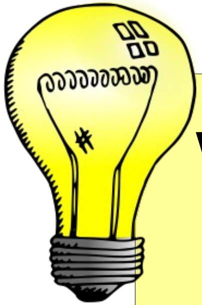
- Testing involves running software and comparing observed behavior against expected behavior
  - Select an input, look at the output
- Problem: The ***input domain*** is infinite or pragmatically infinite.

Recall

- Testing involves running software and comparing observed behavior against expected behavior
  - Select an input, look at the output
- Problem: The ***input domain*** is infinite or pragmatically infinite.
- Test suites select a finite subset of inputs that help measure quality

Recall

- Testing involves running software and comparing observed behavior against expected behavior
  - Select an input, look at the output
- Problem: The ***input domain*** is infinite or pragmatically infinite.
- Test suites select a finite subset of inputs that help measure quality



We can take a direct approach:
Focus on the input!

Input Space Partitioning

- ***Input Space Partitioning*** (AKA Partition Testing)
  - Divide (partition) the set of possible inputs into *equivalence classes*
  - Test one input from each class

Input Space Partitioning

- ***Input Space Partitioning*** (AKA Partition Testing)
  - Divide (partition) the set of possible inputs into *equivalence classes*
  - Test one input from each class

What does this show?
What does it *not* show?

Input Space Partitioning

- ***Input Space Partitioning*** (AKA Partition Testing)
  - Divide (partition) the set of possible inputs into *equivalence classes*
  - Test one input from each class

e.g. abs(x)

Input Domain: …, -3, -2, -1, 0, 1, 2, 3, …

How many tests if done exhaustively?

Input Space Partitioning

- ***Input Space Partitioning*** (AKA Partition Testing)
  - Divide (partition) the set of possible inputs into *equivalence classes*
  - Test one input from each class

e.g. abs(x)

Input Domain: …, -3, -2, -1, 0, 1, 2, 3, …

Partitions: …, -3, -2, -1, 0, 1, 2,
3, …

Input Space Partitioning

- ***Input Space Partitioning*** (AKA Partition Testing)
  - Divide (partition) the set of possible inputs into *equivalence classes*
  - Test one input from each class

e.g. abs(x)

Input Domain: …, -3, -2, -1, 0, 1, 2, 3, …

Partitions: …, -3, -2, -1, 0, 1, 2,
3, …

What might reasonable partitions be?

Input Space Partitioning

- ***Input Space Partitioning*** (AKA Partition Testing)
  - Divide (partition) the set of possible inputs into *equivalence classes*
  - Test one input from each class

e.g. abs(x)

Input Domain: ..., -3, -2, -1, 0, 1, 2, 3, ...

Partitions: ..., -3, -2, -1, 0, 1, 2, 3, ...

What might reasonable partitions be?

Input Space Partitioning

- ***Input Space Partitioning*** (AKA Partition Testing)
  - Divide (partition) the set of possible inputs into *equivalence classes*
  - Test one input from each class

e.g. abs(x)

Input Domain: …, -3, -2, -1, 0, 1, 2, 3, …

Partitions: …, -3, -2, -1, 0, 1, 2, 3, …

What might reasonable partitions be?

How many tests for the partitions?

Input Space Partitioning

1) Identify the component

abs(x)

Input Space Partitioning

1) Identify the component

- Whole program
- Module
- Class
- Function

abs(x)

Input Space Partitioning

1) Identify the component
2) Identify the inputs / parameters

abs(x) | $x \in \mathbb{Z}$

Input Space Partitioning

1) Identify the component
2) Identify the inputs / parameters

- Function/method parameters
- Object state
- Global variables
- File contents
- User provided inputs
- …

abs(x)

$x \in \mathbb{Z}$

Input Space Partitioning

1) Identify the component
2) Identify the inputs / parameters
3) Develop an ***input domain model*** for input characteristics*

A way of partitioning the input space

abs(x)

$x \in \mathbb{Z}$
$x < 0$
$x = 0$
$x > 0$

Input Space Partitioning

1) Identify the component
2) Identify the inputs / parameters
3) Develop an input domain model for input characteristics
4) Refine combinations with constraints*

abs(x)	$x \in \mathbb{Z}$
	$x < 0$
	$x = 0$
	$x > 0$

Input Space Partitioning

1) Identify the component
2) Identify the inputs / parameters
3) Develop an input domain model for input characteristics
4) Refine combinations with constraints
5) Generate combinations / *test frames**

	$x \in \mathbb{Z}$	
abs(x)	$x < 0$	frame 1: $x < 0$
	$x = 0$	frame 2: $x = 0$
	$x > 0$	frame 3: $x > 0$

Input Space Partitioning

1) Identify the component
2) Identify the inputs / parameters
3) Develop an input domain model for input characteristics
4) Refine combinations with constraints
5) Generate combinations / test frames
6) Select concrete inputs*

abs(x)	$x \in \mathbb{Z}$		
	$x < 0$	frame 1: $x < 0$	test 1: abs(-3) = 3
	$x = 0$	frame 2: $x = 0$	test 2: abs(0) = 0
	$x > 0$	frame 3: $x > 0$	test 3: abs(7) = 7

Input Space Partitioning

1) Identify the component
2) Identify the inputs / parameters
3) Develop an input domain model for input characteristics
4) Refine c

* some parts are more subtle than they appear

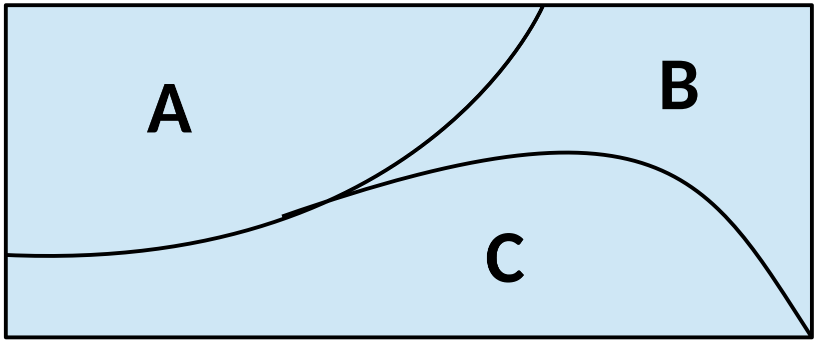
5) Generate combinations / test frames
6) Select concrete inputs

abs(x)	$x \in \mathbb{Z}$		
	$x < 0$	frame 1: $x < 0$	test 1: abs(-3) = 3
	$x = 0$	frame 2: $x = 0$	test 2: abs(0) = 0
	$x > 0$	frame 3: $x > 0$	test 3: abs(7) = 7

Input Domain Modeling

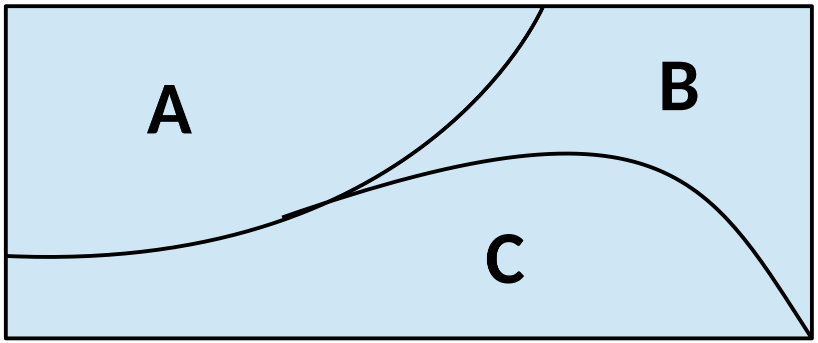
- ***Partition*** a domain D on ***characteristics***

D =



Input Domain Modeling

- ***Partition*** a domain D on ***characteristics***
- Must satisfy 2 criteria:
  - Disjoint: $\mathbf{P_i \cap P_j = \varnothing}$
  - Cover: $\mathbf{A \cup B \cup C = D}$

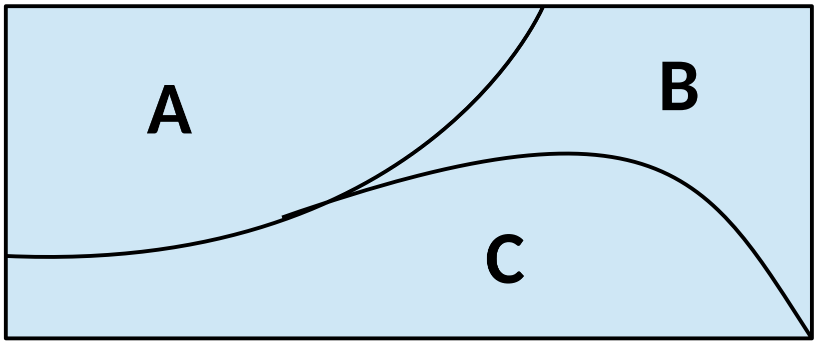
D = 

Input Domain Modeling

- ***Partition*** a domain D on ***characteristics***
- Must satisfy 2 criteria:
  - Disjoint: $\mathbf{P_i \cap P_j = \varnothing}$
  - Cover: $\mathbf{A \cup B \cup C = D}$

What do these criteria intuitively provide?

D =



Input Domain Modeling

- Characteristics of the input space drive the process

Input Domain Modeling

- Characteristics of the input space drive the process
  - Can come from many sources:
    requirements, types, formal specifications, …

Input Domain Modeling

- Characteristics of the input space drive the process
  - Can come from many sources:
    requirements, types, formal specifications, …
  - May be boolean *or* non-boolean

Input Domain Modeling

- Characteristics of the input space drive the process
  - Can come from many sources:
    requirements, types, formal specifications, …
  - May be boolean *or* non-boolean

List **s** is sorted ascending (boolean)

Input Domain Modeling

- Characteristics of the input space drive the process
  - Can come from many sources:
    requirements, types, formal specifications, …
  - May be boolean *or* non-boolean

List **s** is sorted ascending (boolean)

Sortedness of **s** (non-boolean)
- sorted ascending
- sorted descending
- unsorted

?

Input Domain Modeling

- Characteristics of the input space drive the process
  - Can come from many sources:
    requirements, types, formal specifications, …
  - May be boolean *or* non-boolean

List **s** is sorted ascending (boolean)

Sortedness of **s** (non-boolean)

- $|s| < 2$
- $\exists k. \forall x \in s, x = k$ ($|s| \geq 2$)
- sorted ascending ($|s| \geq 2, \nexists k\ldots$)
- sorted descending ($|s| \geq 2, \nexists k\ldots$)
- unsorted ($|s| \geq 2, \nexists k\ldots$)

Input Domain Modeling

- Characteristics of the input space drive the process
  - Can come from many sources:
    requirements, types, formal specifications, …
  - May be boolean *or* non-boolean

List **s** is sorted ascending (boolean)

Sortedness of **s** (non-boolean)

- $|s| < 2$
- $\exists k. \forall x \in s, x = k$ ($|s| \geq 2$)
- sorted ascending ($|s| \geq 2, \nexists k\ldots$)
- sorted descending ($|s| \geq 2, \nexists k\ldots$)
- unsorted ($|s| \geq 2, \nexists k\ldots$)

Reference **x** is null (boolean)

Input Domain Modeling

- Characteristics of the input space drive the process
  - Can come from many sources:
    requirements, types, formal specifications, …
  - May be boolean *or* non-boolean

List **s** is sorted ascending (boolean)

Sortedness of **s** (non-boolean)
- $|s| < 2$
- $\exists k. \forall x \in s, x = k$ ($|s| \geq 2$)
- sorted ascending ($|s| \geq 2, \nexists k\ldots$)
- sorted descending ($|s| \geq 2, \nexists k\ldots$)
- unsorted ($|s| \geq 2, \nexists k\ldots$)

Reference **x** is null (boolean)

Size of string **s** (non-boolean)
- 0
- 1
- 2
- $2 < |s| < 1024$
- $|s| \geq 1024$

Input Domain Modeling: Partitioning Practice

- Suppose we have

```
classifyParallelogram(p1: Parallelogram) -> Kind
```

  characteristic: “The kind/subtype of parallelogram”

Input Domain Modeling: Partitioning Practice

- Suppose we have

```
classifyParallelogram(p1: Parallelogram) -> Kind
```

  characteristic: “The kind/subtype of parallelogram”
  - How can we partition based on this characteristic?
  - What problems might arise?

Input Domain Modeling: Partitioning Practice

- Suppose we have

```
classifyParallelogram(p1: Parallelogram) -> Kind
```

  characteristic: “The kind/subtype of parallelogram”
  - How can we partition based on this characteristic?
  - What problems might arise?

- In class exercise:
  Suppose we have

```
classifyTriangle(s1: int, s2: int, s3: int) -> Kind
```

  what should partitions be?

Input Domain Modeling – Choosing Characteristics

2 main approaches:

Input Domain Modeling – Choosing Characteristics

2 main approaches:

- ***Interface*** based
  - Guided directly by identified parameters & domains
  - Simple
  - Automatable

Input Domain Modeling – Choosing Characteristics

2 main approaches:

- *Interface* based
  - Guided directly by identified parameters & domains
  - Simple
  - Automatable
- ***Functionality***/Requirements based
  - Derived from expected input/output relationship by spec.
  - Requires more design & more thought
  - May be better (smaller, goal oriented, ...)

Input Domain Modeling – Interface Based

- Consider parameters individually
  - Examine their types/domains
  - Ignore relationships & dependencies

Input Domain Modeling – Interface Based

- Consider parameters individually
  - Examine their types/domains
  - Ignore relationships & dependencies

How does this apply to our triangle classifier?

```
classifyTriangle(s1: int, s2: int, s3: int) -> Kind
```

Input Domain Modeling – Interface Based

- Consider parameters individually
  - Examine their types/domains
  - Ignore relationships & dependencies

How does this apply to our triangle classifier?

```
classifyTriangle(s1: int, s2: int, s3: int) -> Kind
```

$s1 \in \mathbb{Z}$	$s2 \in \mathbb{Z}$	$s3 \in \mathbb{Z}$
$s1 < 0$	$s2 < 0$	$s3 < 0$
$s1 = 0$	$s2 = 0$	$s3 = 0$
$s1 > 0$	$s2 > 0$	$s3 > 0$

We will revisit how this is good / bad

Input Domain Modeling – Functionality Based

- Characteristics correspond to behaviors in the requirements
  - Includes knowledge from the problem domain
  - Accounts for relationships between parameters

Input Domain Modeling – Functionality Based

- Characteristics correspond to behaviors in the requirements
  - Includes knowledge from the problem domain
  - Accounts for relationships between parameters
  - Same parameter may play a role in multiple characteristics
    - Need to reason about constraints & conflicts!

Input Domain Modeling – Functionality Based

- Characteristics correspond to behaviors in the requirements
  - Includes knowledge from the problem domain
  - Accounts for relat
  - Same parameter r teristics
    - Need to reason about constraints & conflicts!

How does this apply to our triangle classifier?

```
classifyTriangle(s1: int, s2: int, s3: int) -> Kind
```

Input Domain Modeling – Functionality Based

- Characteristics correspond to behaviors in the requirements
  - Includes knowledge from the problem domain
  - Accounts for relat
  - Same parameter r teristics
    - Need to reason about constraints & conflicts!

How does this apply to our triangle classifier?

```
classifyTriangle(s1: int, s2: int, s3: int) -> Kind
```

Kind
Scalene
Isosceles - Equilateral
Equilateral
Invalid

Input Domain Modeling

Common sources for characteristics:

Input Domain Modeling

Common sources for characteristics:

- Component specifications
  - Preconditions
  - Postconditions

Input Domain Modeling

Common sources for characteristics:

- Component specifications
  - Preconditions
  - Postconditions
- Domain knowledge
  - Relationships to special values
  - Relationships between variables

Input Domain Modeling

Common sources for characteristics:

- Component specifications
  - Preconditions
  - Postconditions
- Domain knowledge
  - Relationships to special values
  - Relationships between variables
- Checklists [Langr, Hunt, Thomas]
  - Correctness
  - Ordering
  - Range
  - Reference
  - Existence
  - Cardinality
  - Time

Generating Combinations / Test Frames

- We may have multiple ways / dimensions of partitioning.
- We can plan our tests by creating ***test frames*** that identify the combinations of partitions used in each abstract / planned test.

Generating Combinations / Test Frames

- We may have multiple ways / dimensions of partitioning.
- We can plan our tests by creating ***test frames*** that identify the combinations of partitions used in each abstract / planned test.

```
classifyTriangle(s1: int, s2: int, s3: int) -> Kind
```

$s1 \in \mathbb{Z}$	$s2 \in \mathbb{Z}$	$s3 \in \mathbb{Z}$
$s1 < 0$	$s2 < 0$	$s3 < 0$
$s1 = 0$	$s2 = 0$	$s3 = 0$
$s1 > 0$	$s2 > 0$	$s3 > 0$

Generating Combinations / Test Frames

- We may have multiple ways / dimensions of partitioning.
- We can plan our tests by creating ***test frames*** that identify the combinations of partitions used in each abstract / planned test.

```
classifyTriangle(s1: int, s2: int, s3: int) -> Kind
```

$s1 \in \mathbb{Z}$	$s2 \in \mathbb{Z}$	$s3 \in \mathbb{Z}$
$s1 < 0$	$s2 < 0$	$s3 < 0$
$s1 = 0$	$s2 = 0$	$s3 = 0$
$s1 > 0$	$s2 > 0$	$s3 > 0$

We need to choose a value for each side.

Generating Combinations / Test Frames

- We may have multiple ways / dimensions of partitioning.
- We can plan our tests by creating ***test frames*** that identify the combinations of partitions used in each abstract / planned test.

```
classifyTriangle(s1: int, s2: int, s3: int) -> Kind
```

$s1 \in \mathbb{Z}$	$s2 \in \mathbb{Z}$	$s3 \in \mathbb{Z}$
$s1 < 0$	$s2 < 0$	$s3 < 0$
$s1 = 0$	$s2 = 0$	$s3 = 0$
$s1 > 0$	$s2 > 0$	$s3 > 0$

Frame 1	Frame 2	Frame 3	Frame 4	Frame 5	Frame 6	Frame ...
S1: s1 < 0	S1: s1 = 0	S1: s1 < 0	S1: s1 < 0	S1: s1 > 0	S1: s1 < 0	S1: ...
S2: s2 < 0	S2: s2 < 0	S2: s2 = 0	S2: s2 < 0	S2: s2 < 0	S2: s2 > 0	S2: ...
S3: s3 < 0	S3: s3 < 0	S3: s3 < 0	S3: s3 = 0	S3: s3 < 0	S3: s3 < 0	S3: ...

Practical Issues: Interface Based

```
classifyTriangle(s1: int, s2: int, s3: int) -> Kind
```

$s1 \in \mathbb{Z}$	$s2 \in \mathbb{Z}$	$s3 \in \mathbb{Z}$
$s1 < 0$	$s2 < 0$	$s3 < 0$
$s1 = 0$	$s2 = 0$	$s3 = 0$
$s1 > 0$	$s2 > 0$	$s3 > 0$

How many tests does this create?

Practical Issues: Interface Based

```
classifyTriangle(s1: int, s2: int, s3: int) -> Kind
```

$s1 \in \mathbb{Z}$	$s2 \in \mathbb{Z}$	$s3 \in \mathbb{Z}$
$s1 < 0$	$s2 < 0$	$s3 < 0$
$s1 = 0$	$s2 = 0$	$s3 = 0$
$s1 > 0$	$s2 > 0$	$s3 > 0$

How many tests does this create?

What will this test well?
What won't this test well?

Practical Issues: Interface Based

```
classifyTriangle(s1: int, s2: int, s3: int) -> Kind
```

$s1 \in \mathbb{Z}$	$s2 \in \mathbb{Z}$	$s3 \in \mathbb{Z}$
$s1 < 0$	$s2 < 0$	$s3 < 0$
$s1 = 0$	$s2 = 0$	$s3 = 0$
$s1 = 1$	$s2 = 1$	$s3 = 1$
$s1 > 1$	$s2 > 1$	$s3 > 1$

$\{sn > 0\} \rightarrow \{sn = 1\}, \{sn > 1\}$

Practical Issues: Interface Based

```
classifyTriangle(s1: int, s2: int, s3: int) -> Kind
```

$s1 \in \mathbb{Z}$	$s2 \in \mathbb{Z}$	$s3 \in \mathbb{Z}$
$s1 < 0$	$s2 < 0$	$s3 < 0$
$s1 = 0$	$s2 = 0$	$s3 = 0$
$s1 = 1$	$s2 = 1$	$s3 = 1$
$s1 > 1$	$s2 > 1$	$s3 > 1$

How many tests now?

Practical Issues: Interface Based

```
classifyTriangle(s1: int, s2: int, s3: int) -> Kind
```

$s1 \in \mathbb{Z}$	$s2 \in \mathbb{Z}$	$s3 \in \mathbb{Z}$
$s1 < 0$	$s2 < 0$	$s3 < 0$
$s1 = 0$	$s2 = 0$	$s3 = 0$
$s1 = 1$	$s2 = 1$	$s3 = 1$
$s1 > 1$	$s2 > 1$	$s3 > 1$

How many tests now?

Is it still disjoint? Complete?

Practical Issues: Interface Based

```
classifyTriangle(s1: int, s2: int, s3: int) -> Kind
```

$s1 \in \mathbb{Z}$	$s2 \in \mathbb{Z}$	$s3 \in \mathbb{Z}$
$s1 < 0$	$s2 < 0$	$s3 < 0$
$s1 = 0$	$s2 = 0$	$s3 = 0$
$s1 = 1$	$s2 = 1$	$s3 = 1$
$s1 > 1$	$s2 > 1$	$s3 > 1$

How many tests now?

Is it still disjoint? Complete?

What does it test well? Not well?

Practical Issues: Functionality Based

```
classifyTriangle(s1: int, s2: int, s3: int) -> Kind
```

Kind
Scalene
Isosceles - Equilateral
Equilateral
Invalid

Are there alternatives?

Practical Issues: Functionality Based

```
classifyTriangle(s1: int, s2: int, s3: int) -> Kind
```

Kind
Scalene
Isosceles - Equilateral
Equilateral
Invalid

Is equilateral?	×	Is isosceles?	×	Is scalene?
True		True		True
False		False		False

Practical Issues: Functionality Based

```
classifyTriangle(s1: int, s2: int, s3: int) -> Kind
```

Kind
Scalene
Isosceles - Equilateral
Equilateral
Invalid

Is equilateral?	×	Is isosceles?	×	Is scalene?
True		True		True
False		False		False

Frame 1	Frame 2	Frame ...
Is Eq: False	Is Eq: True	Is Eq: ...
Is Iso: False	Is Iso: False	Is Iso: ...
Is Sc: False	Is Sc: False	Is Sc: ...

Why might you use it?

Another Functionality Based Example

- Suppose we have a simple function:

```
symmetricDifference(s1: list, s2: list) -> list
```

  that returns all elements unique to either **`s1`** or **`s2`**.

Another Functionality Based Example

- Suppose we have a simple function:

```
symmetricDifference(s1: list, s2: list) -> list
```

that returns all elements unique to either **`s1`** or **`s2`**.

- Try to construct a functionality based input domain model.
- Keep disjointness and coverage in mind.

Try it out, and we'll discuss

Refining Combinations with Constraints

Is equilateral?	×	Is isosceles?	×	Is scalene?
True		True		True
False		False		False

Frame 2

Is Eq: ***True***

Is Iso: ***False***

Is Sc: False

What is wrong with this?

Refining Combinations with Constraints

Is equilateral?	×	Is isosceles?	×	Is scalene?
True		True		True
False		False		False

Frame 2

Is Eq: ***True***

Is Iso: ***False***

Is Sc: False

We can add ***properties*** and ***constraints*** to prune impossible or redundant tests

Is equilateral?	×	Is isosceles?	×	Is scalene?
True **[if Iso]**		True **[property Iso]**		True **[if not Iso]**
False		False		False

Refining Combinations with Constraints

Is equilateral?	×	Is isosceles?	×	Is scalene?
True		True		True
False		False		False

Frame 2
Is Eq: ***True***
Is Iso: ***False***
Is Sc: False

We can add ***properties*** and ***constraints*** to prune impossible or redundant tests

Is equilateral?	×	Is isosceles?	×	Is scalene?
True **[if Iso]**		True **[property Iso]**		True **[if not Iso]**
False		False		False

[Error] annotations can identify cases not benefiting from combinations

Selecting Concrete Inputs

- In theory, any value in a partition can represent it. (equivalence classes)

Selecting Concrete Inputs

- In theory, any value in a partition can represent it. (equivalence classes)
- In practice…
  bugs often live on ***extreme*** cases & the ***boundaries*** of partitions

Selecting Concrete Inputs

- In theory, any value in a partition can represent it. (equivalence classes)
- In practice…
  bugs often live on ***extreme*** cases & the ***boundaries*** of partitions

abs(INT_MIN) → ?

Selecting Concrete Inputs

- In theory, any value in a partition can represent it. (equivalence classes)
- In practice…
  bugs often live on ***extreme*** cases & the ***boundaries*** of partitions

abs(INT_MIN) → ?

```
void toUppercase(char *str) {
  for (int i = 0, e = strlen(str) - 1; i < e; ++i) {
    if (isletter(str[i]) && islower(str[i])) {
      str[i] = str[i] - 32;
    }
  }
  printf("%s\n", str);
}
```

Selecting Concrete Inputs

- In theory, any value in a partition can represent it. (equivalence classes)
- In practice…
  bugs often live on ***extreme*** cases & the ***boundaries*** of partitions
  - This is the same reason we look for OBOEs (off by one errors)

Selecting Concrete Inputs

- In theory, any value in a partition can represent it. (equivalence classes)
- In practice…
  bugs often live on ***extreme*** cases & the ***boundaries*** of partitions
  - This is the same reason we look for OBOEs (off by one errors)
- So we might consider
  - Expected values
  - Invalid, valid, and special values
  - Boundary values

Selecting Concrete Inputs

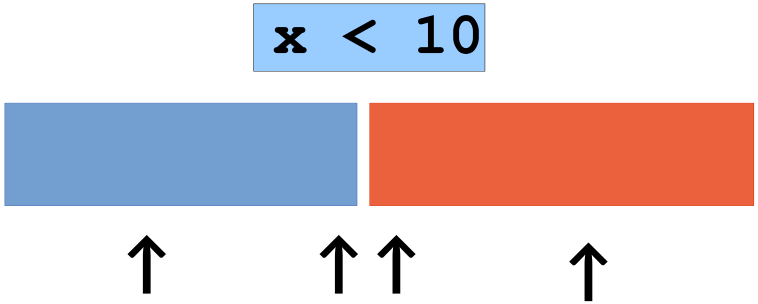
- In theory, any value in a partition can represent it. (equivalence classes)
- In practice…
  bugs often live on ***extreme*** cases & the ***boundaries*** of partitions
  - This is the same reason we look for OBOEs (off by one errors)
- So we might consider
  - Expected values
  - Invalid, valid, and special values
  - Boundary values

abs(x) …, -3, -2, -1, 0, 1, 2, 3, …

↑ ↑ ↑ ↑ ↑

Selecting Concrete Inputs

- In theory, any value in a partition can represent it. (equivalence classes)
- In practice…
  bugs often live on ***extreme*** cases & the ***boundaries*** of partitions
  - This is the same reason we look for OBOEs (off by one errors)
- So we might consider
  - Expected values
  - Invalid, valid, and special values
  - Boundary values



One last example

Command	FIND
Syntax	FIND <pattern> <file>
Function	The FIND command is used to **locate one or more instances** of a given **pattern in a text file**. All lines in the file that contain the pattern are written to standard output. A line containing the pattern is **written only once**, regardless of the number of times the pattern occurs on it. The pattern is any sequence of characters whose **length does not exceed** the maximum length of a line in the file. To include a blank in the pattern, the entire pattern must be **enclosed in quotes (")**. To include a quotation mark in the pattern, **two quotes in a row ("")** must be used.

Part 1: Analyze the specification

- What is the *component*?
- What are the *parameters*?
- What are the *characteristics*?

One last example

```
Parameters:
   Pattern
   Input file (& its contents!)
```

Command	
Syntax	
Function	The FIND command is used to **locate one or more instances** of a given **pattern in a text file**. All lines in the file that contain the pattern are written to standard output. A line containing the pattern is **written only once**, regardless of the number of times the pattern occurs on it. The pattern is any sequence of characters whose **length does not exceed** the maximum length of a line in the file. To include a blank in the pattern, the entire pattern must be **enclosed in quotes (")**. To include a quotation mark in the pattern, **two quotes in a row ("")** must be used.

Part 1: Analyze the specification

- What is the *component*?
- What are the *parameters*?
- What are the *characteristics*?

One last example

Command

Syntax

Function

```
Parameters:
   Pattern
   Input file (& its contents!)
```

```
Characteristics:
   Pattern
   Input file
   Pattern Size
   Quoting
   Embedded Quotes
   File Name
Environment / System Characteristics:
   # of pattern occurrences in file
   # of occurrences on a particular line:
```

es of a given
ttern are
written only
s on it.

oes not exceed
in the pattern,
de a quotation
d.

Part 1: Analyze

- What is the *component*?
- What are the *parameters*?
- What are the *characteristics*?

One last example

- Part 2: Partition the Input Space
  - Guided by intelligence and intuition
  - Combine interface and functionality based approaches as necessary

```
Parameters:
  Pattern Size:
    Empty
    Single character
    Many characters
    Longer than any line in the file
  Quoting:
...
```

One last example

- Part 3: Refine with constraints

```
Pattern size : empty
Quoting : pattern is quoted
Embedded blanks : several embedded blanks
Embedded quotes : no embedded quotes
File name : good file name
Number of occurrences of pattern in file : none
Pattern occurrences on target line : one
```

Problem?

One last example

- Part 3: Refine with constraints

```
Pattern size : empty
Quoting : pattern is quoted
Embedded blanks : several embedded blanks
Embedded quotes : no embedded quotes
File name : good file name
Number of occurrences of pattern in file : none
Pattern occurrences on target line : one
```

Problem?

One last example

- Part 3: Refine with constraints

```
Pattern size : empty
```

Problem?

```
Pattern Size:
  Empty                             [Property Empty]
  Single character                  [Property NonEmpty]
  Many characters                   [Property NonEmpty]
  Longer than any line in the file  [Property NonEmpty]
```

```
Pattern occurrences on target line : one
```

One last example

- Part 3: Refine with constraints

```
Pattern size : empty
```

Problem?

```
Pattern Size:
  Empty                              [Property Empty]
  Single character                   [Property NonEmpty]
  Many characters                    [Property NonEmpty]
  Longer than any line in the file   [Property NonEmpty]
```

```
Quoting:
  Pattern is quoted                  [Property Quoted]
  Pattern is not quoted              [If NonEmpty]
  Pattern is improperly quoted       [If NonEmpty]
```

One last example

- Part 3: Refine with constraints

```
Pattern size : empty
```

Problem?

```
Pattern Size:
  Empty                              [Property Empty]
  Single character                   [Property NonEmpty]
  Many characters                    [Property NonEmpty]
  Longer than any line in the file   [Property NonEmpty]
```

```
Quoting:
  Pattern is quoted                  [Property Quoted]
  Pattern is not quoted              [If NonEmpty]
  Pattern is improperly quoted       [If NonEmpty]
```

What should this do to the number of tests?
To the quality of tests?

One last example

- Part 4:
  - Generate test frames.
  - Analyze.
  - Select concrete values and tests.
  - Prune redundant tests.

Why might scenarios be redundant?

One last example

- Part 4:
  - Generate test frames.
  - Analyze.
  - Select concrete values and tests.
  - Prune redundant tests.

- Then take your tests and automate them

Summary

- Partition based testing allows for testing software without detailed knowledge of its implementation

- Careful design of an input domain model helps
  ensure useful tests and
  avoid less useful tests

- The assumption of equivalence in a partition is a convenience.