Coverage

Martin Kellogg

Quiz!

Not just about the reading! Other questions about last week's lecture.

You may use any **hand-written** notes that you took during last week's class. No other aids are permitted.

- you must turn in hand-written notes that you use along with your quiz

Three ways to think about test suite quality

Consider three ways to think about test suite quality:

- test suite quality through the lens of logic
  - intuition: if we don't test it, we can't find bugs
  - leads to ***coverage*** (main subject of today's lecture)
- test suite quality through the lens of **statistics**
  - intuition: test what happens to real users
- test suite quality through the lens of adversity
  - intuition: inject bugs and see if the test suite catches them
  - leads to ***mutation testing***, which we'll cover later this semester

The Lens of Logic

Informally, we want the following property:

- The program passes the tests if and only if it does **all the right things** and **none of the wrong things**.

The Lens of Logic

Informally, we want the following property:

- The program passes the tests if and only if it does **all the right things** and **none of the wrong things**.
  - Pass all tests → program adheres to requirements
  - Each failing test → program behaves incorrectly

The Lens of Logic: intuition

- Suppose you were writing a sqrt program and one of the requirements was that it should abort gracefully on negative inputs.

The Lens of Logic: intuition

- Suppose you were writing a sqrt program and one of the requirements was that it should abort gracefully on negative inputs.
- Suppose further that your test suite does not include any negative inputs.

The Lens of Logic: intuition

- Suppose you were writing a sqrt program and one of the requirements was that it should abort gracefully on negative inputs.
- Suppose further that your test suite does not include any negative inputs.
- Can we conclude that passing all of the tests implies adhering to all of the requirements?

The Lens of Logic: coverage

- We desire all of the requirements to be covered (“checked”) by the test suite.

The Lens of Logic: coverage

- We desire all of the requirements to be covered (“checked”) by the test suite.
- For our purposes, ***X* coverage** is the degree to which ***X*** is executed/exercised by the test suite.

The Lens of Logic: coverage

- We desire all of the requirements to be covered (“checked”) by the test suite.
- For our purposes, **X coverage** is the degree to which **X** is executed/exercised by the test suite.
- **Code coverage** is the degree to which the source code is executed by the test suite.

The Lens of Logic: coverage

- We desire all of the requirements to be covered (“checked”) by the test suite.
- For our purposes, ***X* coverage** is the degree to which ***X*** is executed/exercised by the test suite.
- **Code coverage** is the degree to which the source code is executed by the test suite.
    - How do we actually **measure** code coverage?

Coverage: statement coverage

Definition: ***Statement coverage*** is the fraction of source statements that are executed by the test suite.

Coverage: statement coverage

Definition: ***Statement coverage*** is the fraction of source statements that are executed by the test suite.

- **Key Logical Observation**: If we **never test** line X then testing **cannot rule out** the presence of a bug on line X

Coverage: statement coverage

Definition: ***Statement coverage*** is the fraction of source statements that are executed by the test suite.

- **Key Logical Observation**: If we **never test** line X then testing **cannot rule out** the presence of a bug on line X
- Example: if our test executes lines 1 and 2, but there is a bug on line 3, there is **no way** that our test will find the bug!

Aside: “don’t do bad things”

- We can test that programs **do not do certain bad things**
  - e.g., “don't segfault”, “don't send my password to Microsoft”, “on this one particular input, don't get the wrong answer”

Aside: “don’t do bad things”

- We can test that programs **do not do certain bad things**
  - e.g., “don't segfault”, “don't send my password to Microsoft”, “on this one particular input, don't get the wrong answer”
- Note that “I never do bad things” is not the same as “I always/eventually do good things”
  - For more information, take a class on *Modal Logic* or read about *Liveness* vs. *Safety properties*

Coverage: statement coverage

Implication for statement coverage: you could test line X and still have a bug on line X

- e.g., foo(a,b) { return a/b; }
- test: foo(6,2) does not throw DivideByZeroException

Coverage: statement coverage

Implication for statement coverage: you could test line X and still have a bug on line X

- e.g., foo(a,b) { return a/b; }
- test: foo(6,2) does not throw DivideByZeroException

But testing line X gives us some **small but non-zero confidence** in the correctness of line X

Coverage: statement coverage: assumptions

We've made some **assumptions** in our discussion of statement coverage so far:

Coverage: statement coverage: assumptions

We've made some **assumptions** in our discussion of statement coverage so far:

- We gain the same amount of confidence (or information) for each visited line
- The amount of confidence (or information) we gain per visited line is positive
- ...

Coverage: computing statement coverage

Coverage: computing statement coverage

- At its simplest, this is just **print-statement debugging**

Coverage: computing statement coverage

- At its simplest, this is just **print-statement debugging**
- Put a print statement before every line of the program
  - Run all the tests, collect all the printed information, remove duplicates, count

Coverage: computing statement coverage

- At its simplest, this is just **print-statement debugging**
- Put a print statement before every line of the program
    - Run all the tests, collect all the printed information, remove duplicates, count
- Practical concern: the **observer effect** (from physics) is the fact that simply observing a situation or phenomenon necessarily changes that phenomenon.

Coverage: computing statement coverage

- At its simplest, this is just **print-statement debugging**
- Put a print statement before every line of the program
  - Run all the tests, collect all the printed information, remove duplicates, count
- Practical concern: the **observer effect** (from physics) is the fact that simply observing a situation or phenomenon necessarily changes that phenomenon.
  - Implication for computing statement coverage: program might depend on timing info, amount of I/O, etc.

Coverage: computing statement coverage

Definition: ***Coverage instrumentation*** modifies a program to record coverage information in a way that minimizes the observer effect.

Coverage: computing statement coverage

Definition: ***Coverage instrumentation*** modifies a program to record coverage information in a way that minimizes the observer effect.

- This can be done at the source or binary level.
- Don't actually print to stdout/stderr
- Don't slow things down too much
  - Pre-check before printing a duplicate?
- Don't introduce infinite loops
  - Instrument “print” with a call to “print”?

Coverage: computing statement coverage

Definition: ***Coverage instrumentation*** mo

coverage information in a way that minin

- This can be done at the source or bin
- Don't actually print to stdout/stderr
- Don't slow things down too much
  - Pre-check before printing a duplicate?
- Don't introduce infinite loops
  - Instrument “print” with a call to “print”?

Good news: coverage instrumentation is a “solved” problem:

- e.g., `gcov`, Python’s `coverage`, etc.

Background: control flow graphs

Definition: a ***control flow graph*** (or ***CFG***) is a representation, using graph notation, of all *paths* that might be traversed through a program during its execution

Background: control flow graphs

Definition: a ***control flow graph*** (or ***CFG***) is a representation, using graph notation, of all *paths* that might be traversed through a program during its execution

- a ***path*** in a program is a possible execution trace

Background: control flow graphs

Definition: a ***control flow graph*** (or ***CFG***) is a representation, using graph notation, of all *paths* that might be traversed through a program during its execution

- a ***path*** in a program is a possible execution trace
- the CFG is the internal representation used by many program analysis tools

Background: control flow graphs

Definition: a ***control flow graph*** (or ***CFG***) is a representation, using graph notation, of all *paths* that might be traversed through a program during its execution

- a ***path*** in a program is a possible execution trace
- the CFG is the internal representation used by many program analysis tools
- brief CFG example on the whiteboard

Example: computing statement coverage

```
public double avgAbs(double... numbers) {
  // We expect the array to be non-null and non-empty
  if (numbers == null || numbers.length == 0) {
     throw new IllegalArgumentException("numbers is null or empty!");
  }
  double sum = 0;
  for (int i = 0; i < numbers.length; ++i) {
    double d = numbers[i];
    if (d < 0) { sum -= d; }
    else { sum += d; }
 }
 return sum / numbers.length;
}
```

Example: computing stat

What does this method do?

It averages the absolute values of an array of doubles.

```java
public double avgAbs(double... numbers) {
  // We expect the array to be non-null and non-empty
  if (numbers == null || numbers.length == 0) {
     throw new IllegalArgumentException("numbers is null or empty!");
  }
  double sum = 0;
  for (int i = 0; i < numbers.length; ++i) {
    double d = numbers[i];
    if (d < 0) { sum -= d; }
    else { sum += d; }
 }
 return sum / numbers.length;
}
```

Example: computing stat

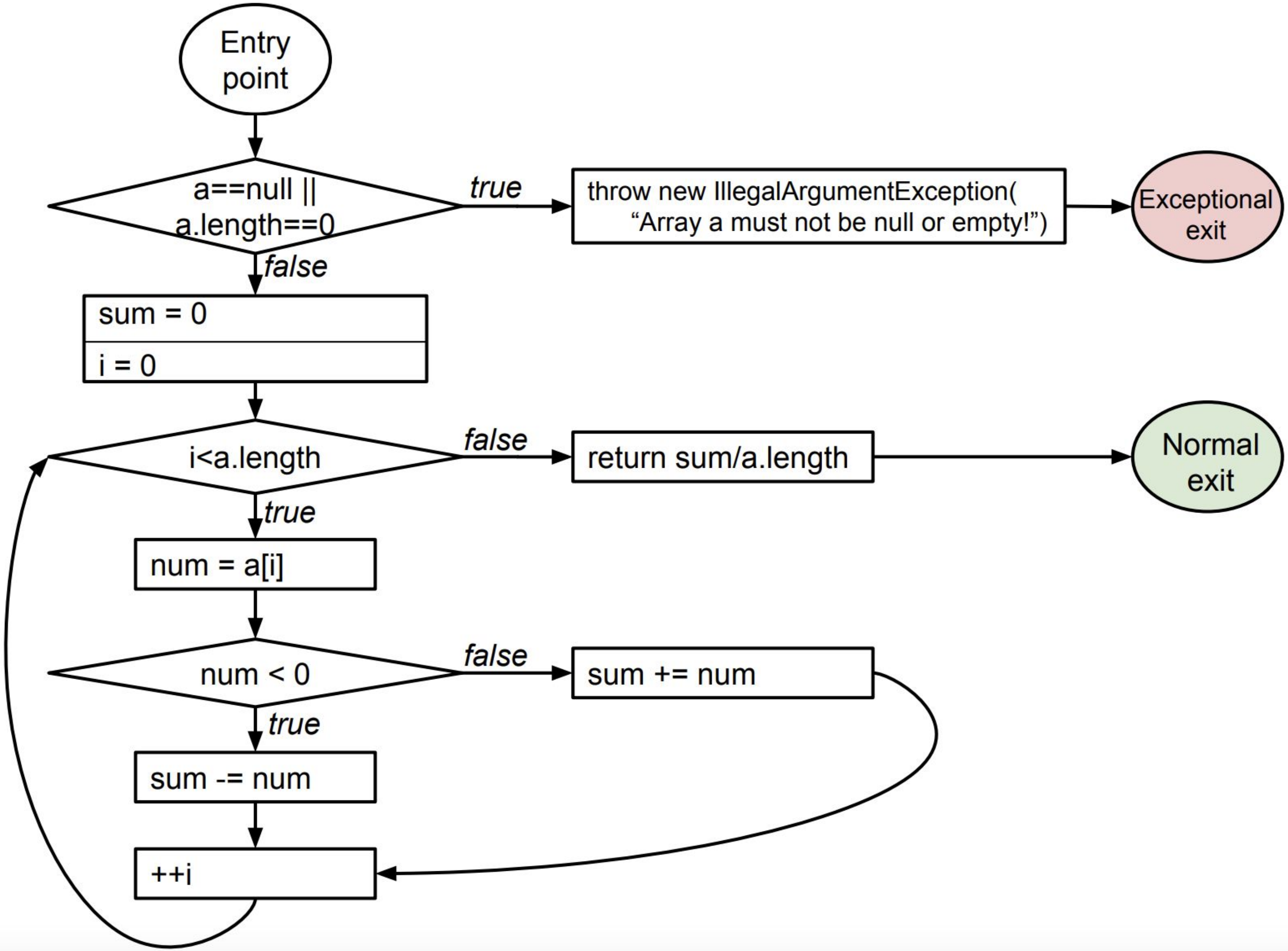
What does this method do?
It averages the absolute values of an array of doubles.

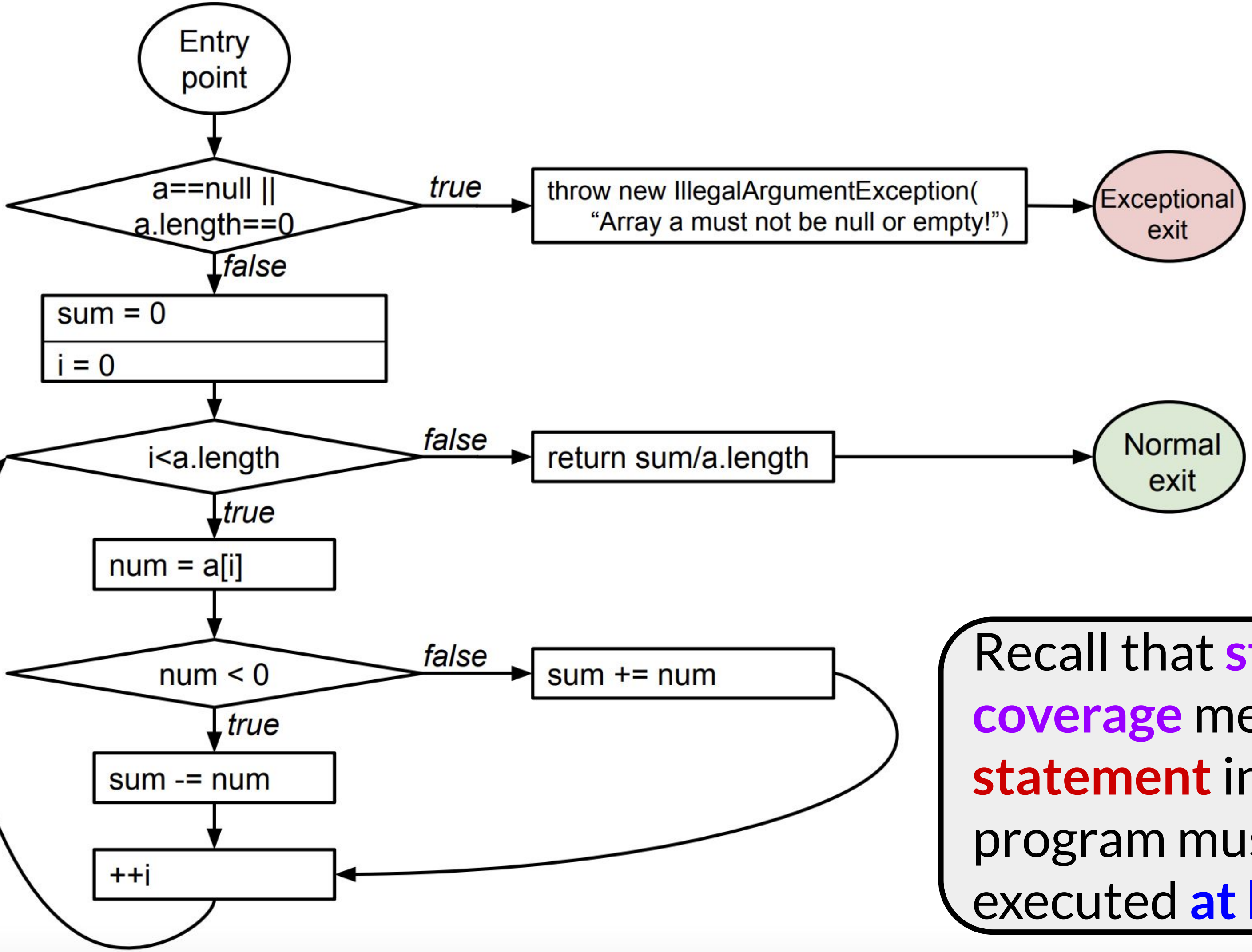
```
public double avgAbs(double... numbers) {
  // We expect the array to be non-null and non-empty
  if (numbers == null || numbers.length == 0) {
     throw new IllegalArgumentException("numbers is null or empty!");
  }
  double sum = 0;
  for (int i = 0; i < numbers.length; ++i) {
    double d = numbers[i];
    if (d < 0) { sum -= d; }
    else { sum += d; }
 }
 return sum / numbers.length;
}
```

Example: computing statement coverage

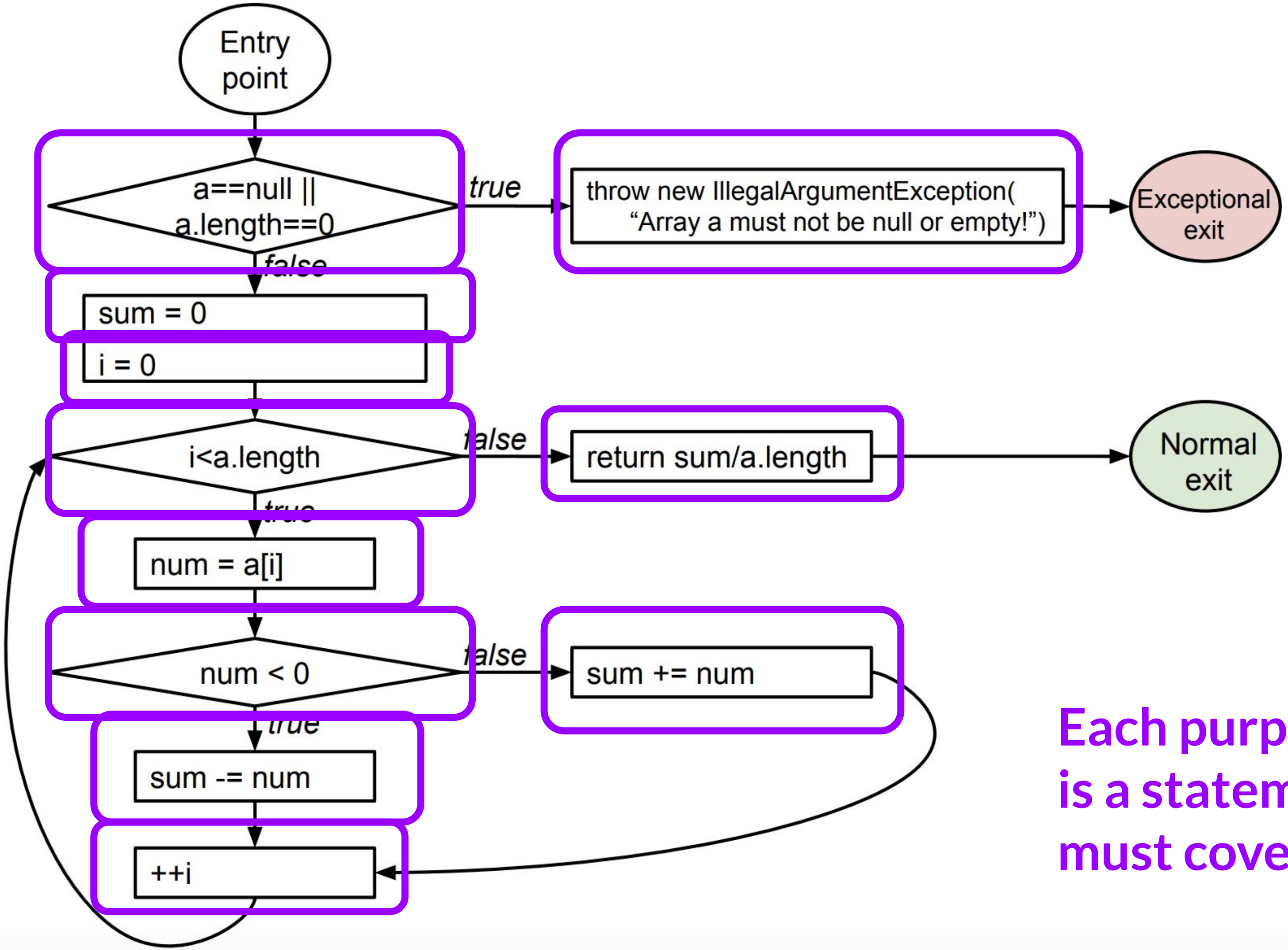
```java
public double avgAbs(double... numbers) {
  // We expect the array to be non-null and non-empty
  if (numbers == null || numbers.length == 0) {
     throw new IllegalArgumentException("numbers is null or empty!");
  }
  double sum = 0;
  for (int i = 0; i < numbers.length; ++i) {
    double d = numbers[i];
    if (d < 0) { sum -= d; }
    else { sum += d; }
 }
 return sum / numbers.length;
}
```

With a partner, draw the control flow graph for this method. (3 min)


Entry point
a==null || a.length==0
true
throw new IllegalArgumentException(“Array a must not be null or empty!”)
Exceptional exit
false
sum = 0
i = 0
i<a.length
false
return sum/a.length
Normal exit
true
num = a[i]
num < 0
false
sum += num
true
sum -= num
++i



Recall that **statement coverage** means "**every statement** in the program must be executed **at least once**."



Each purple box is a statement we must cover!

The CFG is useful for computing coverage

- Statement coverage is equivalent to ***node coverage*** in the CFG

The CFG is useful for computing coverage

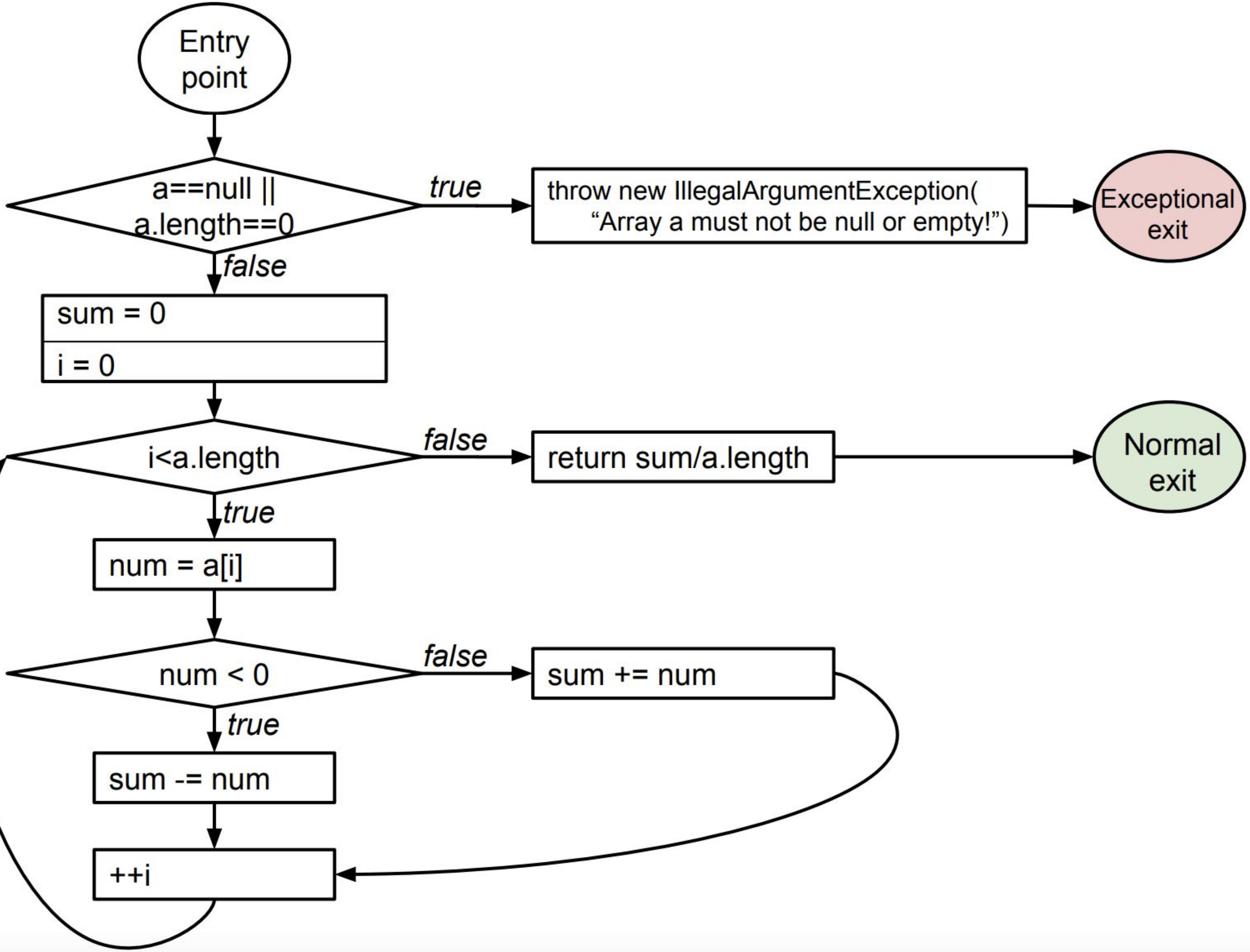
- Statement coverage is equivalent to ***node coverage*** in the CFG
  - that is, if we have a test that causes every CFG node to be executed, we are guaranteed to have 100% statement coverage

The CFG is useful for computing coverage

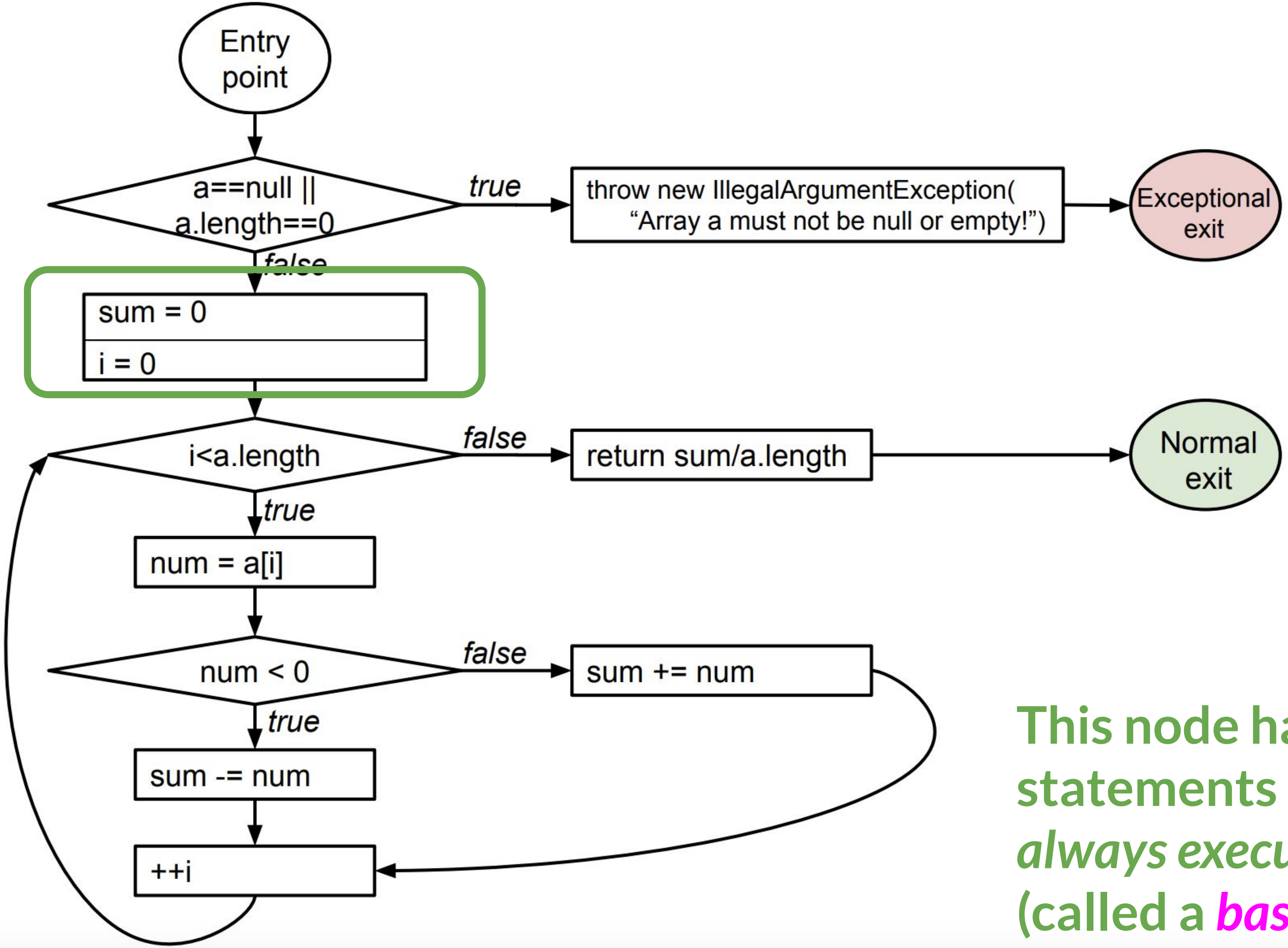
- Statement coverage is equivalent to ***node coverage*** in the CFG
  - that is, if we have a test that causes every CFG node to be executed, we are guaranteed to have 100% statement coverage
- In practice, this means that a tool for computing coverage will instrument each CFG node rather than each statement

The CFG is useful for computing coverage

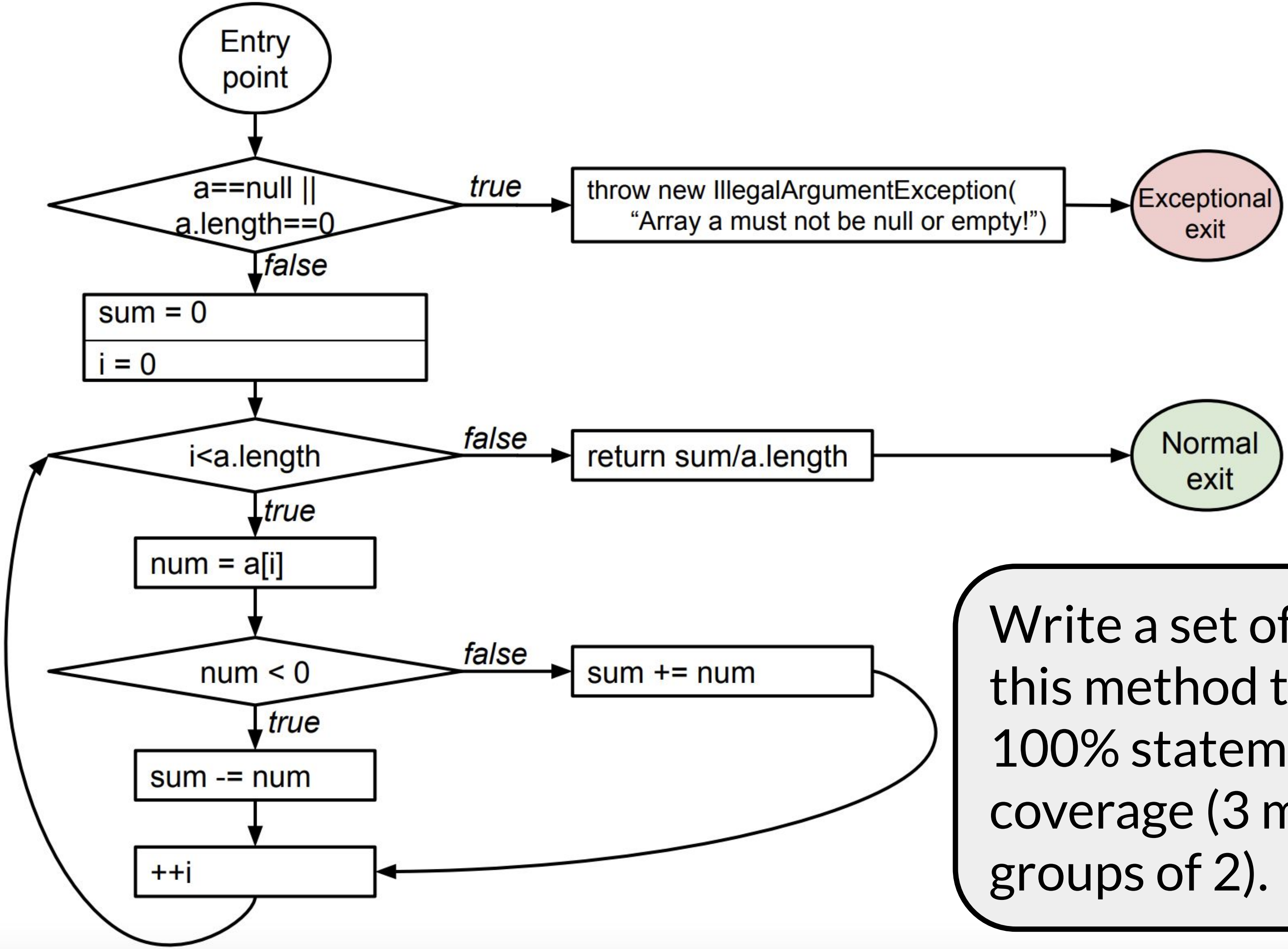
- Statement coverage is equivalent to ***node coverage*** in the CFG
  - that is, if we have a test that causes every CFG node to be executed, we are guaranteed to have 100% statement coverage
- In practice, this means that a tool for computing coverage will instrument each CFG node rather than each statement
  - where does this cause a difference in our example?



Entry point
a==null || a.length==0
true
throw new IllegalArgumentException("Array a must not be null or empty!")
Exceptional exit
false
sum = 0
i = 0
i<a.length
false
return sum/a.length
Normal exit
true
num = a[i]
num < 0
false
sum += num
true
sum -= num
++i



This node has two statements that are *always executed together* (called a *basic block*)



Write a set of tests for this method that achieves 100% statement (= node) coverage (3 minutes, groups of 2).

Coverage: limitations of statement coverage

- Executing every line doesn’t guarantee no bugs

Coverage: limitations of statement coverage

- Executing every line doesn't guarantee no bugs
- Not only that, but executing every line doesn't even guarantee that we cover all of the program's **behaviors**

Coverage: limitations of statement coverage

- Executing every line doesn't guarantee no bugs
- Not only that, but executing every line doesn't even guarantee that we cover all of the program's **behaviors**
  - many behaviors are dependent on data that causes particular **control flows**: that is, that cause different branches of conditionals to be executed

Coverage: limitations of statement coverage

- Executing every line doesn't guarantee no bugs
- Not only that, but executing every line doesn't even guarantee that we cover all of the program's **behaviors**
  - many behaviors are dependent on data that causes particular **control flows**: that is, that cause different branches of conditionals to be executed
- Informally, the problem of ensuring that we cover interesting data values may **reduce** to the problem of ensuring that we **cover all branches** of conditionals

Aside: reductions

Your CS education is incomplete until you have **reduced** one problem to another

Aside: reductions

Your CS education is incomplete until you have **reduced** one problem to another

- examples: reducing something to the halting problem to show that it is not computable; reducing something to satisfiability to show that it is NP-hard

Aside: reductions

Your CS education is incomplete until you have **reduced** one problem to another

- examples: reducing something to the halting problem to show that it is not computable; reducing something to satisfiability to show that it is NP-hard
- should be covered in a theory of computation class (likely near the end of the semester)

Aside: reductions

Your CS education is incomplete until you have **reduced** one problem to another

- examples: reducing something to the halting problem to show that it is not computable; redu[...] show that it is NP-hard
- should be covered in a theory [...] the end of the semester)

Reduction is a **powerful tool** for thinking about problems: it lets you solve difficult problems indirectly by re-using solutions for other, related problems.

Coverage: branch coverage

Definition: ***Branch coverage*** is a test suite quality metric that counts the total number of conditional branches exercised by that test suite (i.e., if true and if false are counted separately)

Coverage: branch coverage

Definition: ***Branch coverage*** is a test suite quality metric that counts the total number of conditional branches exercised by that test suite (i.e., if true and if false are counted separately)

Recall that branch coverage can subsume line coverage:

```
foo(a):
  if a > 5:
    print “x”
  print “y”
```

Coverage: branch coverage

Definition: *Branch coverage* is a test suite quality metric that counts the total number of conditional branches exercised by that test suite (i.e., if true and if false are counted separately)

Recall that branch coverage can subsume line coverage:

```
foo(a):
  if a > 5:
    print "x"
  print "y"
```

Test Suite { `foo(7)` } has 100% line coverage but 50% branch coverage.

Coverage: branch coverage

Definition: ***Branch coverage*** is a test suite quality metric that counts the total number of conditional branches exercised by that test suite (i.e., if true and if false are counted separately)

Recall that branch coverage can subsume line coverage:

```
foo(a):
  if a > 5:
    print "x"
  print "y"
```

Test Suite { `foo(7), foo(4)` } has 100% line coverage and 100% branch coverage.

Coverage: branch vs statement coverage

Coverage: branch vs statement coverage

- Branch coverage typically gives us **more confidence** than line coverage

Coverage: branch vs statement coverage

- Branch coverage typically gives us **more confidence** than line coverage
  - Typically, 100% branch coverage **implies** 100% line coverage

Coverage: branch vs statement coverage

- Branch coverage typically gives us **more confidence** than line coverage
    - Typically, 100% branch coverage **implies** 100% line coverage
- However, branch coverage is "**more expensive**" in the sense that it is harder for a test suite to have high branch coverage than to have high line coverage

Coverage: branch vs statement coverage

- Branch coverage typically gives us **more confidence** than line coverage
  - Typically, 100% branch coverage **implies** 100% line coverage
- However, branch coverage is “**more expensive**” in the sense that it is harder for a test suite to have high branch coverage than to have high line coverage
  - Note: quality isn't really “more expensive”, you were just fooling yourself before by thinking line coverage was OK. It is ***being correct*** that is expensive.

Beyond branch coverage

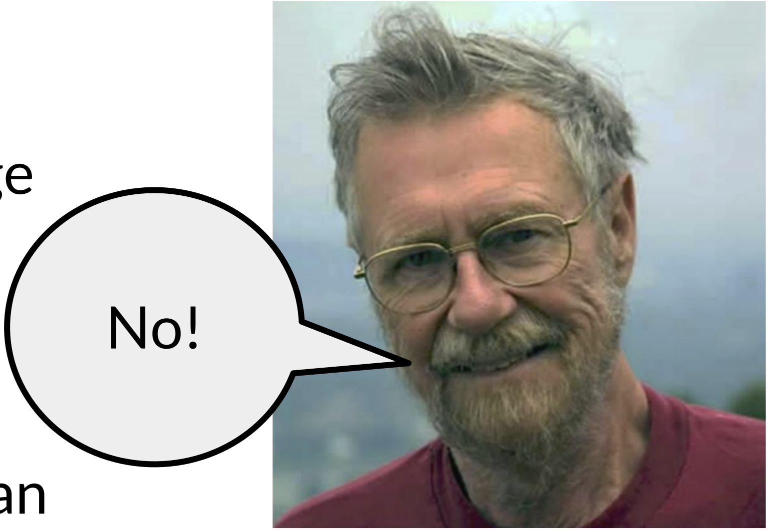
- We know 100% statement coverage **doesn't guarantee** no bugs

Beyond branch coverage

- We know 100% statement coverage **doesn't guarantee** no bugs
  - what about **100% branch coverage**? If we have 100% branch coverage, does that mean no bugs?

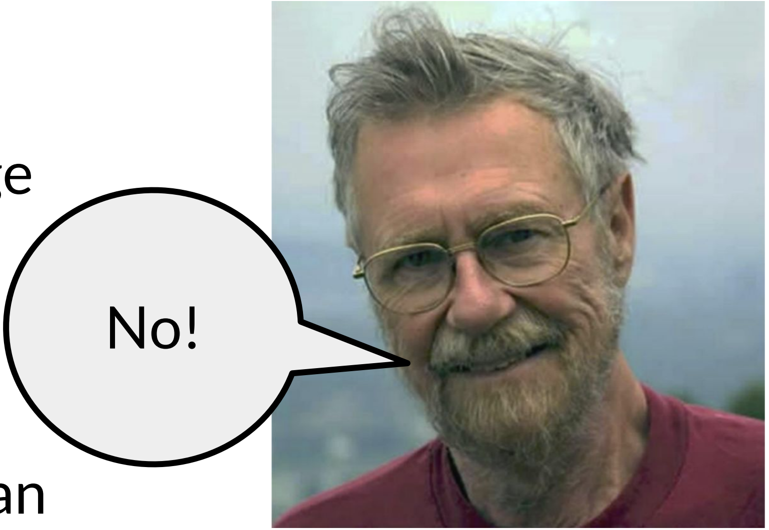
Beyond branch coverage

- We know 100% statement coverage **doesn't guarantee** no bugs
  - what about **100% branch coverage**? If we have 100% branch coverage, does that mean no bugs?



Beyond branch coverage

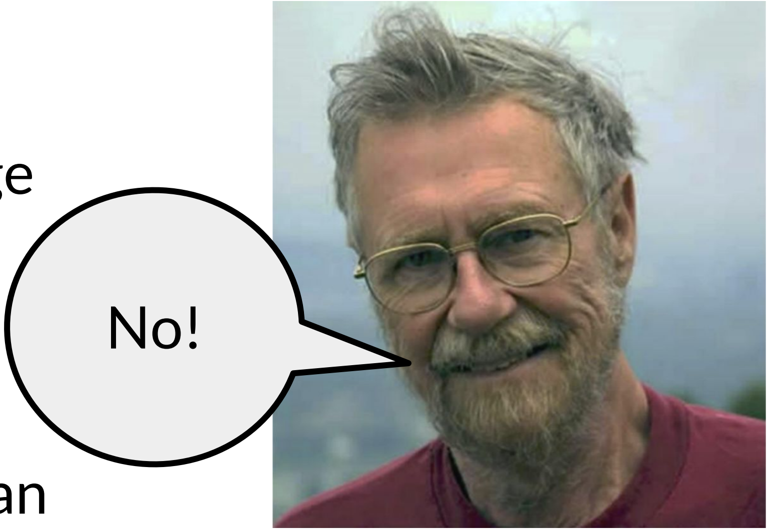
- We know 100% statement coverage **doesn't guarantee** no bugs
  - what about **100% branch coverage**? If we have 100% branch coverage, does that mean no bugs?



- Recall: “tests can only show the presence of bugs, not their absence”

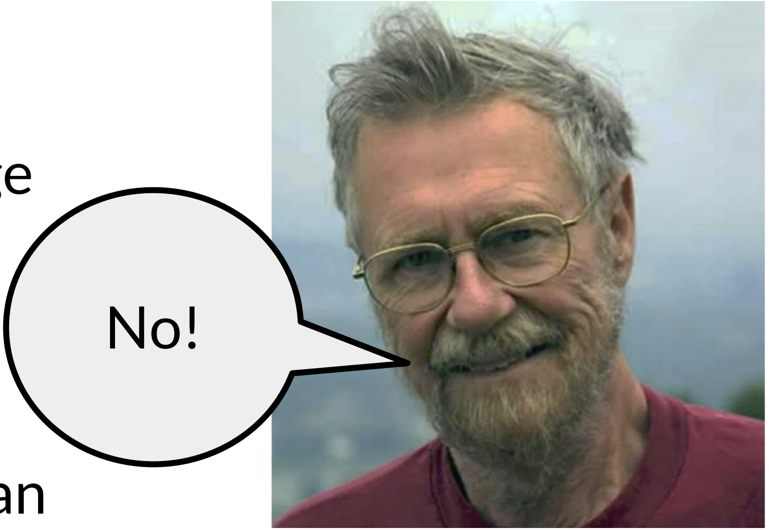
Beyond branch coverage

- We know 100% statement coverage **doesn't guarantee** no bugs
  - what about **100% branch coverage**? If we have 100% branch coverage, does that mean no bugs?
- Recall: “tests can only show the presence of bugs, not their absence”
- More coverage = more confidence, but **no guarantees**!



Beyond branch coverage

- We know 100% statement coverage **doesn't guarantee** no bugs
  - what about **100% branch coverage**? If we have 100% branch coverage, does that mean no bugs?



- Recall: "tests can only show the presence of bugs, not their absence"
- More coverage = more confidence, but **no guarantees**!
- Can we get **finer-grained** than branch coverage?

Conditions and decisions

- A ***condition*** is a boolean expression that cannot be decomposed into simpler boolean expressions (atomic).

Conditions and decisions

- A ***condition*** is a boolean expression that cannot be decomposed into simpler boolean expressions (atomic).
- A ***decision*** is a boolean expression that is composed of conditions, using 0 or more logical connectors.
    - note: a decision with 0 logical connectors is a condition

Conditions and decisions

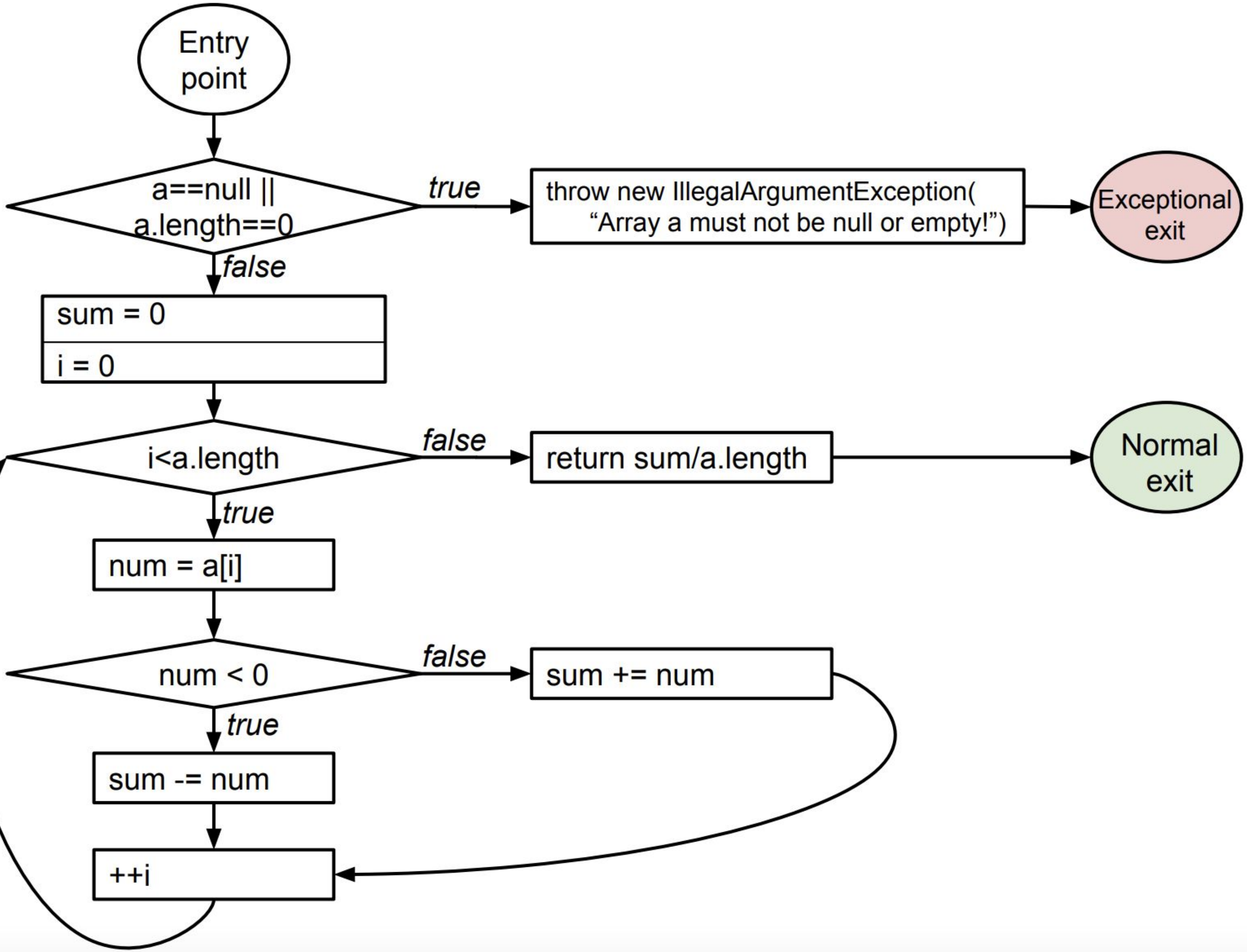
- A *condition* is a boolean expression that cannot be decomposed into simpler boolean expressions (atomic).
- A ***decision*** is a boolean expression that is composed of conditions, using 0 or more logical connectors.
  - note: a decision with 0 logical connectors is a condition
- Example: `if (a | b) {...}`

Conditions and decisions

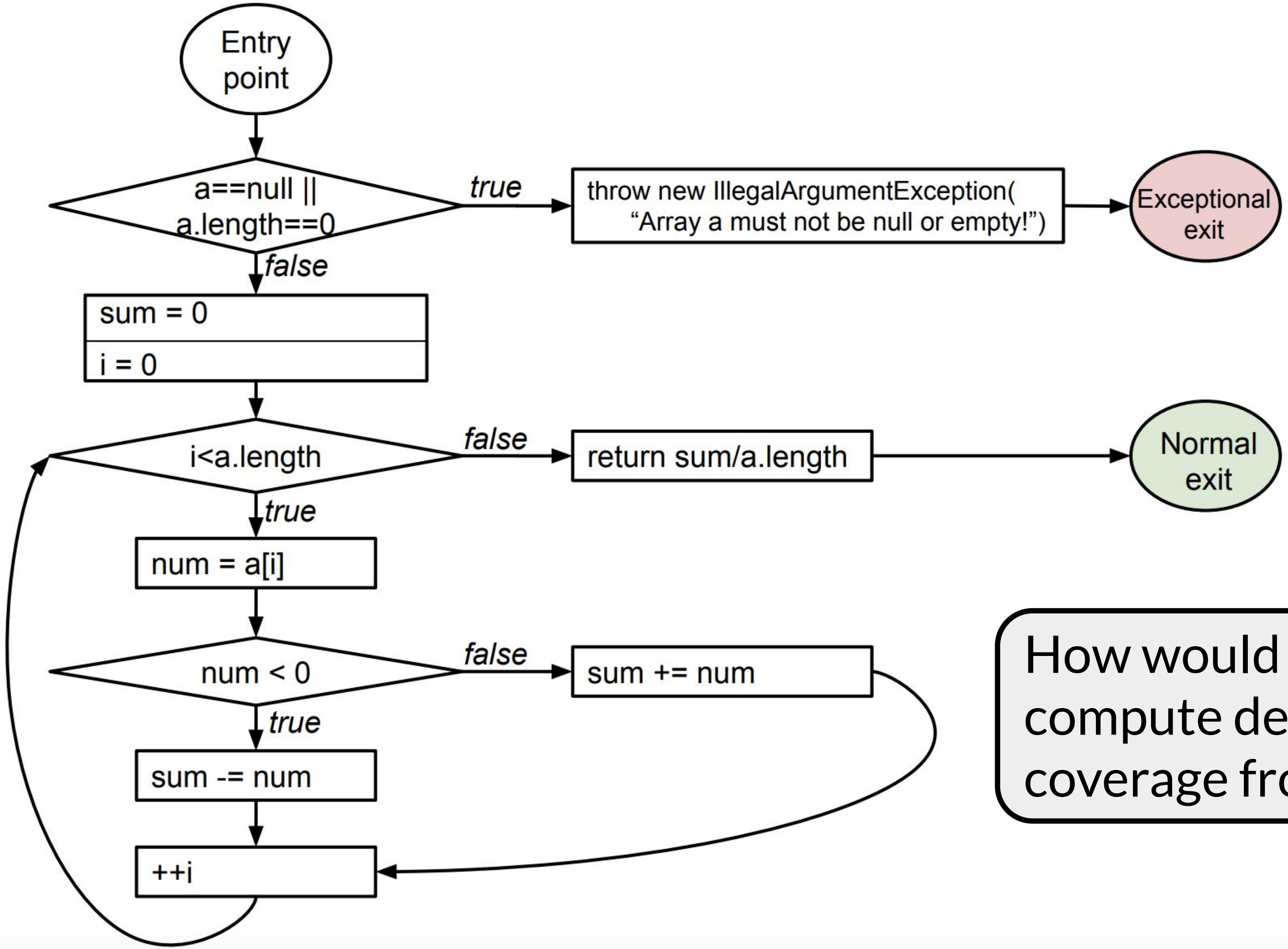
- A ***condition*** is a boolean expression that cannot be decomposed into simpler boolean expressions (atomic).
- A ***decision*** is a boolean expression that is composed of conditions, using 0 or more logical connectors.
    - note: a decision with 0 logical connectors is a condition
- Example: `if (a | b) {...}`
    - *a* and *b* are **conditions**
    - the boolean expression *a* | *b* is a **decision**

Decision coverage

- ***Decision coverage*** is the percentage of decisions that are true on at least one execution and false on at least one execution
  - i.e., 100% decision coverage = "**every decision** in the program must take on **all possible outcomes** (true/false) **at least once**"



Entry point
a==null || a.length==0
true
throw new IllegalArgumentException(“Array a must not be null or empty!”)
Exceptional exit
false
sum = 0
i = 0
i<a.length
false
return sum/a.length
Normal exit
true
num = a[i]
num < 0
false
sum += num
true
sum -= num
++i



How would you compute decision coverage from the CFG?

Decision coverage

- ***Decision coverage*** is the percentage of decisions that are true on at least one execution and false on at least one execution
  - i.e., 100% decision coverage = "**every decision** in the program must take on **all possible outcomes** (true/false) **at least once**"
- Decision coverage is equal to **edge coverage** in the CFG

Decision coverage

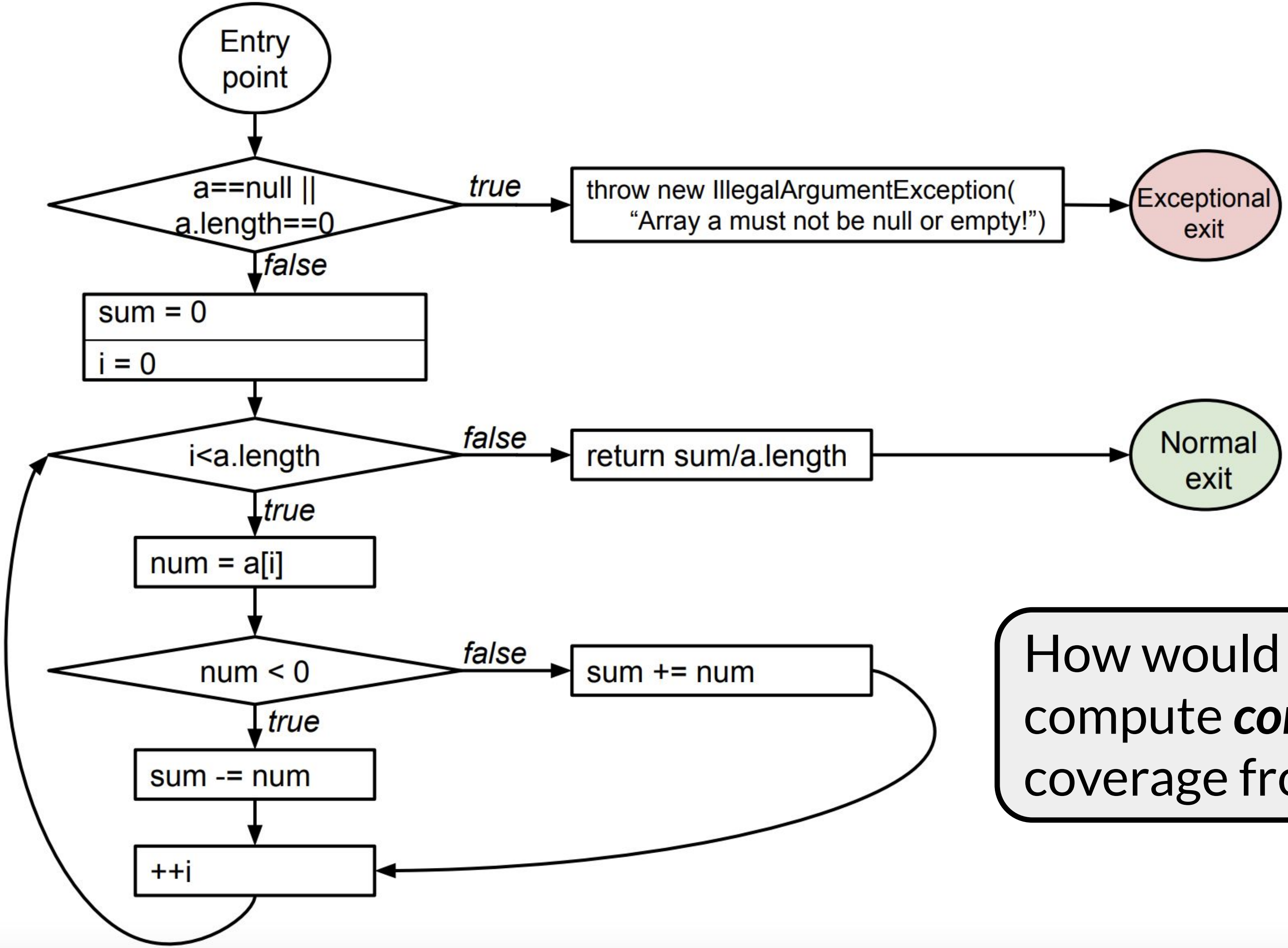
- ***Decision coverage*** is the percentage of decisions that are true on at least one execution and false on at least one execution
  - i.e., 100% decision coverage = "**every decision** in the program must take on **all possible outcomes** (true/false) **at least once**"
- Decision coverage is equal to **edge coverage** in the CFG
  - this is straightforward to instrument

Condition coverage

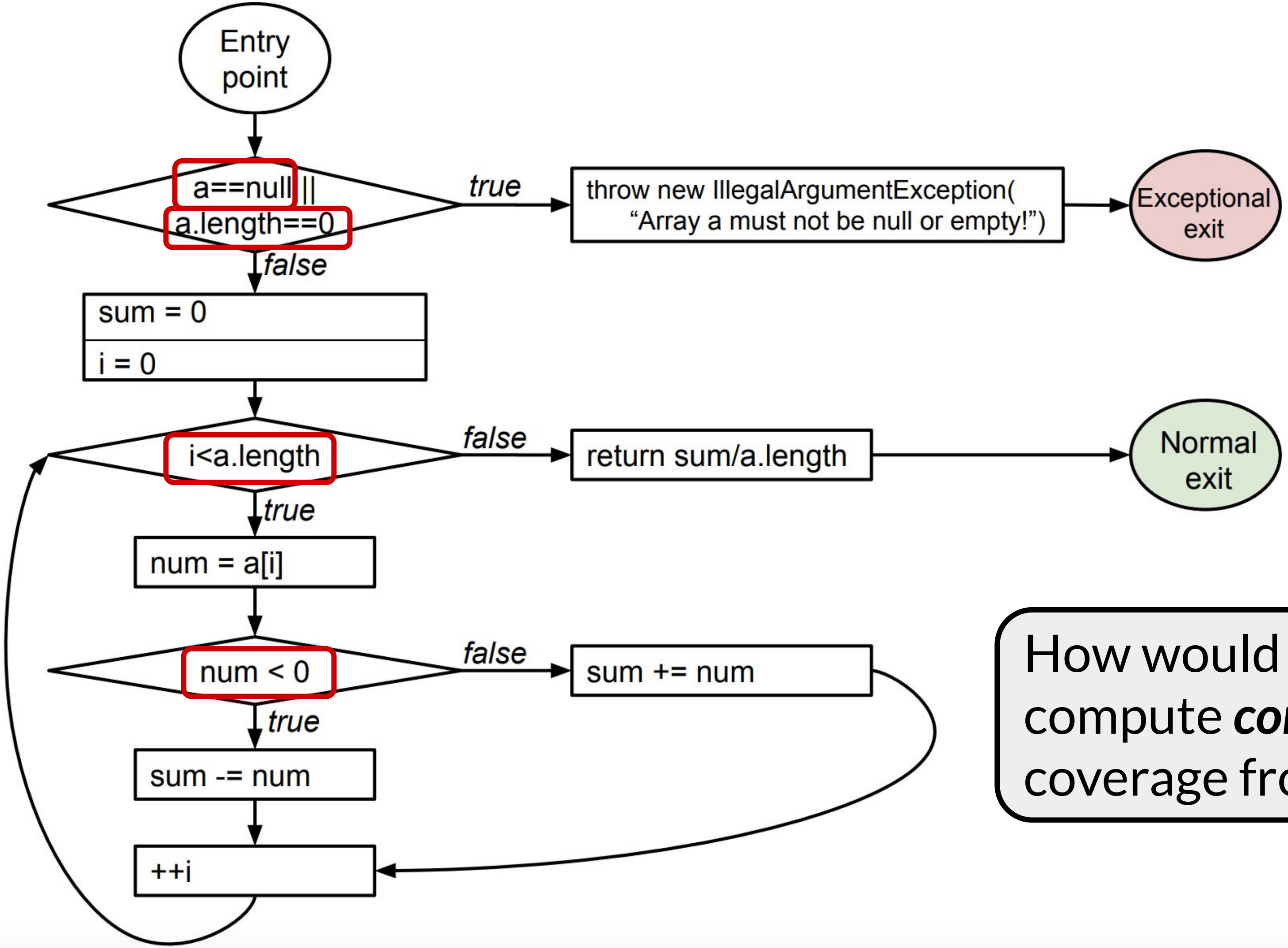
- ***Condition coverage*** is the percentage of ***conditions*** that are true on at least one execution and false on at least one execution

Condition coverage

- ***Condition coverage*** is the percentage of ***conditions*** that are true on at least one execution and false on at least one execution
  - i.e., 100% condition coverage = "***every condition*** in the program must take on **all possible outcomes** (true/false) **at least once**"



How would you compute ***condition*** coverage from the CFG?



How would you compute ***condition*** coverage from the CFG?

Subsumption

- We said earlier that branch coverage **subsumes** statement coverage
  - **Definition**: given two coverage criteria *A* and *B*, *A* **subsumes** *B* iff satisfying *A* implies satisfying *B*
- What about other kinds of coverage?
  - Does statement coverage subsume decision coverage?
  - Does decision coverage subsume statement coverage?
  - Does decision coverage subsume condition coverage?
  - Does condition coverage subsume decision coverage?

Subsumption

Take a three minutes and discuss these four with a partner.

- We said earlier that branch coverage **subsumes** statement coverage
  - **Definition**: given two coverage criteria *A* and *B*, *A* **subsumes** *B* iff satisfying *A* implies satisfying *B*
- What about other kinds of coverage?
  - Does statement coverage subsume decision coverage?
  - Does decision coverage subsume statement coverage?
  - Does decision coverage subsume condition coverage?
  - Does condition coverage subsume decision coverage?

Subsumption

Take a three minutes and discuss these four with a partner.

- We said earlier that branch coverage **subsumes** branch coverage
  - **Definition**: given two coverage criteria *A* and *B*, *A* **subsumes** *B* iff satisfying *A* implies satisfying *B*
- What about other kinds of coverage?
  - Does statement coverage subsume decision coverage? **NO**
  - Does decision coverage subsume statement coverage?
  - Does decision coverage subsume condition coverage?
  - Does condition coverage subsume decision coverage?

Subsumption

Take a three minutes and discuss these four with a partner.

- We said earlier that branch coverage **subsumes** branch coverage
  - **Definition**: given two coverage criteria *A* and *B*, *A* **subsumes** *B* iff satisfying *A* implies satisfying *B*
- What about other kinds of coverage?
  - Does statement coverage subsume decision coverage? **NO**
  - Does decision coverage subsume statement coverage? **YES**
  - Does decision coverage subsume condition coverage?
  - Does condition coverage subsume decision coverage?

Subsumption

Take a three minutes and discuss these four with a partner.

- We said earlier that branch coverage **subsumes** branch coverage
  - **Definition**: given two coverage criteria *A* and *B*, *A* **subsumes** *B* iff satisfying *A* implies satisfying *B*
- What about other kinds of coverage?
  - Does statement coverage subsume decision coverage? **NO**
  - Does decision coverage subsume statement coverage? **YES**
  - Does decision coverage subsume condition coverage? **NO**
  - Does condition coverage subsume decision coverage?

Subsumption

Take a three minutes and discuss these four with a partner.

- We said earlier that branch coverage **subsumes** branch coverage
  - **Definition**: given two coverage criteria *A* and *B*, *A* **subsumes** *B* iff satisfying *A* implies satisfying *B*
- What about other kinds of coverage?
  - Does statement coverage subsume decision coverage? NO
  - Does decision coverage subsume statement coverage? YES
  - Does decision coverage subsume condition coverage? NO
  - Does condition coverage subsume decision coverage? NO

Subsumption: decisions vs conditions

- There are 4 possible tests for the decision $a \mid b$:
  - $a = 0, b = 0$ $\quad\quad a \mid b = 0$
  - $a = 0, b = 1$ $\quad\quad a \mid b = 1$
  - $a = 1, b = 0$ $\quad\quad a \mid b = 1$
  - $a = 1, b = 1$ $\quad\quad a \mid b = 1$

Subsumption: decisions vs conditions

- There are 4 possible tests for the decision $a \mid b$:
  - $a = 0, b = 0$ $\quad a \mid b = 0$
  - $a = 0, b = 1$ $\quad a \mid b = 1$
  - $a = 1, b = 0$ $\quad a \mid b = 1$
  - $a = 1, b = 1$ $\quad a \mid b = 1$

These two tests satisfy **condition** but **not decision** coverage.

Subsumption: decisions vs conditions

- There are 4 possible tests for the decision $a \mid b$:
  - $a = 0, b = 0$ $\quad\quad$ $a \mid b = 0$
  - $a = 0, b = 1$ $\quad\quad$ $a \mid b = 1$
  - $a = 1, b = 0$ $\quad\quad$ $a \mid b = 1$
  - $a = 1, b = 1$ $\quad\quad$ $a \mid b = 1$

These two tests satisfy **decision** but **not condition** coverage.

Subsumption: decisions vs conditions

- There are 4 possible tests for the decision $a \mid b$:
  - $a = 0, b = 0$ — $a \mid b = 0$
  - $a = 0, b = 1$ — $a \mid b = 1$
  - $a = 1, b = 0$ — $a \mid b = 1$
  - $a = 1, b = 1$ — $a \mid b = 1$
- Implication: **neither** decision coverage **nor** condition coverage subsumes the other!

MC/DC coverage

- ***modified condition and decision coverage*** (***MC/DC coverage***) is a coverage criterion that requires:

MC/DC coverage

- ***modified condition and decision coverage*** (***MC/DC coverage***) is a coverage criterion that requires:
  - every decision in the program must take on all possible outcomes (true/false) at least once (**100% decision coverage**)

MC/DC coverage

- ***modified condition and decision coverage*** (***MC/DC coverage***) is a coverage criterion that requires:
  - every decision in the program must take on all possible outcomes (true/false) at least once (**100% decision coverage**)
  - every condition in the program must take on all possible outcomes (true/false) at least once (**100% condition coverage**)

MC/DC coverage

- ***modified condition and decision coverage*** (***MC/DC coverage***) is a coverage criterion that requires:
    - every decision in the program must take on all possible outcomes (true/false) at least once (**100% decision coverage**)
    - every condition in the program must take on all possible outcomes (true/false) at least once (**100% condition coverage**)
    - each condition in a decision has been shown to **independently** affect that decision's **outcome**.
        - A condition is shown to independently affect a decision's outcome by varying just that condition while holding all other conditions fixed

MC/DC coverage

- ***modified condition and decision coverage*** (***MC/DC coverage***) is a coverage criterion that requires:
  - every decision in the program mu
    outcomes (true/false) at least on
  - every condition in the program m
    outcomes (true/false) at least once (**100% condition coverage**)
  - each condition in a decision has been shown to **independently** affect that decision's **outcome**.
    - A condition is shown to independently affect a decision's outcome by varying just that condition while holding all other conditions fixed

100% MC/DC coverage is **required** for safety critical systems (DO-178B/C)!

MC/DC coverage example

```
if (a | b)
```

a	b	outcome
0	0	0
0	1	1
1	0	1
1	1	1

MC/DC =

- 100% decision coverage
- 100% condition coverage
- each condition shown to independently affect outcome

Which tests (combinations of a and b) satisfy MC/DC?

MC/DC coverage example

```
if (a | b)
```

a	b	outcome
0	0	0
0	1	1
1	0	1
1	1	1

MC/DC =

- 100% decision coverage
- 100% condition coverage
- each condition shown to independently affect outcome

Which tests (combinations of a and b) satisfy MC/DC?

MC/DC is still cheaper than testing all possible combinations!

MC/DC: another example

```
if (a || b)
```

a	b	outcome
0	0	0
0	1	1
1	-	1
1	-	1

MC/DC =
- 100% decision coverage
- 100% condition coverage
- each condition shown to independently affect outcome

Why is this example different? Short-circuiting operators may not evaluate all conditions.

MC/DC: a third example

`if (!a) … if (a || b)`

a	b	outcome
0	0	0
0	1	1
1	0	1
1	1	1

MC/DC =

- 100% decision coverage
- 100% condition coverage
- each condition shown to independently affect outcome

What about this example?

MC/DC: a third example

`if (!a) … if (a || b)`

a	b	outcome
0	0	0
0	1	1
X	X	X
X	X	X

MC/DC =
- 100% decision coverage
- 100% condition coverage
- each condition shown to independently affect outcome

What about this example?

Not all combinations of conditions may be possible!

MC/DC coverage: complex expressions

- With a partner, take a few minutes to provide an MC/DC-adequate test suite for:
  - `a | b | c`
  - `a & b & c`

MC/DC =
- 100% decision coverage
- 100% condition coverage
- each condition shown to independently affect outcome

Coverage: other kinds of coverage

- You can define coverage over any kind of program structure
    - e.g., what do you think **function coverage** is?

Coverage: other kinds of coverage

- You can define coverage over any kind of program structure
  - e.g., what do you think **function coverage** is?
- You can also define coverage over non-programmatic things
  - e.g., **requirements coverage** or **user-story coverage** are sometimes used in industry

Coverage: summary

Coverage: summary

- Coverage is **easy to compute**

Coverage: summary

- Coverage is **easy to compute**
- Coverage has an **intuitive interpretation**

Coverage: summary

- Coverage is **easy to compute**
- Coverage has an **intuitive interpretation**
- Coverage is **common in industry** (e.g., think about today's reading about Google)

Coverage: summary

- Coverage is **easy to compute**
- Coverage has an **intuitive interpretation**
- Coverage is **common in industry** (e.g., think about today's reading about Google)
- But coverage on its own is **not sufficient** to guarantee correctness

Coverage: summary

- Coverage is **easy to compute**
- Coverage has an **intuitive interpretation**
- Coverage is **common in industry** (e.g., think about today's reading about Google)
- But coverage on its own is **not sufficient** to guarantee correctness
  - just because you executed a line does not mean that that line did the right thing! (oracles!)

Three ways to think about test suite quality

Consider three ways to think about test suite quality:

- test suite quality through the lens of logic
  - intuition: if we don't test it, we can't find bugs
  - leads to ***coverage*** (main subject of today's lecture)
- test suite quality through the lens of **statistics**
  - intuition: test what happens to real users
- test suite quality through the lens of adversity
  - intuition: inject bugs and see if the test suite catches them
  - leads to ***mutation testing***, which we'll cover later this semester

Three ways to think about test suite quality

Consider three ways to think about test suite quality:

- test suite quality through the lens of
  - intuition: if we don’t test it, we ca
  - leads to *coverage* (main subject of
- test suite quality through the lens of **statistics**
  - intuition: test what happens to real users
- test suite quality through the lens of **adversity**
  - intuition: inject bugs and see if the test suite catches them
  - leads to ***mutation testing***, which we’ll cover later this semester

Rest of today’s lecture:
a brief discussion of the
lens of statistics

The Lens of Statistics: intuition

The Lens of Statistics: intuition

- The bugs experienced **by users** are the ones that matter.

The Lens of Statistics: intuition

- The bugs experienced **by users** are the ones that matter.
  - Dually, bugs never experienced by users do not matter.

The Lens of Statistics: intuition

- The bugs experienced **by users** are the ones that matter.
  - Dually, bugs never experienced by users do not matter.
- If user-experienced bugs are the ones that matter, testing should be devoted to **sampling** those inputs that users will provide

The Lens of Statistics: intuition

- The bugs experienced **by users** are the ones that matter.
  - Dually, bugs never experienced by users do not matter.
- If user-experienced bugs are the ones that matter, testing should be devoted to **sampling** those inputs that users will provide
- Two views:

The Lens of Statistics: intuition

- The bugs experienced **by users** are the ones that matter.
  - Dually, bugs never experienced by users do not matter.
- If user-experienced bugs are the ones that matter, testing should be devoted to **sampling** those inputs that users will provide
- Two views:
  - Sample what users do **most commonly**

The Lens of Statistics: intuition

- The bugs experienced **by users** are the ones that matter.
  - Dually, bugs never experienced by users do not matter.
- If user-experienced bugs are the ones that matter, testing should be devoted to **sampling** those inputs that users will provide
- Two views:
  - Sample what users do **most commonly**
  - Sample what causes the **most harm** if users do it

The Lens of Statistics: intuition

- The bugs experienced **by users** are the ones that matter.
  - Dually, bugs never experienced by users do not matter.
- If user-experienced bugs are the ones that matter, testing should be devoted to **sampling** those inputs that users will provide
- Two views:
  - Sample what users do **most commonly**
  - Sample what causes the **most harm** if users do it
- Compare:
  - Risk = (Probability of Event) * (Damage if Event Occurs)

Example: limited input domain

- Suppose you are writing a point-of-sale cashier application that makes change for a dollar. Given any price between 1 and 100 cents, you must indicate the coins to give out as change.
  - e.g., 23 → return 3 quarters and 2 pennies

Example: limited input domain

- Suppose you are writing a point-of-sale cashier application that makes change for a dollar. Given any price between 1 and 100 cents, you must indicate the coins to give out as change.
  - e.g., 23 → return 3 quarters and 2 pennies
- In this scenario, you can **exhaustively test** all 100 inputs that will occur to real users in the real world
  - In some sense, it does not matter if that is 100% statement or code coverage (e.g., dead code): your testing is still exhaustive of the inputs that will matter in the real world

Limitations on users in the real world

- Usually, in the real world, your input domain isn't so limited
- But, you might still be able to say:

Limitations on users in the real world

- Usually, in the real world, your input domain isn't so limited
- But, you might still be able to say:
  - Suppose users will only ever cause lines 1, 2 and 3 of your program to be executed

Limitations on users in the real world

- Usually, in the real world, your input domain isn't so limited
- But, you might still be able to say:
  - Suppose users will only ever cause lines 1, 2 and 3 of your program to be executed
  - If you can be sure of this, then there is no need to test line 4

Limitations on users in the real world

- Usually, in the real world, your input domain isn't so limited
- But, you might still be able to say:
    - Suppose users will only ever cause lines 1, 2 and 3 of your program to be executed
    - If you can be sure of this, then there is no need to test line 4
        - Aside: why do you have line 4?

Limitations on users in the real world

- Usually, in the real world, your input domain isn't so limited
- But, you might still be able to say:
    - Suppose users will only ever cause lines 1, 2 and 3 of your program to be executed
    - If you can be sure of this, then there is no need to test line 4
        - Aside: why do you have line 4?
    - Even if line 4 has a bug, users will **never** encounter it

Limitations on users in the real world

- Usually, in the real world, your input domain isn't so limited
- But, you might still be able to say:
    - Suppose users will only ever cause lines 1, 2 and 3 of your program to be executed
    - If you can be sure of this, then there is no need to test line 4
        - Aside: why do you have line 4?
    - Even if line 4 has a bug, users will **never** encounter it
- Note "will": this either requires a **prediction of the future** or a **finite input domain**

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 poll republicans.

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 poll republicans.
  - 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: ***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

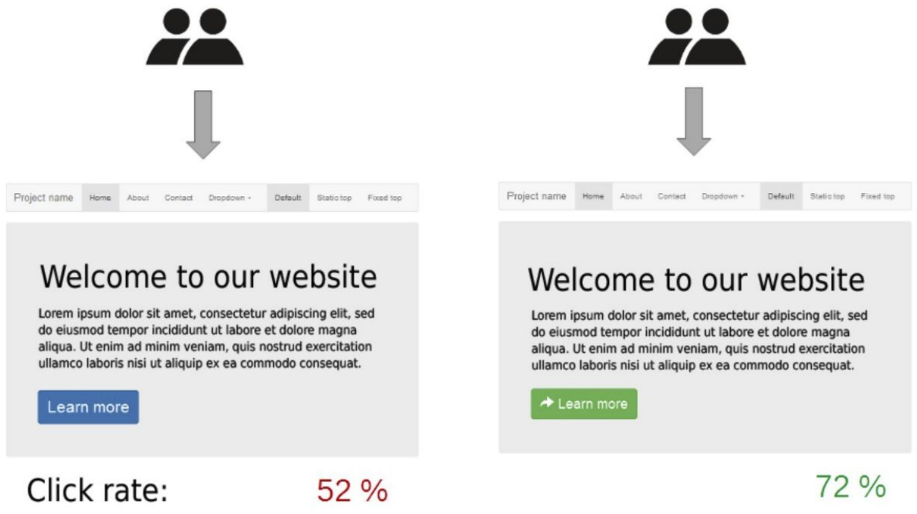
But beware of sampling errors! Who signs up to be a beta tester?
Hint: **not the average user**!

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

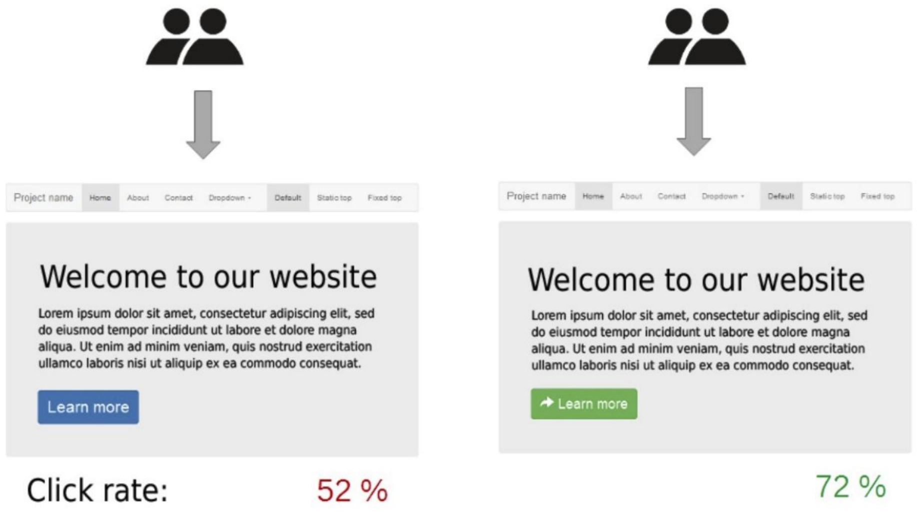
Definition: ***A/B testing*** involves two variants of your software, A and B, which differ only in one feature. Different users are shown different variants and responses are recorded.



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"

Today’s in-class

- Achieve higher coverage on libpng
  - inputs are image files
- This assignment is supposed to be harder than HW2
  - libpng is ~85k LoC, so I don’t expect you to read it all
  - this is indicative of real-world engineering: you usually don’t have time to read all the relevant code
  - getting started can be tricky, so use us for the rest of class to get around any difficulties collecting coverage locally
  - good luck!