Recall Unit Tests

- We started off the semester by talking about testing
  - Input to drive a behavior
  - An oracle to check a behavior
Recall Unit Tests

- We started off the semester by talking about testing
  - Input to drive a behavior
  - An oracle to check a behavior
Recall Unit Tests

- We started off the semester by talking about testing
  - Input to drive a behavior
  - An oracle to check a behavior

- Testing samples the concrete behaviors of the program
Recall Unit Tests

- We started off the semester by talking about testing
  - Input to drive a behavior
  - An oracle to check a behavior

- Testing samples the concrete behaviors of the program
  - Analyzing equivalence classes
Recall Unit Tests

- We started off the semester by talking about testing
  - Input to drive a behavior
  - An oracle to check a behavior

- Testing samples the concrete behaviors of the program
  - Analyzing equivalence classes
  - Program analysis can find richer bugs over a test suite
Recall Unit Tests

- We started off the semester by talking about testing
  - Input to drive a behavior
  - An oracle to check a behavior

- Testing samples the concrete behaviors of the program
  - Analyzing equivalence classes
  - Program analysis can find richer bugs over a test suite
  - Fuzzing can proactively explore the input space but faces hurdles
Recall Unit Tests

• We started off the semester by talking about testing
  – Input to drive a behavior
  – An oracle to check a behavior

• Testing samples the concrete behaviors of the program
  – Analyzing equivalence classes
  – Program analysis can find richer bugs over a test suite
  – Fuzzing can proactively explore the input space but faces hurdles

• Formal reasoning and program analysis can also make each test cover more behavior
Recall Unit Tests

- We started off the semester by talking about testing
  - Input to drive a behavior
  - An oracle to check a behavior

- Testing samples the concrete behaviors of the program
  - Analyzing equivalence classes
  - Program analysis can find richer bugs over a test suite
  - Fuzzing can proactively explore the input space but faces hurdles

- Formal reasoning and program analysis can also make each test cover more behavior

- Property based testing
  - Define tests over invariant properties or specifications
  - Sample constructively from the input space to explore & find bugs
Recall Unit Tests

- We started off the semester by talking about testing
  - Input to drive a behavior
  - An oracle to check a behavior

- Testing samples the concrete behaviors of the program
  - Analyzing equivalence classes
  - Program analysis can find richer bugs over a test suite
  - Fuzzing can proactively explore the input space but faces hurdles

- Formal reasoning and program analysis can also make each test cover more behavior

- Property based testing
  - Define tests over invariant properties or specifications
  - Sample constructively from the input space to explore & find bugs
Abstracting Unit Tests

TEST(testCaseName, testName) {
    // Set up scenario
    // Run scenario on component
    // Check oracle
}
Abstracting Unit Tests

```
TEST(testCaseName, testName) {
  // Set up scenario
  // Run scenario on component
  // Check oracle
}
```

- A scenario could be **concrete** or **abstract**
Abstracting Unit Tests

- A scenario could be concrete or abstract

```c
TEST(testCaseName, testName) {
    // Set up scenario
    // Run scenario on component
    // Check oracle
}
```

\[
\forall x : x > 0
\]

\[
x = 5
\]
Abstracting Unit Tests

- A scenario could be **concrete** or **abstract**
  
  \[ x = 5 \quad \forall x : x > 0 \]

- For an abstract test case, we can
  - Generate test cases
  - Consult the oracle

```c
TEST(testCaseName, testName) {
    // Set up scenario
    // Run scenario on component
    // Check oracle
}
```
Abstracting Unit Tests

- A scenario could be **concrete** or **abstract**
  \[ x = 5 \quad \forall x : x > 0 \]

- For an abstract test case, we can
  - Generate test cases
  - Consult the oracle

- The emphasis is on defining (1) the scenario & (2) the oracle

```
TEST(testCaseName, testName) {
  // Set up scenario
  // Run scenario on component
  // Check oracle
}
```
Abstracting Unit Tests

- A scenario could be concrete or abstract
  \[ x = 5 \quad \forall x : x > 0 \]

- For an abstract test case, we can
  - Generate test cases
  - Consult the oracle

- The emphasis is on defining (1) the scenario & (2) the oracle

- And we can use test generation strategies that we have already seen!
  - random testing
  - symbolic execution
Property Based Testing

- This forms the motivation of property based testing
  - The testing process focuses on functional properties and generating many tests for them
Property Based Testing

- This forms the motivation of property based testing
  - The testing process focuses on functional properties and generating many tests for them

- The exact definition is still evolving
  - First developed with QuickCheck for Haskell in 2000
  - Focus was on generating many random tests from rich type information
Property Based Testing

- This forms the motivation of property based testing
  - The testing process focuses on functional properties and generating many tests for them

- The exact definition is still evolving
  - First developed with QuickCheck for Haskell in 2000
  - Focus was on generating many random tests from rich type information
  - Test case reduction was also automatically applied
Property Based Testing

- This forms the motivation of property based testing
  - The testing process focuses on functional properties and generating many tests for them

- The exact definition is still evolving
  - First developed with QuickCheck for Haskell in 2000
  - Focus was on generating many random tests from rich type information
  - Test case reduction was also automatically applied
  - Now includes symbolic execution as a means of generation
Property Based Testing

- Traditional testing can be seen as example based
  \[ x = 5 \]
- Property testing focuses on generic properties that should hold
  \[ \forall x : x > 0 \]
  \[ \forall x, y, z: \varphi(x, y, z) \rightarrow \psi(x, y, z) \]
Property Based Testing

- Traditional testing can be seen as example based
  \[ x = 5 \]
- Property testing focuses on generic properties that should hold
  \[ \forall x : x > 0 \]
  \[ \forall x, y, z: \varphi(x, y, z) \rightarrow \psi(x, y, z) \]
- For random sampling, *generators* provide ways to sample complex types
  - Property testing tools can provide libraries to help define input spaces
  - Some domains may require substantial initial effort (similar to fuzzing)
Property Based Testing

- Traditional testing can be seen as example based
  \[ x = 5 \]
- Property testing focuses on generic properties that should hold
  \[ \forall x : x > 0 \]
  \[ \forall x, y, z : \varphi(x, y, z) \rightarrow \psi(x, y, z) \]
- For random sampling, *generators* provide ways to sample complex types
  - Property testing tools can provide libraries to help define input spaces
  - Some domains may require substantial initial effort (similar to fuzzing)
- Because the process is so specification focused, it can also help developers understand the intent of their own code
Defining Common Properties

- There are common patterns that we saw before with fuzzing:
Defining Common Properties

- There are common patterns that we saw before with fuzzing:
  - Symmetry
    \[ \text{encode(\text{decode(x)})} = x \]
Defining Common Properties

- There are common patterns that we saw before with fuzzing:
  - Symmetry
    \[ \text{encode}(\text{decode}(x)) = x \]
  - Alternatives
    \[ \text{bubblesort}(x) = \text{qsort}(x) \]
Defining Common Properties

- There are common patterns that we saw before with fuzzing:
  - Symmetry: \( \text{encode}(\text{decode}(x)) = x \)
  - Alternatives: \( \text{bubblesort}(x) = \text{qsort}(x) \)
  - Induction: \( \text{car}(\text{cons}(\text{head}, \text{tail})) = \text{head} \)
Defining Common Properties

There are common patterns that we saw before with fuzzing:

- Symmetry: \( \text{encode}(\text{decode}(x)) == x \)
- Alternatives: \( \text{bubblesort}(x) == \text{qsort}(x) \)
- Induction: \( \text{car}(\text{cons}(\text{head},\text{tail})) == \text{head} \)
- Idempotence: \( \text{qsort}(\text{qsort}(x)) == \text{qsort}(x) \)
Defining Common Properties

- There are common patterns that we saw before with fuzzing:
  - Symmetry: \( \text{encode}(\text{decode}(x)) == x \)
  - Alternatives: \( \text{bubblesort}(x) == \text{qsort}(x) \)
  - Induction: \( \text{car}(\text{cons}(\text{head}, \text{tail})) == \text{head} \)
  - Idempotence: \( \text{qsort}(\text{qsort}(x)) == \text{qsort}(x) \)
  - Invariants: \( \text{qsort}(x).\text{size}() == x.\text{size}() \)
Digging Deeper

- What are good properties to check for a sorting function?

```python
def sort(x):
    ...
```
Digging Deeper

- What are good properties to check for a sorting function?
  ```python
def sort(x):
  ...  
```

- What if we have a sort over only one field?

- The actual properties to check can be more subtle than they appear!
Common Benefits

- Tests have a clear mathematical presentation
Common Benefits

- Tests have a clear mathematical presentation
- The testing process moves from examples to the entire input space
  - You do not need to write every case for each property
  - The testing process is thus more goal oriented & value driven
Common Benefits

- Tests have a clear mathematical presentation
- The testing process moves from examples to the entire input space
  - You do not need to write every case for each property
  - The testing process is thus more goal oriented & value driven
- Can actually decrease maintenance costs with the same (or sometimes greater) coverage
  - What happens if you change an API with normal unit tests?
  - What happens with property based tests?
Common Benefits

- Tests have a clear mathematical presentation
- The testing process moves from examples to the entire input space
  - You do not need to write every case for each property
  - The testing process is thus more goal oriented & value driven
- Can actually decrease maintenance costs with the same (or sometimes greater) coverage
  - What happens if you change an API with normal unit tests?
  - What happens with property based tests?
- Failing test cases even have test case reduction applied
In Practice: Hypothesis

- **Hypothesis** [https://hypothesis.works/]
  - Python, Java, ...
  - Presently uses random testing
  - Enables convenient generators for constructing data
In Practice: Hypothesis

- **Hypothesis** [https://hypothesis.works/](https://hypothesis.works/)
  - Python, Java, ...
  - Presently uses random testing
  - Enables convenient generators for constructing data

```python
from hypothesis import given
from hypothesis.strategies import text

given(text())
@example('')
def test_decode_inverts_encode(s):
    assert decode(encode(s)) == s
```
In Practice: Hypothesis

- **Hypothesis** [https://hypothesis.works/]
  - Python, Java, ...
  - Presently uses random testing
  - Enables convenient generators for constructing data

```python
from hypothesis import given
from hypothesis.strategies import text

given = @given(text())
@example('')
def test_decode_inverts_encode(s):
    assert decode(encode(s)) == s
```
In Practice: Hypothesis

- **Hypothesis** [https://hypothesis.works/](https://hypothesis.works/)
  - Python, Java, ...
  - Presently uses random testing
  - Enables convenient generators for constructing data

```python
from hypothesis import given
from hypothesis.strategies import text

given(text())
@example('')
def test_decode_inverts_encode(s):
    assert decode(encode(s)) == s
```
In Practice: Hypothesis

- **Hypothesis** [https://hypothesis.works/]
  - Python, Java, ...
  - Presently uses random testing
  - Enables convenient generators for constructing data

def test_decode_inverts_encode(s):
    assert decode(encode(s)) == s
In Practice: Hypothesis

- Many generators are built in
- Complex input spaces can require custom generators as mentioned
In Practice: Hypothesis

- Many generators are built in
- Complex input spaces can require custom generators as mentioned

```python
@composite
def distinct_strings_with_common_characters(draw):
    x = draw(text(), min_size=1)
    y = draw(text(alphabet=x))
    assume(x != y)
    return (x, y)
```

- A rich set of primitives is available for more complex generator needs
In Practice: DeepState

- **DeepState** [https://github.com/trailofbits/deepstate](https://github.com/trailofbits/deepstate)
  - C and C++ focused
  - API is compatible with GoogleTest
  - Symbolic execution tries to automatically extract inputs
In Practice: DeepState

- **DeepState** [https://github.com/trailofbits/deepstate]
  - C and C++ focused
  - API is compatible with GoogleTest
  - Symbolic execution tries to automatically extract inputs

```c
TEST(PrimePolynomial, OnlyGeneratesPrimes_NoStreaming) {
    symbolic_unsigned x, y, z;
    DeepState_Assume(x > 0);
    unsigned poly = (x * x) + x + 41;
    DeepState_Assume(y > 1);
    DeepState_Assume(z > 1);
    DeepState_Assume(y < poly);
    DeepState_Assume(z < poly);
    DeepState_Assert(poly != (y * z));
    DeepState_Assert(IsPrime(Pump(poly)));
}
```
Summary

- Property based testing combines test generation & unit tests
Summary

- Property based testing combines test generation & unit tests
- By focusing more on goals rather than examples, it can have benefits even outside of testing
Summary

- Property based testing combines test generation & unit tests
- By focusing more on goals rather than examples, it can have benefits even outside of testing
- Adoption can still require effort in defining good generators