Testing (Part 3/3)

Martin Kellogg

Testing (part 3)

Today's agenda:

- Reading Quiz
- Finish up slides from last lecture
- Test input generation (fuzzing)
- Test oracle generation
- Test prioritization & test suite minimization

Testing (part 3)

Today's agenda:

- **Reading Quiz**
- Finish up slides from last lecture
- Test input generation (fuzzing)
- Test oracle generation
- Test prioritization & test suite minimization

Reading quiz

Q1: **TRUE** or **FALSE**: The concept of fuzz testing has been around for decades, but fuzz testing wasn't effective until 2014 when the first practical profile-guided fuzzer ("AFL") was invented.

Q2: Which tool do the authors call "perhaps the most amazing and useful developer tool in the world"?

A. AFL
B. Valgrind
C. Clang Static Analyzer
D. gcov

Reading quiz

Q1: **TRUE** or **FALSE**: The concept of fuzz testing has been around for decades, but fuzz testing wasn't effective until 2014 when the first practical profile-guided fuzzer ("AFL") was invented.

Q2: Which tool do the authors call "perhaps the most amazing and useful developer tool in the world"?

A. AFL
B. Valgrind
C. Clang Static Analyzer
D. gcov

Reading quiz

Q1: **TRUE** or **FALSE**: The concept of fuzz testing has been around for decades, but fuzz testing wasn't effective until 2014 when the first practical profile-guided fuzzer ("AFL") was invented.

Q2: Which tool do the authors call "perhaps the most amazing and useful developer tool in the world"?

A. AFL
B. Valgrind
C. Clang Static Analyzer
D. gcov

Testing (part 3)

Today's agenda:

- Reading Quiz
- **Finish up slides from last lecture**
- Test input generation (fuzzing)
- Test oracle generation
- Test prioritization & test suite minimization

Ways to think about test suite quality

Today we're going to consider three ways to think about test suite quality:

- test suite quality through the lens of logic
- test suite quality through the lens of **statistics**
- test suite quality through the lens of adversity

The Lens of Statistics

- **Key idea:** Sample test inputs from the population of inputs users will actually provide in the real world

The Lens of Statistics

- **Key idea:** Sample test inputs from the population of inputs users will actually provide in the real world
  - This approach inherits both advantages and disadvantages from other kinds of statistical techniques

The Lens of Statistics

- **Key idea:** Sample test inputs from the population of inputs users will actually provide in the real world
  - This approach inherits both advantages and disadvantages from other kinds of statistical techniques

Key advantages:

- confidence that tests are indicative of the real world
- can use statistical techniques to estimate the chance that our tests don't cover some important behavior

The Lens of Statistics: disadvantages

The Lens of Statistics: disadvantages

- In statistics, **sampling error** is incurred when the statistical characteristics of a population are estimated from a subset, or sample, of that population.

The Lens of Statistics: disadvantages

- In statistics, **sampling error** is incurred when the statistical characteristics of a population are estimated from a subset, or sample, of that population.
    - “Our test suite is a sample of inputs that could occur in the real world. Our program behaves well on our test suite.”

The Lens of Statistics: disadvantages

- In statistics, **sampling error** is incurred when the statistical characteristics of a population are estimated from a subset, or sample, of that population.
  - “Our test suite is a sample of inputs that could occur in the real world. Our program behaves well on our test suite.” → later →

The Lens of Statistics: disadvantages

- In statistics, **sampling error** is incurred when the statistical characteristics of a population are estimated from a subset, or sample, of that population.
    - "Our test suite is a sample of inputs that could occur in the real world. Our program behaves well on our test suite." → later → "Our program behaves badly on some other untested real input. Sampling error!"

The Lens of Statistics: disadvantages

- In statistics, **sampling error** is incurred when the statistical characteristics of a population are estimated from a subset, or sample, of that population.
  - "Our test suite is a sample of inputs that could occur in the real world. Our program behaves well on our test suite." → later → "Our program behaves badly on some other untested real input. Sampling error!"
- Testing gives confidence the same way sampling (or polling) gives confidence.

The Lens of Statistics: disadvantages

- In statistics, **sampling bias** is a bias in which a sample is collected in such a way that some members of the intended population are less likely to be included than others.

The Lens of Statistics: disadvantages

- In statistics, **sampling bias** is a bias in which a sample is collected in such a way that some members of the intended population are less likely to be included than others.
    - Suppose you are conducting a poll to see who will win the next election, but you only ask the current governor's staffers

The Lens of Statistics: disadvantages

- In statistics, **sampling bias** is a bias in which a sample is collected in such a way that some members of the intended population are less likely to be included than others.
  - Suppose you are conducting a poll to see who will win the next election, but you only ask the current governor's staffers
  - Suppose you are creating tests to see if your program will crash, but you only poll nice, small, inputs

The Lens of Statistics: disadvantages

- Possible solution: there are a number of **well-established sampling techniques** in the field of statistics to help address such biases

The Lens of Statistics: disadvantages

- Possible solution: there are a number of **well-established sampling techniques** in the field of statistics to help address such biases
  - Unfortunately, they often require knowing something about the **distribution** of the full population from which you want to sample a subpopulation

The Lens of Statistics: disadvantages

- Possible solution: there are a number of **well-established sampling techniques** in the field of statistics to help address such biases
  - Unfortunately, they often require knowing something about the **distribution** of the full population from which you want to sample a subpopulation
- The basic problem in SE is that the underlying distribution of real user inputs is **not known**

The Lens of Statistics: practical options

The Lens of Statistics: practical options

Definition: ***Beta testing*** is testing done by external users (often using a special beta version of the program).

The Lens of Statistics: practical options

Definition: ***Beta testing*** is testing done by external users (often using a special beta version of the program).

- in contrast to ***alpha testing***, which is usually performed by developers or a quality assurance team

The Lens of Statistics: practical options

Definition: ***Beta testing*** is testing done by external users (often using a special beta version of the program).

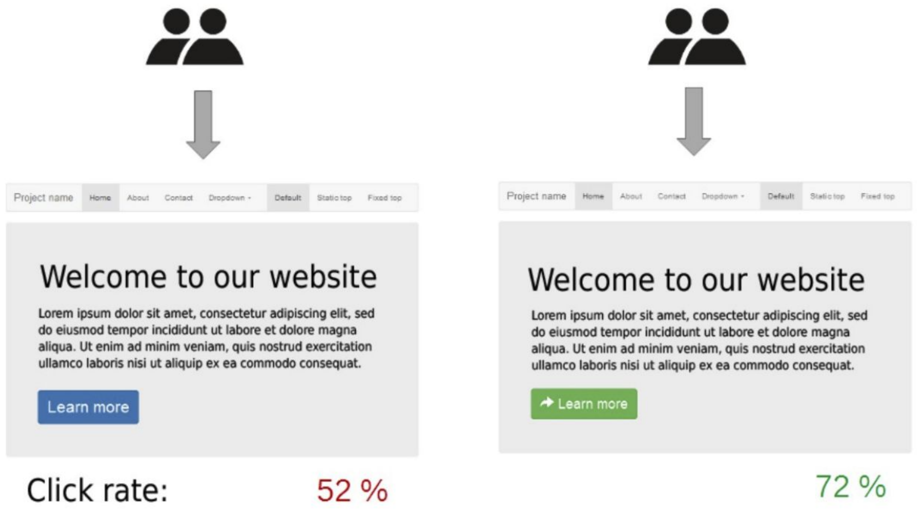
- in contrast to ***alpha testing***, which is usually performed by developers or a quality assurance team
- Beta testing can be viewed as directly sampling the space of user inputs

The Lens of Statistics: practical options

Definition: ***A/B testing*** involves two variants of your software, A and B, which differ only in one feature. Different users are shown different variants and responses are recorded.

The Lens of Statistics: practical options

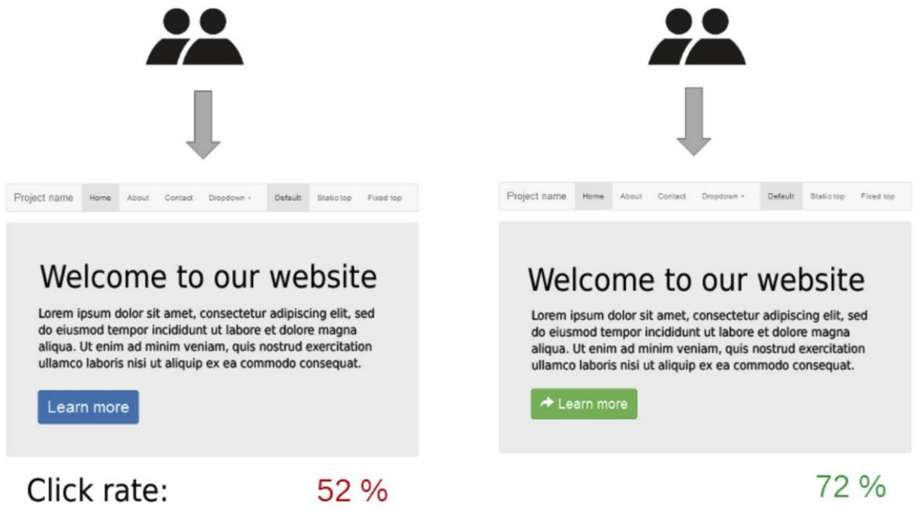
Definition: ***A/B testing*** involves two variants of your software, A and B, which differ only in one feature. Different users are shown different variants and responses are recorded.



The Lens of Statistics: practical options

Definition: ***A/B testing*** involves two variants of your software, A and B, which differ only in one feature. Different users are shown different variants and responses are recorded.

- A/B testing is an instance of two-sample hypothesis testing, like you'd encounter in a statistics class.



The Lens of Statistics: practical options

- Recall two guiding approaches:
    - Sample what users will do **most commonly**
    - Sample what will cause the **most harm**

The Lens of Statistics: practical options

- Recall two guiding approaches:
  - Sample what users will do **most commonly**
  - Sample what will cause the **most harm**
- The former is sometimes called ***workload generation***
  - Common for databases, webservers, etc.

The Lens of Statistics: practical options

- Recall two guiding approaches:
  - Sample what users will do **most commonly**
  - Sample what will cause the **most harm**
- The former is sometimes called ***workload generation***
  - Common for databases, webservers, etc.
- The latter often relates to **computer security**
  - E.g., exploit generation, penetration testing, etc.

The Lens of Statistics: practical options

- Recall two guiding approaches:
  - Sample what users will do **most commonly**
  - Sample what will cause the **most harm**
- The former is sometimes called ***workload generation***
  - Common for databases, webservers, etc.
- The latter often relates to **computer security**
  - E.g., exploit generation, penetration testing, etc.
- **Damage** can also be in other forms
  - e.g., for Amazon, “damage” might be “customer doesn’t complete the purchase”

Ways to think about test suite quality

Today we're going to consider three ways to think about test suite quality:

- test suite quality through the lens of **logic**
- test suite quality through the lens of **statistics**
- test suite quality through the lens of **adversity**

The Lens of Adversity: finding bugs

- Suppose you wanted to evaluate the quality of two truffle-sniffing pigs

The Lens of Adversity: finding bugs

- Suppose you wanted to evaluate the quality of two truffle-sniffing pigs
  - **Intuition**: test whether they can actually find truffles!

The Lens of Adversity: finding bugs

- Suppose you wanted to evaluate the quality of two truffle-sniffing pigs
  - **Intuition**: test whether they can actually find truffles!
- **Test idea**: hide some truffles in your backyard and see how many each pig finds

The Lens of Adversity: finding bugs

- Suppose you wanted to evaluate the quality of two truffle-sniffing pigs
  - **Intuition**: test whether they can actually find truffles!
- **Test idea**: hide some truffles in your backyard and see how many each pig finds
  - The pig that finds more of the hidden truffles in your backyard is assumed to find more real truffles in the wild

The Lens of Adversity: finding bugs

- Suppose you wanted to evaluate the quality of two truffle-sniffing pigs
  - **Intuition**: test whether they can actually find truffles!
- **Test idea**: hide some truffles in your backyard and see how many each pig finds
  - The pig that finds more of the hidden truffles in your backyard is assumed to find more real truffles in the wild
- Suppose you wanted to evaluate the quality of two bug-finding test suites …

The Lens of Adversity: mutation testing

Definition: ***Mutation testing*** (or ***mutation analysis***) is a test suite adequacy metric in which the quality of a test suite is related to the number of intentionally-added defects it finds

The Lens of Adversity: mutation testing

Definition: ***Mutation testing*** (or ***mutation analysis***) is a test suite adequacy metric in which the quality of a test suite is related to the number of intentionally-added defects it finds

- Informally: “You claim your test suite is really great at finding security bugs? Well, I'll just **intentionally add a security bug** to my source code and see if your test suite finds it!”

Mutation testing: verisimilitude

- In the truffle-pig example, if every truffle I hide in my backyard is next to a smelly red flower, a pig that finds them all may not actually do well in the real world

Mutation testing: verisimilitude

- In the truffle-pig example, if every truffle I hide in my backyard is next to a smelly red flower, a pig that finds them all may not actually do well in the real world
  - The truffle placements I made up were **not indicative** of real-world truffles

Mutation testing: verisimilitude

- In the truffle-pig example, if every truffle I hide in my backyard is next to a smelly red flower, a pig that finds them all may not actually do well in the real world
  - The truffle placements I made up were **not indicative** of real-world truffles
- Similarly, if I add a bunch of defects to my software that are not the sort of defects real humans would make, then mutation testing is **uninformative**

Mutation testing: verisimilitude

- In the truffle-pig example, if every truffle I hide in my backyard is next to a smelly red flower, a pig that finds them all may not actually do well in the real world
  - The truffle placements I made up were **not indicative** of real-world truffles
- Similarly, if I add a bunch of defects to my software that are not the sort of defects real humans would make, then mutation testing is **uninformative**
  - **Implication**: mutation testing requires us to know what real bugs look like

Mutation testing: defect seeding

Definition: ***Defect seeding*** is the process of intentionally introducing a defect into a program.

Mutation testing: defect seeding

Definition: ***Defect seeding*** is the process of intentionally introducing a defect into a program.

- The defect introduced is typically intentionally similar to defects introduced by real developers.

Mutation testing: defect seeding

Definition: ***Defect seeding*** is the process of intentionally introducing a defect into a program.

- The defect introduced is typically intentionally similar to defects introduced by real developers.
- The seeding is typically done by changing the source code.

Mutation testing: defect seeding

Definition: ***Defect seeding*** is the process of intentionally introducing a defect into a program.

- The defect introduced is typically intentionally similar to defects introduced by real developers.
- The seeding is typically done by changing the source code.
- For mutation testing, defect seeding is typically done automatically (given a model of what human bugs look like)

Mutation testing: defect seeding

Definition: ***Defect seeding*** is the process of intentionally introducing a defect into a program.

- The defect introduced is typically intentionally similar to defects introduced by real developers.
- The seeding is typically done by changing the source code.
- For mutation testing, defect se
  automatically (given a model o   ike)

This is **exactly** how our "fault injection" system for testing your IP1 tests works.

Mutation testing: mutation operators

Definition: A ***mutation operator*** systematically changes a program point. In mutation testing, the mutation operators are modeled on historical human defects.

Mutation testing: mutation operators

Definition: A ***mutation operator*** systematically changes a program point. In mutation testing, the mutation operators are modeled on historical human defects.

- Example mutations:
    - `if (a < b)` → `if (a <= b)`
    - `if (a == b)` → `if (a != b)`
    - `a = b + c` → `a = b - c`
    - `f(); g();` → `g(); f();`
    - `x = y` → `x = z`

Mutation testing: mutants

Definition: A ***mutant*** (or ***variant***) is a version of the original program produced by applying one or more mutation operators to one or more program locations.

Mutation testing: mutants

Definition: A ***mutant*** (or ***variant***) is a version of the original program produced by applying one or more mutation operators to one or more program locations.

Definition: The ***order*** of a mutant is the number of mutation operators applied.

Mutation testing: mutants

Definition: A ***mutant*** (or ***variant***) is a version of the original program produced by applying one or more mutation operators to one or more program locations.

Definition: The ***order*** of a mutant is the number of mutation operators applied.

```
// original
if (a < b):
x = a + b
print(x)
```

→

```
// 2nd-order mutant
if (a <= b):
x = a - b
print(x)
```

Mutation testing: competent programmers

- The **competent programmer hypothesis** holds that program faults are syntactically small and can be corrected with a few keystrokes.

Mutation testing: competent programmers

- The **competent programmer hypothesis** holds that program faults are syntactically small and can be corrected with a few keystrokes.
    - “Programmers write programs that are largely correct. Thus the mutants simulate the likely effect of real faults.”

Mutation testing: competent programmers

- The **competent programmer hypothesis** holds that program faults are syntactically small and can be corrected with a few keystrokes.
    - "Programmers write programs that are largely correct. Thus the mutants simulate the likely effect of real faults."
    - Therefore, **if the test suite is good at catching the artificial mutants, it will also be good at catching the unknown but real faults** in the program.

Mutation testing: competent programmers

- The **competent programmer hypothesis** holds that program faults are syntactically small and can be corrected with a few keystrokes
  - “ … t. Thus t …
  - T … **tificial r … n but r …**

Is the competent programmer hypothesis true?

- Yes and no.
- It is true that humans often make simple typos (e.g., + vs -).
- But it is also true that some bugs are much more complex than that.

Mutation testing: competent programmers

- The **competent programmer hypothesis** holds that program faults are syntactically small and can be corrected with a few keystrokes
    - “ t. Thus t
    - T **ificial r n but r**

Is the competent programmer hypothesis true?

- Yes and no.
- It is true that humans often make simple typos (e.g., + vs -).
- But it is also true that some bugs are much **more complex** than that!

Mutation testing: coupling effect hypothesis

Mutation testing: coupling effect hypothesis

- The **coupling effect hypothesis** holds that complex faults are "coupled" to simple faults in such a way that a test suite that detects all simple faults in a program will detect a high percentage of the complex faults.

Mutation testing: coupling effect hypothesis

- The **coupling effect hypothesis** holds that complex faults are "coupled" to simple faults in such a way that a test suite that detects all simple faults in a program will detect a high percentage of the complex faults.
- Is this true?

Mutation testing: coupling effect hypothesis

- The **coupling effect hypothesis** holds that complex faults are "coupled" to simple faults in such a way that a test suite that detects all simple faults in a program will detect a high percentage of the complex faults.
- Is this true?
  - Tests that detect simple mutants were **also** able to detect over 99% of second- and third-order mutants historically

[J. Offutt. Investigations of the software testing coupling effect. ACM Trans. Softw. Eng. Methodol., 1(1):5–20, Jan. 1992.]

Mutation testing: putting it all together

- A test suite is said to ***kill*** (or ***detect***, or ***reveal***) a mutant if the mutant fails a test that the original passes.

Mutation testing: putting it all together

- A test suite is said to ***kill*** (or ***detect***, or ***reveal***) a mutant if the mutant fails a test that the original passes.
- ***Mutation testing*** (or ***mutation analysis***) of a test suite proceeds by making a number of mutants and measuring the fraction of them killed by that test suite. This fraction is called the ***mutation adequacy score*** (or ***mutation score***).

Mutation testing: putting it all together

- A test suite is said to ***kill*** (or ***detect***, or ***reveal***) a mutant if the mutant fails a test that the original passes.
- ***Mutation testing*** (or ***mutation analysis***) of a test suite proceeds by making a number of mutants and measuring the fraction of them killed by that test suite. This fraction is called the ***mutation adequacy score*** (or ***mutation score***).
  - A test suite with a **higher score is better**.
  - (Sorry for all of the vocabulary!)

Mutation testing: pros and cons

Mutation testing: pros and cons

- Has the potential to subsume other test suite adequacy criteria (it **can be very good**)

Mutation testing: pros and cons

- Has the potential to subsume other test suite adequacy criteria (it **can be very good**)
- **Difficult** to do well:
  - Which mutation operators do you use?
  - Where do you apply them? How often do you apply them?
    - Typically done at random, but how?

Mutation testing: pros and cons

- Has the potential to subsume other test suite adequacy criteria (it **can be very good**)
- **Difficult** to do well:
  - Which mutation operators do you use?
  - Where do you apply them? How often do you apply them?
    - Typically done at random, but how?
- It is **very expensive**. If you make 1,000 mutants, you must now run your test suite 1,000 times!
  - We started by saying testing (1x) was expensive!

Mutation testing: equivalent mutant problem

- Suppose you have "`x = a + b; y = c + d;`" and you swap those two statements.

Mutation testing: equivalent mutant problem

- Suppose you have “`x = a + b; y = c + d;`” and you swap those two statements.
- The resulting program is a mutant, but it is **semantically equivalent** to the original.

Mutation testing: equivalent mutant problem

- Suppose you have "`x = a + b; y = c + d;`" and you swap those two statements.
- The resulting program is a mutant, but it is **semantically equivalent** to the original.
    - So it will pass and fail all of the tests that the original passes and fails.

Mutation testing: equivalent mutant problem

- Suppose you have "`x = a + b; y = c + d;`" and you swap those two statements.
- The resulting program is a mutant, but it is **semantically equivalent** to the original.
    - So it will pass and fail all of the tests that the original passes and fails.
    - So it will dilute the mutation score

Mutation testing: equivalent mutant problem

- Suppose you have "`x = a + b; y = c + d;`" and you swap those two statements.
- The resulting program is a mutant, but it is **semantically equivalent** to the original.
    - So it will pass and fail all of the tests that the original passes and fails.
    - So it will dilute the mutation score
- Detecting these "***equivalent mutants***" is a big deal. How hard is it?

Mutation testing: equivalent mutant problem

- Suppose you have "`x = a + b; y = c + d;`" and you swap those two statements.
- The resulting program i… **ally equivalent** to the origi…
  - So it will pass and fa… ginal passes and fails.
  - So it will dilute the mutation score
- Detecting these "***equivalent mutants***" is a big deal. How hard is it?

Remember when I mentioned **reductions** earlier? Now is a good time to do one!

Mutation testing: equivalent mutant problem

- Detecting these "***equivalent mutants***" is a big deal. How hard is it?

Mutation testing: equivalent mutant problem

- Detecting these "***equivalent mutants***" is a big deal. How hard is it?
- It is **undecidable**! (= there is no algorithm for it that can always give the correct answer)

Mutation testing: equivalent mutant problem

- Detecting these "***equivalent mutants***" is a big deal. How hard is it?
- It is **undecidable**! (= there is no algorithm for it that can always give the correct answer)
    - by direct reduction to the **Halting Problem** (or by **Rice's theorem**)

```
def foo():                # foo halts if and only if
    if p1() == p2():   # p1 is equivalent to p2
        return 0
    foo()
```

Takeaways

- Individual tests should be hermetic and focused
  - avoid flaky and brittle tests
- Three lenses for test suite quality: logic, statistics, and adversity
- Lens of **Logic**: “no visit X → no find bug in X”
  - leads to statement and branch coverage.
- Lens of **Statistics**: “sample the inputs the users will make”
  - leads to beta testing, A/B testing.
- Lens of **Adversity**: “poke realistic holes in the program and see if you find them”
  - leads to mutation testing.

Testing (part 3)

Today's agenda:

- Reading Quiz
- Finish up slides from last lecture
- **Test input generation** (fuzzing)
- Test oracle generation
- Test prioritization & test suite minimization

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 … n the user's keyboard b … 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
3. 1 & 2 imply test input generation must also control the environment

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 orld through *system cal*
  - 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

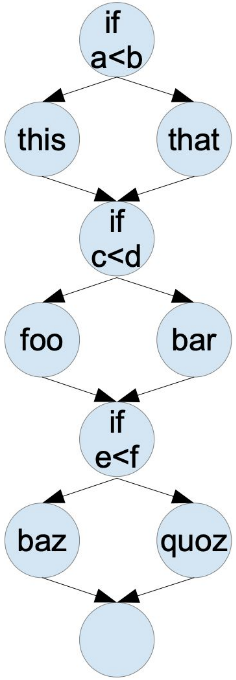
Lens of Logic: maximize coverage

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
```

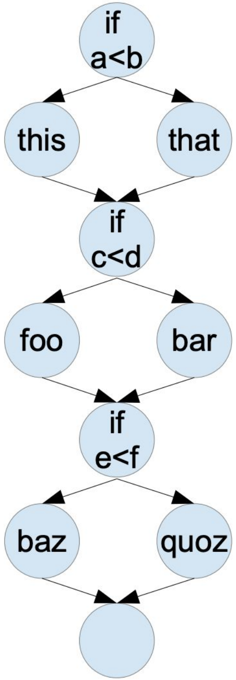
Lens of Logic: maximize coverage

```
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
```

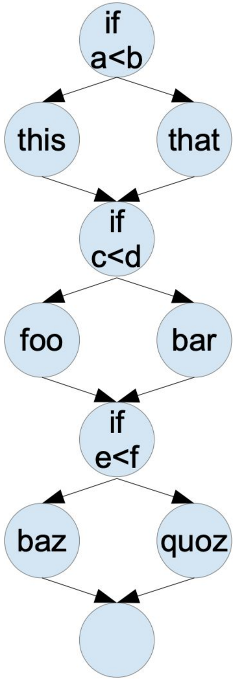


How would you choose inputs that **maximize**:

- 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
```

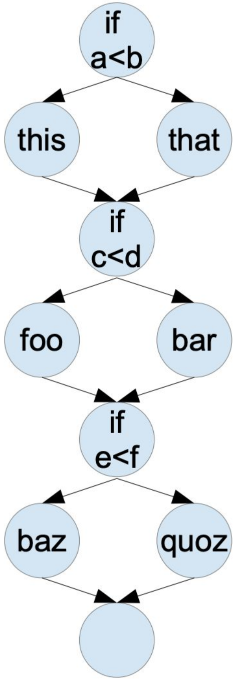


How would you choose inputs that **maximize**:

- **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
```

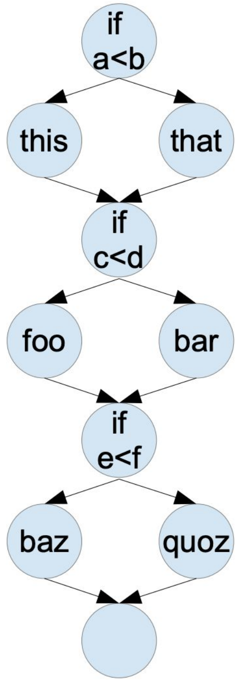


How would you choose inputs that **maximize**:

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

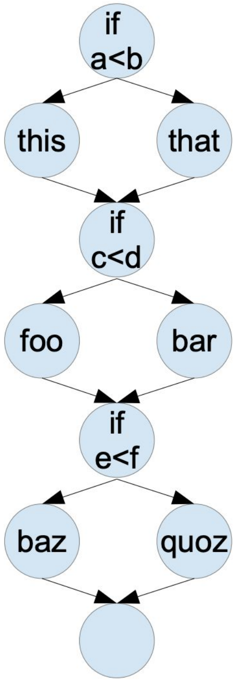
Lens of Logic: maximize coverage

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



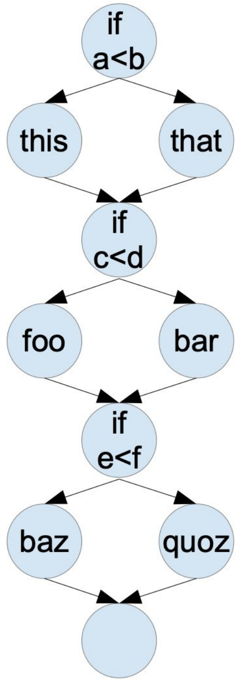
Lens of Logic: maximize coverage

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



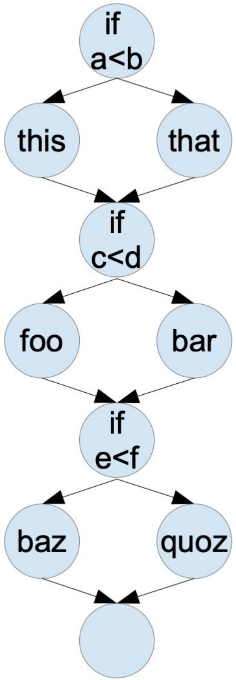
Lens of Logic: maximize coverage

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



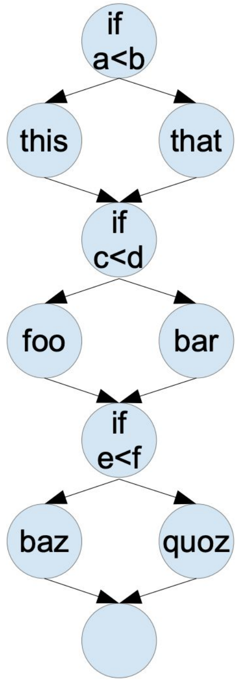
Lens of Logic: maximize coverage

- 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



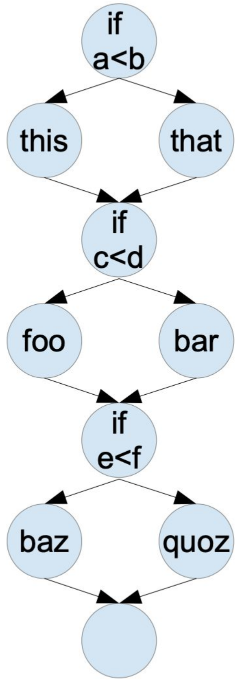
Lens of Logic: maximize coverage

- 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**



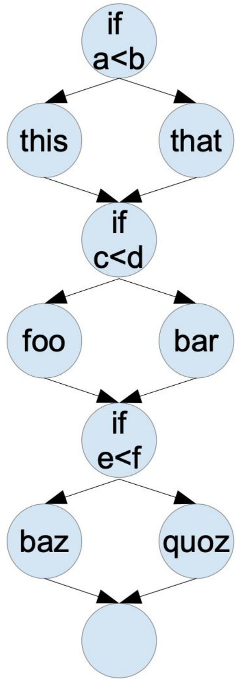
Lens of Logic: maximize coverage

- 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**

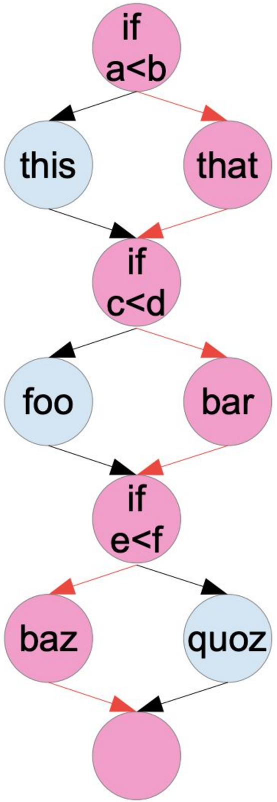
Lens of Logic: maximize coverage

- 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

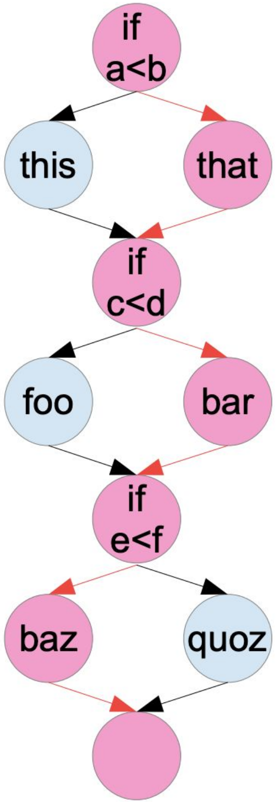
Lens of Logic: path predicate example

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



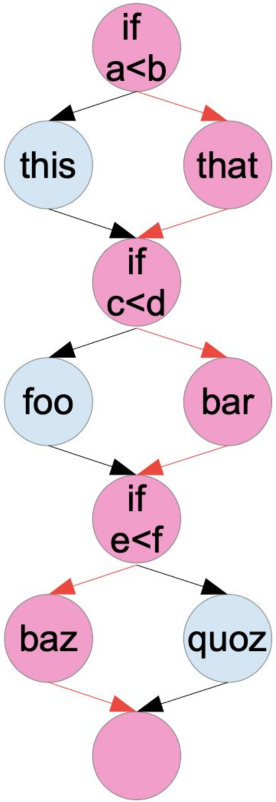
Lens of Logic: path predicate example

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



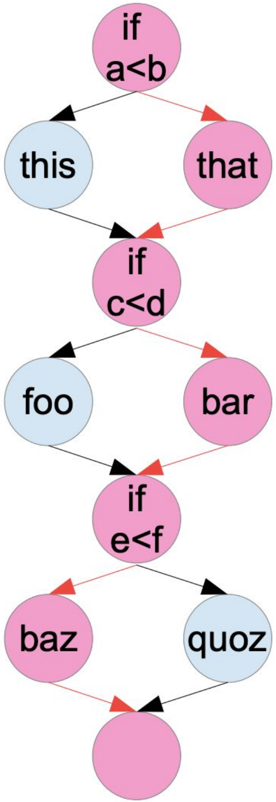
Lens of Logic: path predicate example

- 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: solving path predicates

- How do we find satisfying assignments in general?
  - Option 1: **ask humans**
    - labor-intensi
  - Option 2: repeat
    - works surpris
  - 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.)

Ask me about how an **SMT solver** works in office hours if you want to know more!

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.

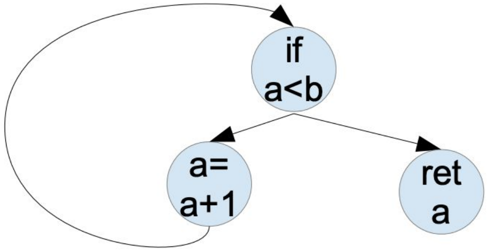
Lens of Logic: enumerating paths

- 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!

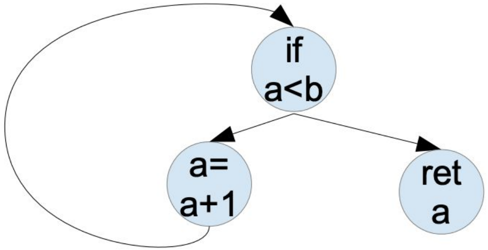
```
while a < b:
  a = a + 1
return a
```



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

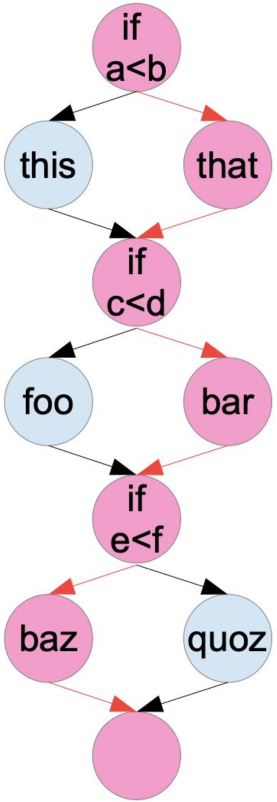
- **Key idea**: don't enumerate all paths, **approximate** instead
- Typical Approximations:
  - Consider only **acyclic** paths (corresponds to taking each loop zero times or one time)
  - Consider only taking each loop **at most *k*** times
  - Enumerate paths breadth-first or depth-first and **stop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

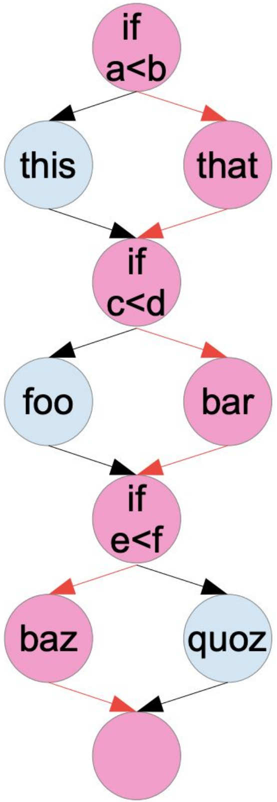
Lens of Logic: collecting path predicates

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



Lens of Logic: collecting path predicates

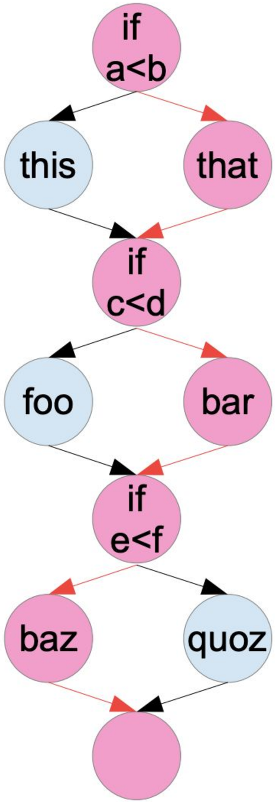
- Now we have a path through the program
- What could go wrong with **collecting** the path predicate?
  - The path predicate may not be **expressible** in terms of the inputs we control



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

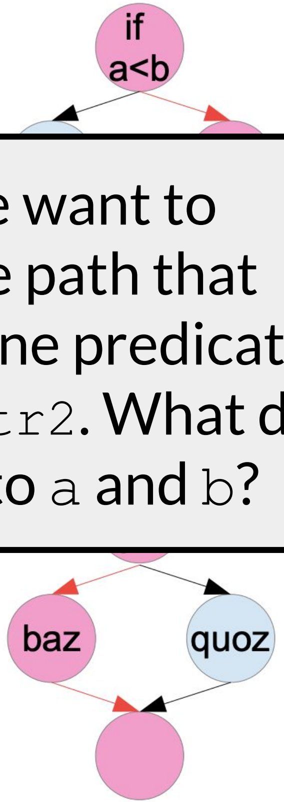
```
foo(a,b):
  str1 = read_from_url("abc.com")
  str2 = read_from_url("xyz.com")
  if (str1 == str2): bar()
```



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

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



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

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:

- Reading Quiz
- Finish up slides from last lecture
- Test input generation (fuzzing)
- **Test oracle generation**
- Test prioritization & test suite minimization

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 (October 6 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:

- Reading Quiz
- Finish up slides from last lecture
- Test input generation (**fuzzing**)
- Test oracle generation
- Test prioritization & test suite minimization

This is how far we got on 9/20/24. This lecture will resume on October 2.

Test input generation

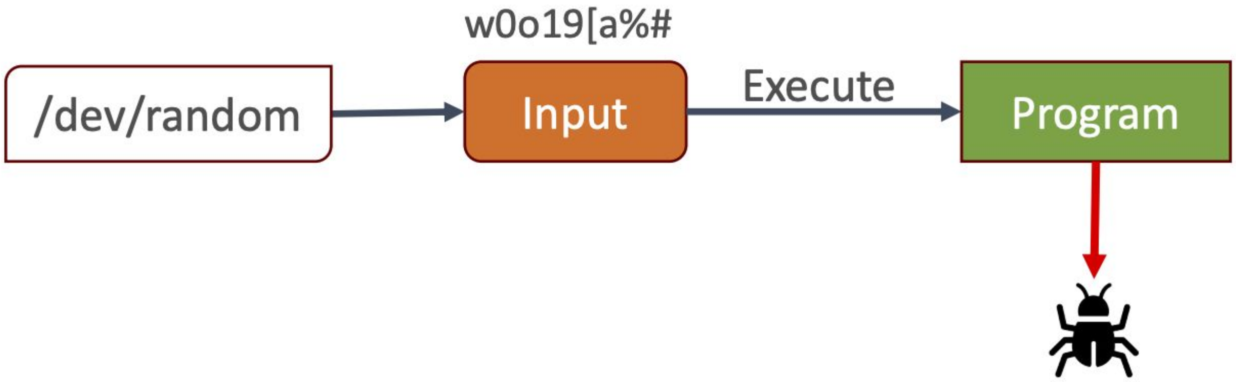
- As a human, often **choosing good test inputs** is the hardest part of writing a test
- For a computer, that’s not true: computers can pick inputs **very fast** (given some policy)
- **Key problem**: which inputs should we pick?
    - Lens of **Logic**: choose inputs that will maximize coverage
  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

Aside: red-bordered slides

- The **red border** on the this slide indicates material that is **not** fair game for exam questions
  - Generally I will only use this border in 490 for slides that get skipped due to time constraints
  - Be warned: red-bordered slides may make another appearance later on in another lecture!

Testing (part 3)

Today's agenda:

- Reading Quiz
- Finish up slides from last lecture
- Test input generation (fuzzing)
- Test oracle generation
- **Test prioritization & test suite minimization**

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**)

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