Testing (Part 3/3)

Martin Kellogg

Testing (part 3)

Today's agenda:

- Finish up code level design discussion from lecture 2
- Test input generation (fuzzing)
- Test oracle generation
- Test prioritization & test suite minimization
- Reading Quiz

Testing (part 3)

Today's agenda:

- **Finish up code level design discussion from lecture 2**
- Test input generation (fuzzing)
- Test oracle generation
- Test prioritization & test suite minimization
- Reading Quiz

In-class exercise: rewrite to avoid magic numbers

```
function grossTax(income : number): number {
  if ((0 <= income) && (income <= 10000)) {
    return 0
  } else if ((10000 < income) && (income <= 20000)) {
    return 0.10 * (income - 10000)
  } else if ((20000 < income) && (income <= 50000)) {
    return 1000 + 0.20 * (income - 20000)
  } else {
    return 7000 + 0.25 * (income - 50000)
  }
}
```

In-class exercise: my solution, part 1

```
// defines the tax bracket for income lower < income <= upper.
// if upper is null, then lower < income (no upper bound)
type TaxBracket = {
  lower: number,
  upper: number | null,
  base : number,
  rate : number
}
let brackets : TaxBracket[] = [
  {lower:0, upper:10000, base:0, rate:0},
  {lower:10000, upper:20000, base:0, rate:0.10},
  {lower:20000, upper:50000, base:1000, rate:0.20},
  {lower:50000, upper: null, base:7000, rate:0.25} ]
```

In-class exercise: my solution, part 2

```
// defines the incomes covered by a bracket function
function isInBracket(income : number, bracket : TaxBracket) : boolean {
  return (bracket.upper == null) ?
    (bracket.lower <= income) :
    ((bracket.lower <= income) && (income < bracket.upper))
}
function income2bracket(income : number,
                        brackets : TaxBracket[]) : TaxBracket {
  return brackets.find(b0 => isInBracket(income, b0))
}
function taxByBracket(income : number, bracket : TaxBracket) : number {
  return bracket.base + bracket.rate * (income - bracket.lower)
}
function grossTax(income:number, brackets: TaxBracket[]) : number {
  return taxByBracket(income, income2bracket(income, brackets))
}
```

Avoid magic numbers: another example

- Which of the two is simpler?

Avoid magic numbers: another example

- Which of the two is simpler?
- Answer depends on who you ask:
  - **code writer**: magic numbers version is simpler

Avoid magic numbers: another example

- Which of the two is simpler?
- Answer depends on who you ask:
  - **code writer**: magic numbers version is simpler
  - **code reader**: magic numbers version is shorter, but no magic numbers version is better documented. Toss up.

Avoid magic numbers: another example

- Which of the two is simpler?
- Answer depends on who you ask:
  - **code writer**: magic numbers version is simpler
  - **code reader**: magic numbers version is shorter, but no magic numbers version is better documented. Toss up.
  - **code maintainer who needs to make a change**: magic number version is difficult to deal with, no magic numbers makes the change trivial

Avoid magic numbers: another example

- Which of the two is simpler?
- Answer depends on who you ask:
  - **code writer**: magic numbers version is simpler
  - **code reader**: magic numbers version is shorter, but no magic numbers version is better documented. Toss up.
  - **code maintainer who needs to make a change**: magic number version is difficult to deal with, no magic numbers makes the change trivial

Who to optimize for?

Who to optimize for?

- The code **writer**: only if you expect to throw the code away after you use it once.

Who to optimize for?

- The code **writer**: only if you expect to throw the code away after you use it once.

  Example: simple bash script to accomplish a specific, one-off task

Who to optimize for?

- The code **writer**: only if you expect to throw the code away after you use it once.
- The code **reader**: any code you expect to keep. A good heuristic that I use: am I going to check this into source control?

Who to optimize for?

- The code **writer**: only if you expect to throw the code away after you use it once.
- The code **reader**: any code you expect to keep. A good heuristic that I use: am I going to check this into source control?
- The code **maintainer**: any code that is likely to change. This is most code that you're writing in the real world!

Who to optimize for?

- The code **writer**: only if you expect to throw the code away after you use it once.
- The code **reader**: any code you expect to keep. A good heuristic that I use: am I going to check this into source control?
- The code **maintainer**: any code that is likely to change. This is most code that you're writing in the real world!

DANGER: premature optimization via over-engineering
don't sacrifice readability or usability for maintainability!

Code-level Design

Lecture 2's agenda:

- Why does code-level design matter?
- Some general principles, with examples
- In-class exercise + break
- **Automation and linting**
- Our course style guide
- Reading Quiz

A surprise: non-standard formatting

What's wrong with the following (Java) code?

```
public abstract class racecar {

private final int Number_of_gears = 6;

       public abstract void DRIVE();

 public int GetNumberOfGears(){return Number_of_gears;}

}
```

A surprise: non-standard formatting

What's wrong with the following (Java) code?

```
public abstract class racecar {
private final int Number_of_gears = 6;
       public abstract void DRIVE();
 public int GetNumberOfGears(){return Number_of_gears;}
}
```

A surprise: non-standard formatting

What's wrong with the following (Java) code?

```
public abstract class RaceCar {

  private final int NUMBER_OF_GEARS = 6;

  public abstract void drive();

  public int getNumberOfGears(){
    return NUMBER_OF_GEARS;
  }
}
```

A surprise: non-standard formatting

- Doing this ourselves is time-consuming and error-prone
- How do we decide which format is best?

A surprise: non-standard formatting

- Doing this ourselves is time-consuming and error-prone
- How do we decide which format is best?

Solution to both problems: use an **automatic** formatting tool

A surprise: non-standard formatting

- Doing this ourselves is time-consuming and error-prone
- How do we decide which format is best?

Solution to both problems: use an **automatic** formatting tool

- avoids flamewars about e.g., tabs vs spaces
- automatically enforced = we don't have to think about it
- reduces surprises when reading code

Automated formatters

- There’s at least one for every language you are likely to be using

Automated formatters

- There's at least one for every language you are likely to be using
- E.g.,:
  - Java has Spotless, GoogleJavaFormat, Checkstyle
  - Python has black, autopep8, yapf
  - Go has gofmt
  - JavaScript has prettier (which we'll use in this class)

Automated formatters

- There's at least one for every language you are likely to be using
- E.g.,:
  - Java has Spotless, GoogleJavaFormat, Checkstyle
  - Python has black, autopep8, yapf
  - Go has gofmt
  - JavaScript has prettier (which we'll use in this class)
- **Lesson**: always use an automated formatter

Aside: “opinionated”

Definition: a tool is ***opinionated*** if it builds in assumptions about how its target (e.g., your code for an automated formatter) should be

Aside: “opinionated”

Definition: a tool is ***opinionated*** if it builds in assumptions about how its target (e.g., your code for an automated formatter) should be

A good automated formatter is opinionated: reduces intra-team arguments about formatting.

Automated formatters vs linters

Definition: a ***linter*** is a static code style checker

Automated formatters vs linters

Definition: a ***linter*** is a static code style checker

- Linters **find** style problems.
- Automated formatters **fix** style problems.

Automated formatters vs linters

Definition: a ***linter*** is a static code style checker

- Linters **find** style problems.
- Automated formatters **fix** style problems.

You'll see both terms, and some linters also look for other mistakes.

We'll use both `prettier` (an automated formatter) and `ESLint` (a linter) in this course.

Testing (part 3)

Today's agenda:

- Finish up code level design discussion from lecture 2
- **Test input generation (fuzzing)**
- Test oracle generation
- Test prioritization & test suite minimization
- Reading Quiz

Test data

- What are **all** the inputs to a test?

Test data

- What are **all** the inputs to a test?
  - Many programs (especially student programs) read from a file or stdin …

Test data

- What are **all** the inputs to a test?
  - Many programs (especially student programs) read from a file or stdin …
  - But what **else** is “read in” by a program and may influence its behavior?

Test data

- What are **all** the inputs to a test?
  - Many programs (especially student programs) read from a file

What else besides "input" can **influence** program behavior?

- User Input (e.g., GUI)
- Environment Variables, Command-Line Args
- Scheduler Interleavings
- Data from the Filesystem
  - User configuration, data files
- Data from the Network
  - Server and service responses

Test data: operating systems philosophy

Test data: operating systems philosophy

- “Everything is a file.”

Test data: operating systems philosophy

- “Everything is a file.”
- After a few libraries and levels of indirection, reading from the user's keyboard boils down to opening a **special device file** (e.g., /dev/ttyS0) and reading from it
  - Similarly with mouse clicks, GUI commands, etc.

Test data: operating systems philosophy

- "Everything is a file."
- After a few libraries and levels of indirection, reading from the user's keyboard boils down to opening a **special device file** (e.g., /dev/ttyS0) and reading from it
  - Similarly with mouse clicks, GUI commands, etc.
- Ultimately programs can only interact with the outside world through ***system calls***
  - open, read, write, socket, fork, gettimeofday

Test data: operating systems philosophy

- “Everything is a file.”
- After a few libraries and levels of indirection, reading from the user's keyboard boils down to opening a **special device file** (e.g., /dev/ttyS0) and reading from it
  - Similarly with mouse clicks, GUI commands, etc.
- Ultimately programs can only interact with the outside world through ***system calls***
  - open, read, write, socket, fork, gettimeofday
- System calls (plus OS scheduling, etc.) are the full inputs

Test data: operating systems philosophy

- "Everything is a file"
- After a few librari ... the user's keyboard be ... e (e.g., /dev/ttyS0) and re
  - Similarly with
- Ultimately progra ... orld through *system call*
  - open, read, write, socket, fork, gettimeofday
- System calls (plus OS scheduling, etc.) are the full inputs

1. Fully **hermetic** tests should include all these inputs
2. We want fully hermetic tests

Test data: operating systems philosophy

- "Everything is a file"
- After a few librari … n the user's keyboard b … e (e.g., /dev/ttyS0) and re …
  - Similarly with …
- Ultimately progra … rld through *system call* …
  - open, read, write, socket, fork, gettimeofday
- System calls (plus OS scheduling, etc.) are the full inputs

1. Fully **hermetic** tests should include all these inputs
2. We want fully hermetic tests
3. 1 & 2 imply test input generation must also **control the environment**

Test input generation

Test input generation

- As a human, often **choosing good test inputs** is the hardest part of writing a test

Test input generation

- As a human, often **choosing good test inputs** is the hardest part of writing a test
- For a computer, that's not true: computers can pick inputs **very fast** (given some policy)

Test input generation

- As a human, often **choosing good test inputs** is the hardest part of writing a test
- For a computer, that's not true: computers can pick inputs **very fast** (given some policy)
- **Key problem**: which inputs should we pick?

Test input generation

- As a human, often **choosing good test inputs** is the hardest part of writing a test
- For a computer, that's not true: computers can pick inputs **very fast** (given some policy)
- **Key problem**: which inputs should we pick?
  - Lens of **Logic**: choose inputs that will maximize coverage

Test input generation

- As a human, often **choosing good test inputs** is the hardest part of writing a test
- For a computer, that's not true: computers can pick inputs **very fast** (given some policy)
- **Key problem**: which inputs should we pick?
  - Lens of **Logic**: choose inputs that will maximize coverage
  - Lens of **Statistics**: choose inputs "at random"

Test input generation

- As a human, often **choosing good test inputs** is the hardest part of writing a test
- For a computer, that's not true: computers can pick inputs **very fast** (given some policy)
- **Key problem**: which inputs should we pick?
  - Lens of **Logic**: choose inputs that will maximize coverage
  - Lens of **Statistics**: choose inputs "at random"
  - Lens of **Adversity**: choose inputs that kill mutants

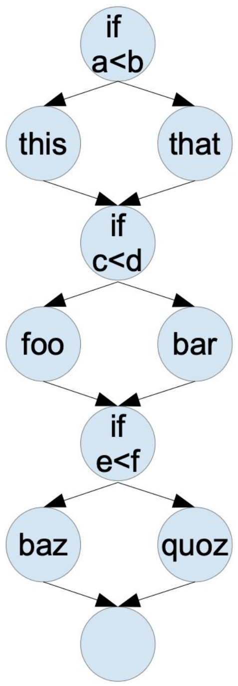
Lens of Logic: maximize coverage

Lens of Logic: maximize coverage

```
foo(a,b,c,d,e,f):
    if a < b: this
    else: that
    if c < d: foo
    else: bar
    if e < f: baz
    else: quoz
```

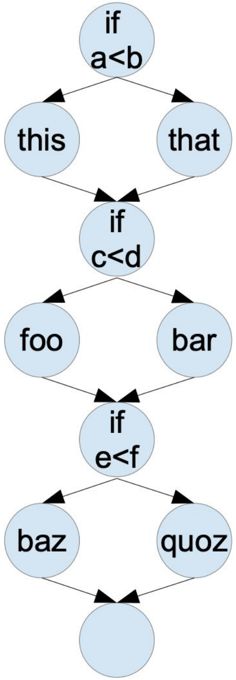
Lens of Logic: maximize coverage

```
foo(a,b,c,d,e,f):
    if a < b: this
    else: that
    if c < d: foo
    else: bar
    if e < f: baz
    else: quoz
```



Lens of Logic: maximize coverage

```
foo(a,b,c,d,e,f):
    if a < b: this
    else: that
    if c < d: foo
    else: bar
    if e < f: baz
    else: quoz
```

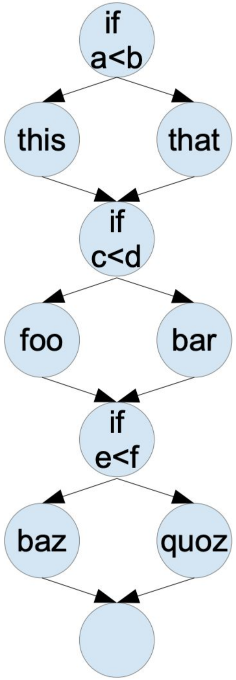


How would you choose inputs that **maximize**:

- **line** coverage?

Lens of Logic: maximize coverage

```
foo(a,b,c,d,e,f):
    if a < b: this
    else: that
    if c < d: foo
    else: bar
    if e < f: baz
    else: quoz
```

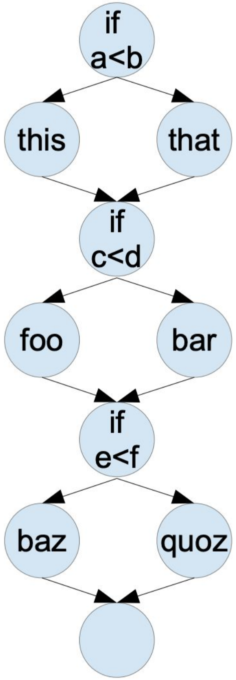


How would you choose inputs that **maximize**:

- **line** coverage?
- **branch** coverage?

Lens of Logic: maximize coverage

```
foo(a,b,c,d,e,f):
    if a < b: this
    else: that
    if c < d: foo
    else: bar
    if e < f: baz
    else: quoz
```

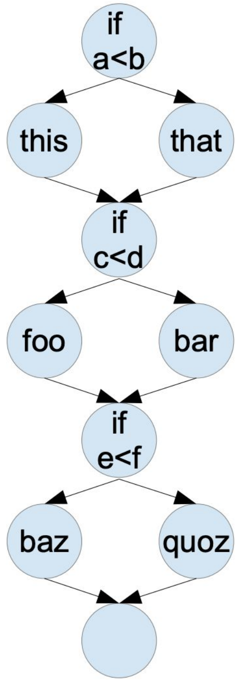


How would you choose inputs that **maximize**:

- **line** coverage?
- **branch** coverage?
- **path** coverage?

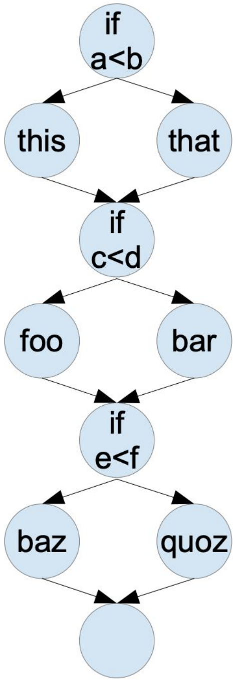
Lens of Logic: maximize coverage

- If you have **N** sequential (or serial) if statements …



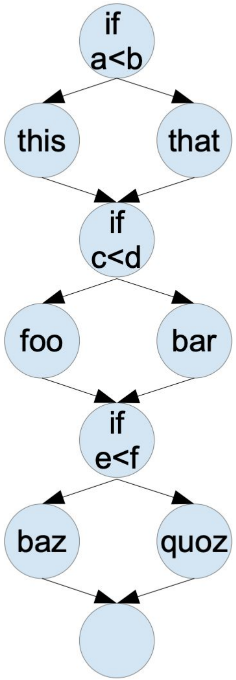
Lens of Logic: maximize coverage

- If you have **N** sequential (or serial) if statements ...
- There are **2N** branch edges



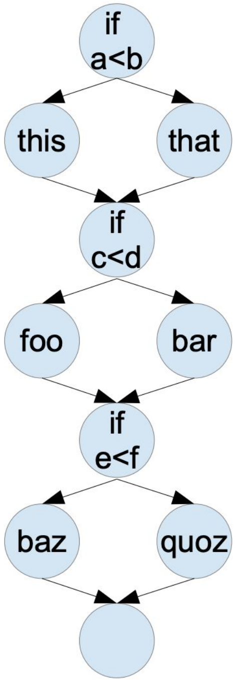
Lens of Logic: maximize coverage

- If you have **N** sequential (or serial) if statements …
- There are **2N** branch edges
  - Which you could cover in 2 tests!



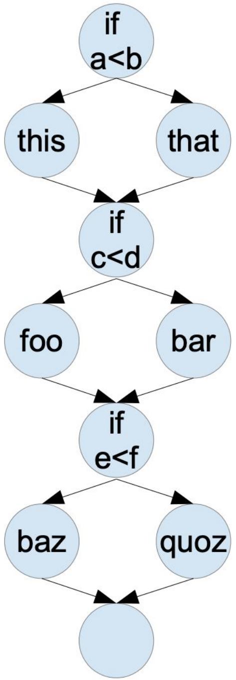
Lens of Logic: maximize coverage

- If you have **N** sequential (or serial) if statements …
- There are **2N** branch edges
  - Which you could cover in 2 tests!
    - One always goes left, one always right



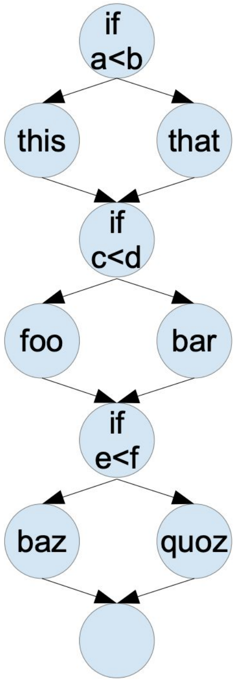
Lens of Logic: maximize coverage

- If you have **N** sequential (or serial) if statements …
- There are **2N** branch edges
  - Which you could cover in 2 tests!
    - One always goes left, one always right
- But there are **2^N paths**



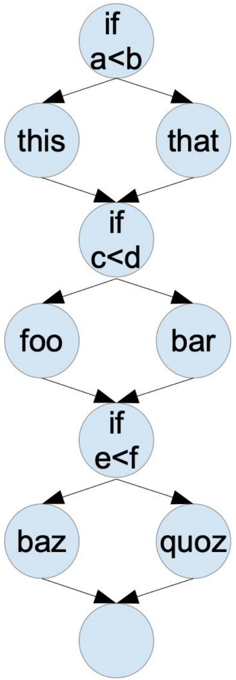
Lens of Logic: maximize coverage

- If you have **N** sequential (or serial) if statements ...
- There are **2N** branch edges
  - Which you could cover in 2 tests!
    - One always goes left, one always right
- But there are **2^N paths**
  - You need **2^N tests** to cover them



Lens of Logic: maximize coverage

- If you have **N** sequential (or serial) if statements …
- There are **2N** branch edges
    - Which you could cover in 2 tests!
        - One always goes left, one always right
- But there are **2^N paths**
    - You need **2^N tests** to cover them
- Path coverage **subsumes** branch coverage



Lens of Logic: maximize coverage

- Consider generating test inputs to cover a path

Lens of Logic: maximize coverage

- Consider generating test inputs to cover a path
  - If we could do that, branch/statement/etc coverage is easy

Lens of Logic: maximize coverage

- Consider generating test inputs to cover a path
  - If we could do that, branch/statement/etc coverage is easy
- **Key idea**: solve this problem with **math**

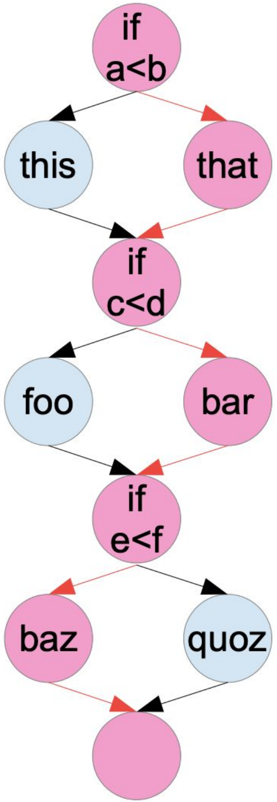
Lens of Logic: maximize coverage

- Consider generating test inputs to cover a path
  - If we could do that, branch/statement/etc coverage is easy
- **Key idea**: solve this problem with **math**

Definition: a ***path predicate*** (or ***path condition***, or ***path constraint***) is a boolean formula over program variables that is true when the program executes the given path

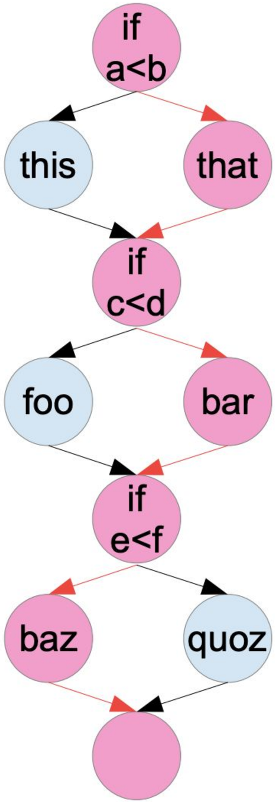
Lens of Logic: path predicate example

- Consider the highlighted (in **pink**) path
  - i.e., “false, false, true”
- What is its path predicate?



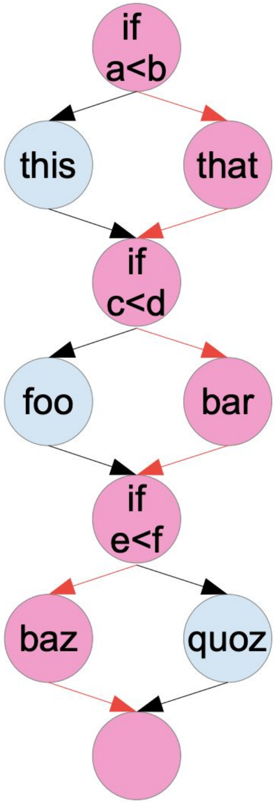
Lens of Logic: path predicate example

- Consider the highlighted (in **pink**) path
  - i.e., “false, false, true”
- What is its path predicate?
  - `a >= b && c >= d && e < f`



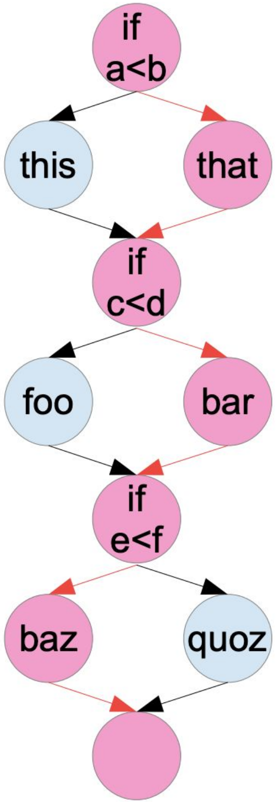
Lens of Logic: path predicate example

- Consider the highlighted (in **pink**) path
  - i.e., “false, false, true”
- What is its path predicate?
  - `a >= b && c >= d && e < f`
- When the path predicate is true, control flow will follow the given path



Lens of Logic: path predicate example

- Consider the highlighted (in **pink**) path
  - i.e., “false, false, true”
- What is its path predicate?
  - `a >= b && c >= d && e < f`
- When the path predicate is true, control flow will follow the given path
- So, given a path predicate, how do we choose a test input that covers the path?



Lens of Logic: solving path predicates

Definition: A ***satisfying assignment*** is a mapping from variables to values that makes a predicate true.

Lens of Logic: solving path predicates

Definition: A ***satisfying assignment*** is a mapping from variables to values that makes a predicate true.

- What is a satisfying assignment for
  - `a >= b && c >= d && e < f` ?

Lens of Logic: solving path predicates

Definition: A ***satisfying assignment*** is a mapping from variables to values that makes a predicate true.

- What is a satisfying assignment for
  - `a >= b && c >= d && e < f` ?
    - `a=5, b=4, c=3, d=2, e=1, f=2`
    - `a=0, b=0, c=0, d=0, e=0, f=1`
    - … many more

Lens of Logic: solving path predicates

- How do we find satisfying assignments in general?

Lens of Logic: solving path predicates

- How do we find satisfying assignments in general?
    - Option 1: **ask humans**
        - labor-intensive, slow, expensive, etc.

Lens of Logic: solving path predicates

- How do we find satisfying assignments in general?
    - Option 1: **ask humans**
        - labor-intensive, slow, expensive, etc.
    - Option 2: repeatedly **guess randomly**
        - works surprisingly well (when answers are **not sparse**)

Lens of Logic: solving path predicates

- How do we find satisfying assignments in general?
  - Option 1: **ask humans**
    - labor-intensive, slow, expensive, etc.
  - Option 2: repeatedly **guess randomly**
    - works surprisingly well (when answers are **not sparse**)
  - Option 3: use an ***automated theorem prover***
    - cf. Wolfram Alpha, MatLab, Mathematica, Z3, etc.
    - works very well for a **restricted class of equations** (e.g., linear but not arbitrary polynomials, etc.)

Lens of Logic: test input generation plan

- Consider generating high-branch-coverage tests for a method:

Lens of Logic: test input generation plan

- Consider generating high-branch-coverage tests for a method:
- **Enumerate** “all” paths in the method

Lens of Logic: test input generation plan

- Consider generating high-branch-coverage tests for a method:
- **Enumerate** "all" paths in the method
- For each path, **collect** the path predicate

Lens of Logic: test input generation plan

- Consider generating high-branch-coverage tests for a method:
- **Enumerate** “all” paths in the method
- For each path, **collect** the path predicate
- For each path predicate, **solve** it

Lens of Logic: test input generation plan

- Consider generating high-branch-coverage tests for a method:
- **Enumerate** "all" paths in the method
- For each path, **collect** the path predicate
- For each path predicate, **solve** it
    - A solution is a satisfying assignment of values to input variables → those are your test input

Lens of Logic: test input generation plan

- Consider generating high-branch-coverage tests for a method:
- **Enumerate** "all" paths in the method
- For each path, **collect** the path predicate
- For each path predicate, **solve** it
  - A solution is a satisfying assignment of values to input variables → those are your test input
  - None found? Dead code, tough predicate, etc.

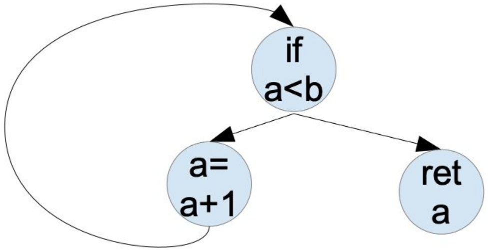
Lens of Logic: enumerating paths

- What could **go wrong** with enumerating paths in a method?

Lens of Logic: enumerating paths

- What could go wrong with enumerating paths in a method?
- There could be infinitely many!

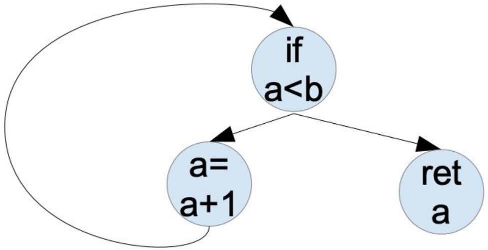
```
while a < b:
  a = a + 1
return a
```



Lens of Logic: enumerating paths

- What could **go wrong** with enumerating paths in a method?
- There could be **infinitely many**!

```
while a < b:
  a = a + 1
return a
```



- One path corresponds to executing the loop once, another to twice, another to three times, etc.

Lens of Logic: enumerating paths: approximation

- **Key idea**: don’t enumerate all paths, **approximate** instead

Lens of Logic: enumerating paths: approximation

- **Key idea**: don’t enumerate all paths, **approximate** instead
- Typical Approximations:

Lens of Logic: enumerating paths: approximation

- **Key idea**: don’t enumerate all paths, approximate instead
- Typical Approximations:
  - Consider only acyclic paths (corresponds to taking each loop zero times or one time)

Lens of Logic: enumerating paths: approximation

- **Key idea**: don't enumerate all paths, **approximate** instead
- Typical Approximations:
  - Consider only **acyclic** paths (corresponds to taking each loop zero times or one time)
  - Consider only taking each loop **at most *k*** times

Lens of Logic: enumerating paths: approximation

- **Key idea**: don't enumerate all paths, **approximate** instead
- Typical Approximations:
  - Consider only **acyclic** paths (corresponds to taking each loop zero times or one time)
  - Consider only taking each loop **at most *k*** times
  - Enumerate paths breadth-first or depth-first and **stop after *k*** paths have been enumerated

Lens of Logic: enumerating paths: approximation

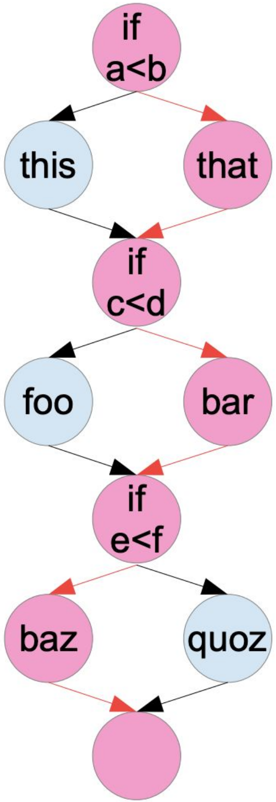
- **Key idea**: don't enumerate all paths, **approximate** instead
- Typical Approximations:
  - Consider only **acyclic** paths (corresponds to taking each loop zero times or one time)
  - Consider only taking each loop **at most *k*** times
  - Enumerate paths breadth-first or depth-first and **stop after *k*** paths have been enumerated
- For more on this topic, take a graduate-level course on program analysis or compilers

Lens of Logic: test input generation plan

- Consider generating high-branch-coverage tests for a method:
- **Enumerate** "all" paths in the method
- For each path, **collect** the path predicate
- For each path predicate, **solve** it
    - A solution is a satisfying assignment of values to input variables → those are your test input
    - None found? Dead code, tough predicate, etc.

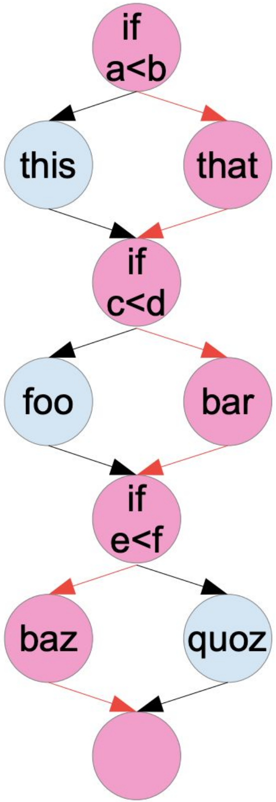
Lens of Logic: collecting path predicates

- Now we have a path through the program
- What could go wrong with **collecting** the path predicate?



Lens of Logic: collecting path predicates

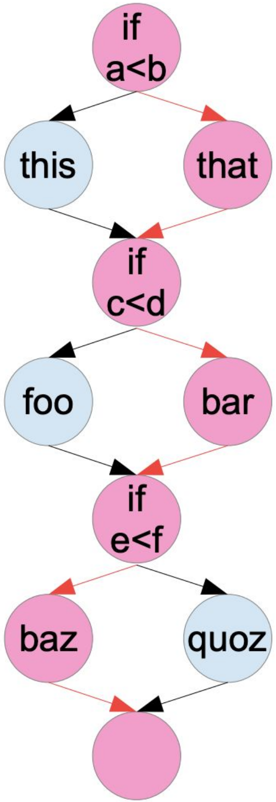
- Now we have a path through the program
- What could go wrong with **collecting** the path predicate?
  - The path predicate may not be **expressible** in terms of the inputs we control



Lens of Logic: collecting path predicates

- Now we have a path through the program
- What could go wrong with **collecting** the path predicate?
  - The path predicate may not be **expressible** in terms of the inputs we control

```
foo(a,b):
  str1 = read_from_url("abc.com")
  str2 = read_from_url("xyz.com")
  if (str1 == str2): bar()
```

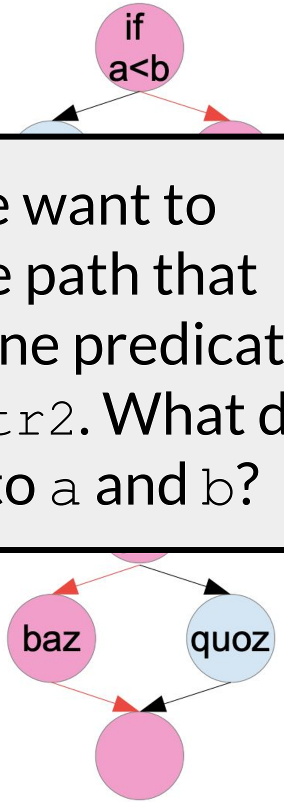


Lens of Logic: collecting path predicates

- Now we have a path through the program
- What could go wrong with **collecting** th
  predicate?
  - The path predicate may not be **expr**
    terms of the inputs we control

Suppose we want to exercise the path that calls `bar`. One predicate is `str1==str2`. What do you assign to `a` and `b`?

```
foo(a,b):
  str1 = read_from_url("abc.com")
  str2 = read_from_url("xyz.com")
  if (str1 == str2): bar()
```



Lens of Logic: path predicate woes

- When we can’t solve for a path predicate, what can we do?

Lens of Logic: path predicate woes

- When we can’t solve for a path predicate, what can we do?
    - **Ignore the problem** (i.e., don’t generate a test)

Lens of Logic: path predicate woes

- When we can’t solve for a path predicate, what can we do?
  - **Ignore the problem** (i.e., don’t generate a test)
- Remember, testing can show the presence of bugs, but not their absence → **no guarantee** either way

Lens of Logic: path predicate woes

- When we can’t solve for a path predicate, what can we do?
    - **Ignore the problem** (i.e., don’t generate a test)
- Remember, testing can show the presence of bugs, but not their absence → **no guarantee** either way
- So, we make a **best effort**:

Lens of Logic: path predicate woes

- When we can't solve for a path predicate, what can we do?
  - **Ignore the problem** (i.e., don't generate a test)
- Remember, testing can show the presence of bugs, but not their absence → **no guarantee** either way
- So, we make a **best effort**:
  - Collect the path predicates as best we can

Lens of Logic: path predicate woes

- When we can’t solve for a path predicate, what can we do?
  - **Ignore the problem** (i.e., don’t generate a test)
- Remember, testing can show the presence of bugs, but not their absence → **no guarantee** either way
- So, we make a **best effort**:
  - Collect the path predicates as best we can
  - Ask the solver to find a solution in terms of the input variables

Lens of Logic: path predicate woes

- When we can't solve for a path predicate, what can we do?
  - **Ignore the problem** (i.e., don't generate a test)
- Remember, testing can show the presence of bugs, but not their absence → **no guarantee** either way
- So, we make a **best effort**:
  - Collect the path predicates as best we can
  - Ask the solver to find a solution in terms of the input variables
  - If it can't (because the math is too hard, we don't control the input, etc.), we give up

Lens of Logic: test input generation plan

- Consider generating high-branch-coverage tests for a method:
- **Enumerate** “all” paths in the method
- For each path, **collect** the path predicate
- For each path predicate, **solve** it
    - A solution is a satisfying assignment of values to input variables → those are your test input
    - None found? Dead code, tough predicate, etc.

Lens of Logic: test input generation plan

- Recall: we want to automatically generate **test cases**

Lens of Logic: test input generation plan

- Recall: we want to automatically generate **test cases**
- We have an approach that works well in practice:
  - **Enumerate** some paths
  - **Extract** their path constraints
  - **Solve** those path constraints

Lens of Logic: test input generation plan

- Recall: we want to automatically generate **test cases**
- We have an approach that works well in practice:
  - **Enumerate** some paths
  - **Extract** their path constraints
  - **Solve** those path constraints
- What are we **missing**?

Lens of Logic: test input generation plan

- Recall: we want to automatically generate **test cases**
- We have an approach that works well in practice:
  - **Enumerate** some paths
  - **Extract** their path constraints
  - **Solve** those path constraints
- What are we **missing**?
  - Oracles!

Testing (part 3)

Today's agenda:

- Finish up code level design discussion from lecture 2
- Test input generation (fuzzing)
- **Test oracle generation**
- Test prioritization & test suite minimization
- Reading Quiz

Oracle generation

- Generating input is of limited value if **we don’t know what the program is supposed to do** with that input

Oracle generation

- Generating input is of limited value if **we don’t know what the program is supposed to do** with that input
- **Key question**: if we generate an input for a given path, **how do we tell** if the program behaved correctly?

Oracle generation: difficulty

- Oracles are **tricky**.

Oracle generation: difficulty

- Oracles are **tricky**.
  - Many believe that formally writing down what a program should do is **as hard** as coding it.

Oracle generation: difficulty

- Oracles are **tricky**.
  - Many believe that formally writing down what a program should do is **as hard** as coding it.
- The ***Oracle Problem*** is the difficulty and cost of determining the correct test oracle (i.e., output) for a given input.

Oracle generation: difficulty

- Oracles are **tricky**.
  - Many believe that formally writing down what a program should do is **as hard** as coding it.
- The ***Oracle Problem*** is the difficulty and cost of determining the correct test oracle (i.e., output) for a given input.
  - "What should the program do?"

Oracle generation: difficulty

- Oracles are **tricky**.
  - Many believe that formally writing down what a program should do is **as hard** as coding it.
- The ***Oracle Problem*** is the difficulty and cost of determining the correct test oracle (i.e., output) for a given input.
  - "What should the program do?"
  - It is **expensive** both for humans and for machines.

Oracle generation: difficulty

- Oracles are **tricky**.
  - Many believe that formally writing down what a program should do is **as hard** as coding it.
- The ***Oracle Problem*** is the difficulty and cost of determining the correct test oracle (i.e., output) for a given input.
  - "What should the program do?"
  - It is **expensive** both for humans and for machines.
    - and, for machines, sometimes impossible!

Oracle generation: implicit oracles

Observation: there are some things programs definitely shouldn't do given **any** input

Oracle generation: implicit oracles

Observation: there are some things programs definitely shouldn't do given **any** input

- crash, segfault, loop forever, exfiltrate user data, etc.

Oracle generation: implicit oracles

Observation: there are some things programs definitely shouldn't do given **any** input

- crash, segfault, loop forever, exfiltrate user data, etc.
- **key idea**: run the program and check if it does any of these **definitely bad** things

Oracle generation: implicit oracles

Observation: there are some things programs definitely shouldn't do given **any** input

- crash, segfault, loop forever, exfiltrate user data, etc.
- **key idea**: run the program and check if it does any of these **definitely bad** things

Definition: an ***implicit oracle*** is one associated with the language or architecture, rather than program-specific semantics (e.g., "don't segfault", "don't loop forever").

Oracle generation: implicit oracles

Observation: there are some things program
given **any** input

- crash, segfault, loop forever, exfiltrate us
- **key idea**: run the program and check if it
  definitely bad things

Implicit oracles like these are used by **most test generation tools** in the real world.

Definition: an ***implicit oracle*** is one associated with the language or architecture, rather than program-specific semantics (e.g., “don't segfault”, “don't loop forever”).

Oracle generation: invariants as oracles

Observation: programs **usually** behave correctly

Oracle generation: invariants as oracles

Observation: programs **usually** behave correctly

- e.g., if I have a human-written test suite with ten tests, and we have `index == array_len - 1` in **every test**

Oracle generation: invariants as oracles

Observation: programs **usually** behave correctly

- e.g., if I have a human-written test suite with ten tests, and we have `index == array_len - 1` in **every test**
- then maybe the correct oracle is that on **every input** we should have `index == array_len - 1`

Oracle generation: invariants as oracles

Observation: programs **usually** behave correctly

- e.g., if I have a human-written test suite with ten tests, and we have `index == array_len - 1` in **every test**
- then maybe the correct oracle is that on **every input** we should have `index == array_len - 1`

Definition: an ***invariant*** is a predicate over program expressions that is true on every execution

Oracle generation: invariants as oracles

Observation: programs **usually** behave correctly

- e.g., if I have a human-written test suite with ten tests, and we have `index == array_len - 1` in **every test**
- then maybe the correct oracle is that on **every input** we should have `index == array_len - 1`

Definition: an ***invariant*** is a predicate over program expressions that is true on every execution

- high-quality invariants can serve as test oracles

Oracle generation: dynamic invariant detection

- There are tools for invariant detection called ***dynamic invariant detectors***

Oracle generation: dynamic invariant detection

- There are tools for invariant detection called ***dynamic invariant detectors***
    - **Key idea**: find invariants that are true on the human-written test suite, then apply those to the test inputs we generate

Oracle generation: dynamic invariant detection

- There are tools for invariant detection called ***dynamic invariant detectors***
  - **Key idea**: find invariants that are true on the human-written test suite, then apply those to the test inputs we generate
    - report any violation to a human

Oracle generation: dynamic invariant detection

- There are tools for invariant detection called ***dynamic invariant detectors***
  - **Key idea**: find invariants that are true on the human-written test suite, then apply those to the test inputs we generate
    - report any violation to a human
  - For more information (e.g., how to build one) take a graduate-level class on program analysis or read the Daikon paper (September 27 optional reading!)

Oracle generation: differential testing

Observation: there are many programs with **similar or identical specifications**

Oracle generation: differential testing

Observation: there are many programs with **similar or identical specifications**

- if we are building such a program, we can use **another implementation** as an oracle

Oracle generation: differential testing

Observation: there are many programs with **similar or identical specifications**

- if we are building such a program, we can use **another implementation** as an oracle
- e.g., if we're writing a C compiler, we can compare our output to gcc

Oracle generation: differential testing

Observation: there are many programs with **similar or identical specifications**

- if we are building such a program, we can use **another implementation** as an oracle
- e.g., if we're writing a C compiler, we can compare our output to gcc

Definition: ***differential testing*** is a technique for testing two related programs by comparing their output on generated test inputs. Any difference indicates non-conformance in one of the two.

Oracle generation: differential testing

Advantages and disadvantages of differential testing:

Oracle generation: differential testing

Advantages and disadvantages of differential testing:

- only applicable in **limited situations**: need another implementation

Oracle generation: differential testing

Advantages and disadvantages of differential testing:

- only applicable in **limited situations**: need another implementation
  - but **useful more often than you might think** - for example, when writing a "fast" version of a routine, you can compare its output to a "slow" but easy-to-implement version

Oracle generation: differential testing

Advantages and disadvantages of differential testing:

- only applicable in **limited situations**: need another implementation
    - but **useful more often than you might think** - for example, when writing a “fast” version of a routine, you can compare its output to a “slow” but easy-to-implement version
- a human needs to decide **which of the two is correct**

Oracle generation: differential testing

Advantages and disadvantages of differential testing:

- only applicable in **limited situations**: need another implementation
  - but **useful more often than you might think** - for example, when writing a “fast” version of a routine, you can compare its output to a “slow” but easy-to-implement version
- a human needs to decide **which of the two is correct**
  - and sometimes neither is!

Oracle generation: differential testing

Advantages and disadvantages of differential testing:

- only applicable in **limited situations**: need another implementation
  - but **useful more often than you might think** - for example, when writing a “fast” version of a routine, you can compare its output to a “slow” but easy-to-implement version
- a human needs to decide **which of the two is correct**
  - and sometimes neither is!
- but, differential testing provides a **much stronger oracle** than other automated techniques

Testing (part 3)

Today's agenda:

- Finish up code level design discussion from lecture 2
- Test input generation (**fuzzing**)
- Test oracle generation
- Test prioritization & test suite minimization
- Reading Quiz

Test input generation

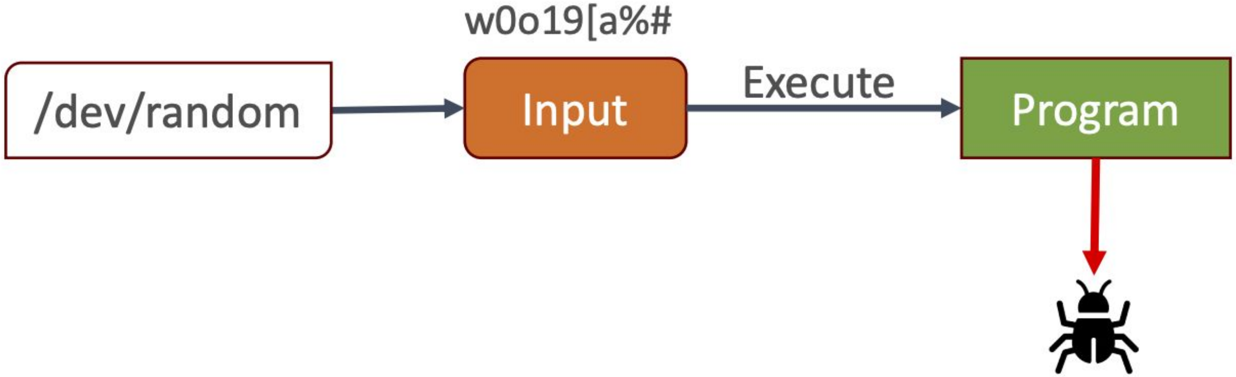
- As a human, often **choosing good test inputs** is the hardest part of writing a test
- For a computer, that's not true: computers can pick inputs **very fast** (given some policy)
- **Key problem**: which inputs should we pick?
    - Lens of **Logic**: choose inputs that will maximize coverage
    - Lens of **Statistics**: choose inputs "at random"
    - Lens of **Adversity**: choose inputs that kill mutants

Lens of Statistics: fuzzing and random testing

Key idea: provide inputs "at random" to the program and use an implicit oracle

Lens of Statistics: fuzzing and random testing

Key idea: provide inputs “at random” to the program and use an implicit oracle



Lens of Statistics: fuzzing and random testing

Definition: ***fuzzing*** (or ***fuzz testing***) is an automated testing technique that involves providing random or semi-random inputs to a program and monitoring for violations of an implicit oracle.

Lens of Statistics: fuzzing and random testing

Definition: ***fuzzing*** (or ***fuzz testing***) is an automated testing technique that involves providing random or semi-random inputs to a program and monitoring for violations of an implicit oracle.

- typical oracle: **crashes**

Lens of Statistics: fuzzing and random testing

Definition: ***fuzzing*** (or ***fuzz testing***) is an automated testing technique that involves providing random or semi-random inputs to a program and monitoring for violations of an implicit oracle.

- typical oracle: **crashes**
- totally random input rarely works well

Lens of Statistics: fuzzing and random testing

Definition: ***fuzzing*** (or ***fuzz testing***) is an automated testing technique that involves providing random or semi-random inputs to a program and monitoring for violations of an implicit oracle.

- typical oracle: **crashes**
- totally random input rarely works well
  - most programs have **structured input**

Lens of Statistics: fuzzing and random testing

Definition: ***fuzzing*** (or ***fuzz testing***) is an automated testing technique that involves providing random or semi-random inputs to a program and monitoring for violations of an implicit oracle.

- typical oracle: **crashes**
- totally random input rarely works well
  - most programs have **structured input**
  - so modern fuzzers use some kind of **semi-random, directed search**

Lens of Statistics: fuzzing: input structure

Modern fuzzers deal with structured input in a few ways:

Lens of Statistics: fuzzing: input structure

Modern fuzzers deal with structured input in a few ways:

- **mutating seed inputs**:

Lens of Statistics: fuzzing: input structure

Modern fuzzers deal with structured input in a few ways:

- **mutating seed inputs**:
    - start with a ***seed pool*** of valid or useful inputs

Lens of Statistics: fuzzing: input structure

Modern fuzzers deal with structured input in a few ways:

- **mutating seed inputs**:
    - start with a ***seed pool*** of valid or useful inputs
    - new test cases are **evolved** from old ones

Lens of Statistics: fuzzing: input structure

Modern fuzzers deal with structured input in a few ways:

- **mutating seed inputs**:
  - start with a ***seed pool*** of valid or useful inputs
  - new test cases are **evolved** from old ones
- **reward** or **fitness functions**:

Lens of Statistics: fuzzing: input structure

Modern fuzzers deal with structured input in a few ways:

- **mutating seed inputs**:
  - start with a ***seed pool*** of valid or useful inputs
  - new test cases are **evolved** from old ones
- **reward** or **fitness functions**:
  - when an input **increases coverage** (or some other test goal), choose more inputs like that (e.g., add it to the seed pool)

Lens of Statistics: fuzzing: input structure

Modern fuzzers deal with structured input in a few ways:

- **mutating seed inputs**:
  - start with a ***seed pool*** of valid or useful inputs
  - new test cases are **evolved** from old ones
- **reward** or **fitness functions**:
  - when an input **increases coverage** (or some other test goal), choose more inputs like that (e.g., add it to the seed pool)
- **combination with path predicates**:

Lens of Statistics: fuzzing: input structure

Modern fuzzers deal with structured input in a few ways:

- **mutating seed inputs**:
  - start with a ***seed pool*** of valid or useful inputs
  - new test cases are **evolved** from old ones
- **reward** or **fitness functions**:
  - when an input **increases coverage** (or some other test goal), choose more inputs like that (e.g., add it to the seed pool)
- **combination with path predicates**:
  - add inputs that are guaranteed to increase coverage to the seed pool

Lens of Statistics: fuzzing in practice

Lens of Statistics: fuzzing in practice

- Fuzzing is **common in industry**
    - AFL (most famous coverage-guided fuzzer) was built at Google
    - oss-fuzz project fuzzes many important open-source projects constantly using industry resources

Lens of Statistics: fuzzing in practice

- Fuzzing is **common in industry**
  - AFL (most famous coverage-guided fuzzer) was built at Google
  - oss-fuzz project fuzzes many important open-source projects constantly using industry resources
- Fuzzing is **machine-intensive**
  - most inputs aren’t useful

Lens of Statistics: fuzzing in practice

- Fuzzing is **common in industry**
  - AFL (most famous coverage-guided fuzzer) was built at Google
  - oss-fuzz project fuzzes many important open-source projects constantly using industry resources
- Fuzzing is **machine-intensive**
  - most inputs aren't useful
- Fuzzing **finds real bugs**
  - especially useful for finding security bugs

Test input generation

- As a human, often **choosing good test inputs** is the hardest part of writing a test
- For a computer, that's not true: computers can pick inputs **very fast** (given some policy)
- **Key problem**: which inputs should we pick?
    - Lens of **Logic**: choose inputs that will maximize coverage
    - Lens of **Statistics**: choose inputs "at random"
    - Lens of **Adversity**: choose inputs that kill mutants

Lens of Adversity: killing mutants

- Actually, **not as useful as it seems** for automatic test generation
  - still need to use either path predicates or fuzzing to choose inputs

Lens of Adversity: killing mutants

- Actually, **not as useful as it seems** for automatic test generation
    - still need to use either path predicates or fuzzing to choose inputs
- Can be a useful **fitness function** or guide for other automated test input generation approaches

Testing (part 3)

Today's agenda:

- Finish up code level design discussion from lecture 2
- Test input generation (fuzzing)
- Test oracle generation
- **Test prioritization & test suite minimization**
- Reading Quiz

Too many tests

- At this point, we may actually have **too many** test cases

Too many tests

- At this point, we may actually have **too many** test cases
  - Surprisingly, this is **normal in industry**: you almost always have far too few or far too many!

Too many tests

- At this point, we may actually have **too many** test cases
  - Surprisingly, this is **normal in industry**: you almost always have far too few or far too many!
- This is especially true when using automated test generation tools

Too many tests

- At this point, we may actually have **too many** test cases
  - Surprisingly, this is **normal in industry**: you almost always have far too few or far too many!
- This is especially true when using automated test generation tools
  - Which many produce many tests but **lower-quality** ones than humans would produce

Too many tests

- At this point, we may actually have **too many** test cases
  - Surprisingly, this is **normal in industry**: you almost always have far too few or far too many!
- This is especially true when using automated test generation tools
  - Which many produce many tests but **lower-quality** ones than humans would produce
  - A **big cost problem**!

Test suite minimization

Definition: given a set of test cases and coverage information for each one, the ***test suite minimization problem*** is to find the minimal number of test cases that still have the maximum coverage.

Test suite minimization

Definition: given a set of test cases and coverage information for each one, the ***test suite minimization problem*** is to find the minimal number of test cases that still have the maximum coverage.

Example:

- T1 covers lines 1,2,3
- T2 covers lines 2,3,4,5
- T3 covers lines 1,2
- T4 covers lines 1, 6

Test suite minimization

Definition: given a set of test cases and coverage information for each one, the ***test suite minimization problem*** is to find the minimal number of test cases that still have the maximum coverage.

Example:

- T1 covers lines 1,2,3
- T2 covers lines 2,3,4,5
- T3 covers lines 1,2
- T4 covers lines 1, 6

Which of these tests would you pick to minimize the number that need to be run?

Test suite minimization

Definition: given a set of test cases and coverage information for each one, the ***test suite minimization problem*** is to find the minimal number of test cases that still have the maximum coverage.

Example:

- ~~T1 covers lines 1,2,3~~
- T2 covers lines 2,3,4,5
- ~~T3 covers lines 1,2~~
- T4 covers lines 1, 6

Which of these tests would you pick to minimize the number that need to be run?

Test suite prioritization

Definition: given a budget of time, number of tests to run, or similar, the ***test suite prioritization problem*** is deciding which tests to run to maximize coverage while staying within the budget

Test suite prioritization

Definition: given a budget of time, number of tests to run, or similar, the ***test suite prioritization problem*** is deciding which tests to run to maximize coverage while staying within the budget

- very similar to test suite minimization (same techniques are useful for both)

Test suite prioritization

Definition: given a budget of time, number of tests to run, or similar, the ***test suite prioritization problem*** is deciding which tests to run to maximize coverage while staying within the budget

- very similar to test suite minimization (same techniques are useful for both)
- **question**: how **hard** are these problems?

Test suite prioritization

Definition: given a budget of time, number of tests to run, or similar, the ***test suite prioritization problem*** is deciding which tests to run to maximize coverage while staying within the budget

- very similar to test suite minimization (same techniques are useful for both)
- **question**: how **hard** are these problems?
  - theory strikes again!

Test suite prioritization

Definition: given a budget of time, number of tests to run, or similar, the ***test suite prioritization problem*** is deciding which tests to run to maximize coverage while staying within the budget

- very similar to test suite minimization (same techniques are useful for both)
- **question**: how **hard** are these problems?
    - theory strikes again!
    - answer: it's "hard" (similar "traditional" problem that you might consider a reduction to: **knapsack**)

Reading quiz

Q1: Approximately what is the ratio of source to test code in SQLite?

A. about 590 lines of source code to 1 line of test code
B. about 1 line of source code to 1 line of test code
C. about 1 line of source code to 590 lines of test code

Q2: **TRUE** or **FALSE**: A well-written C program will typically contain some defensive conditionals which in practice are always true or always false. This leads to a programming dilemma: does SQLite remove defensive code in order to obtain 100% branch coverage?

Reading quiz

Q1: Approximately what is the ratio of source to test code in SQLite?

A. about 590 lines of source code to 1 line of test code
B. about 1 line of source code to 1 line of test code
C. about 1 line of source code to 590 lines of test code

Q2: **TRUE** or **FALSE**: A well-written C program will typically contain some defensive conditionals which in practice are always true or always false. This leads to a programming dilemma: does SQLite remove defensive code in order to obtain 100% branch coverage?

Reading quiz

Q1: Approximately what is the ratio of source to test code in SQLite?

A. about 590 lines of source code to 1 line of test code
B. about 1 line of source code to 1 line of test code
C. about 1 line of source code to 590 lines of test code

Q2: **TRUE** or **FALSE**: A well-written C program will typically contain some defensive conditionals which in practice are always true or always false. This leads to a programming dilemma: does SQLite remove defensive code in order to obtain 100% branch coverage?

Takeaways

- two typical ways to generate test inputs:
  - solve path constraints
  - “at random” via fuzzing
- both common in practice
- both suffer from the oracle problem
  - implicit oracles are most common solution
  - invariants, differential testing, etc. also options
- in practice, you often have too many tests
  - deciding which to run is a hard problem, too