Coverage-based testing

UW CSE P 504

Today

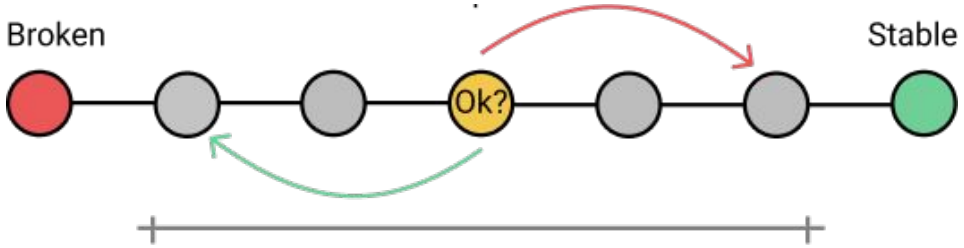
- Recap: Git bisect exercise
- Software testing 101
- Test adequacy: structural code coverage
  - Statement coverage
  - Decision coverage
  - Condition coverage
  - Modified condition and decision coverage (MCDC)
- Discussion of papers
- In-class activity: code coverage

Recap: git bisect

- **Git bisect run-time complexity is …?**

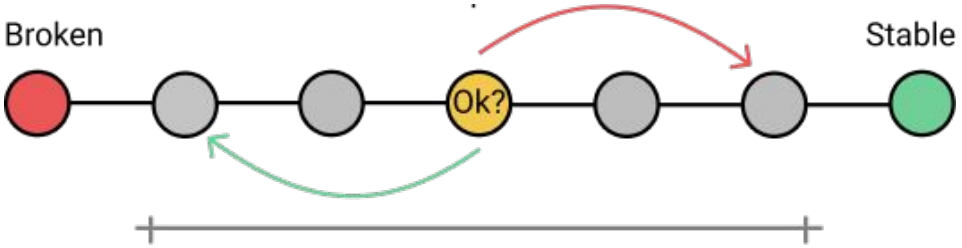
Recap: git bisect

- **Git bisect run-time complexity is always O(log(n))**

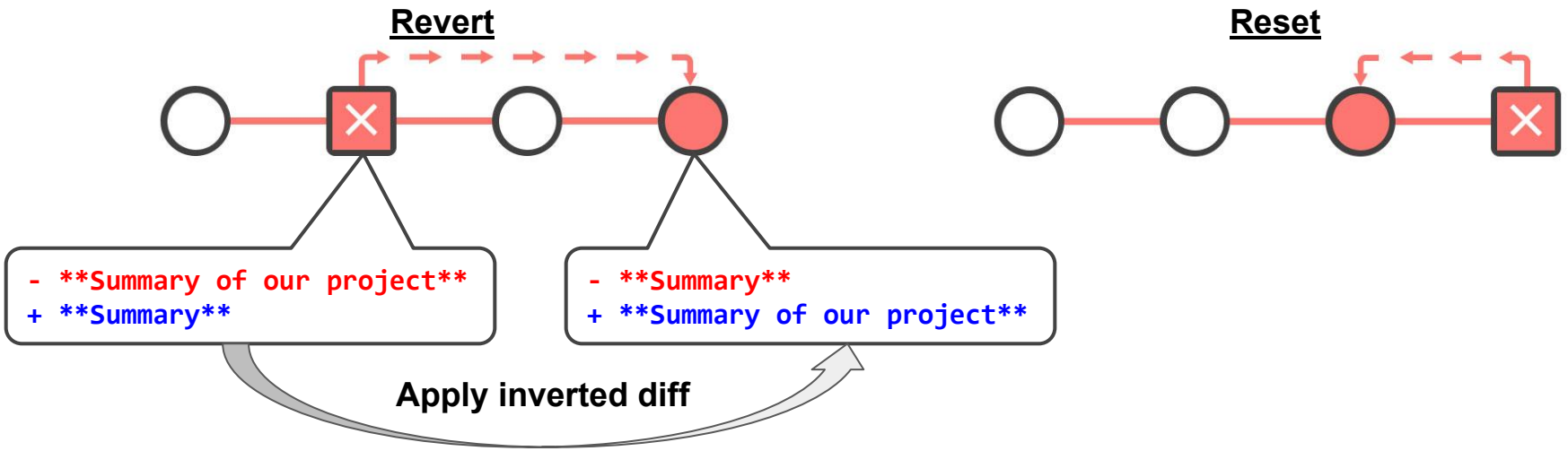


Recap: git bisect

- **Git bisect run-time complexity is always O(log(n))**

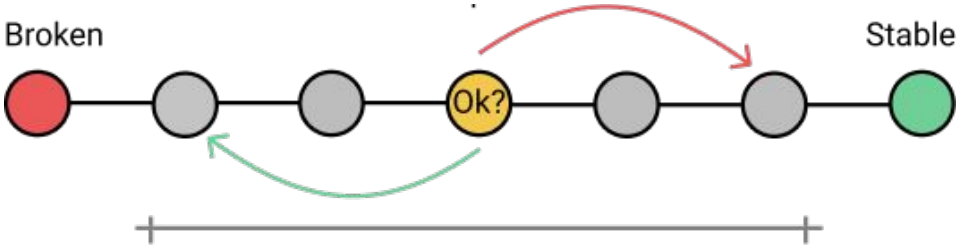


- **Git revert = add inverted commit; git reset = remove a commit**

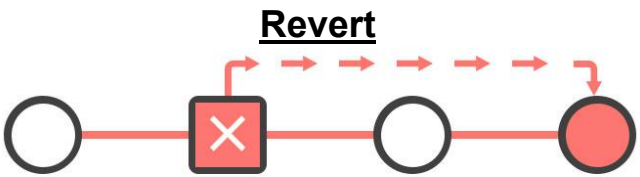


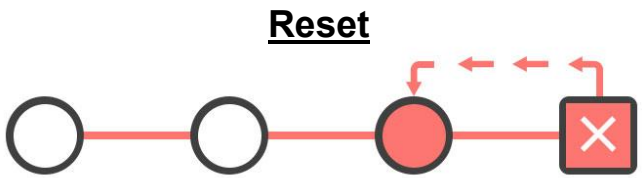
Recap: git bisect

- **Git bisect run-time complexity is always O(log(n))**

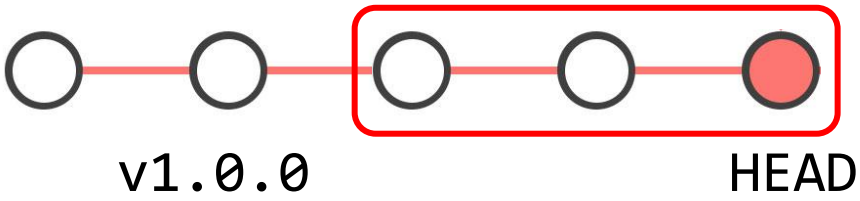


- **Git revert = add inverted commit; git reset = remove a commit**





- `git rev-list v1.0.0..HEAD (or HEAD ^v1.0.0)`



Recap: git bisect

Questions

- How could the developers improve the build or testing infrastructure to notice test failures in the future?
- Which git command can you use to undo a defect-inducing commit? Briefly explain what problem may generally occur when undoing a commit and what best practices mitigate this problem.
- Can you undo the defect-inducing commit using the proposed git command?

Recap: git bisect

Questions

- How could the developers improve the build or testing infrastructure to notice test failures in the future?
- Which git command can you use to undo a defect-inducing commit? Briefly explain what problem may generally occur when undoing a commit and what best practices mitigate this problem.
- Can you undo the defect-inducing commit using the proposed git command?

Meta-level discussion

- Is Git bisect a realistic choice for the JavaParser example?
- I don’t use Java, so why should I care?
- Forum participation is great!

Software testing 101

Testing vs. debugging

```
1  double avg(double[] nums) {
2    int n = nums.length;
3    double sum = 0;
4
5    int i = 0;
6    while (i<n) {
7      sum = sum + nums[i];
8      i = i + 1;
9    }
10
11   double avg = sum * n;
12   return avg;
13 }
```

Testing vs. debugging

```
1  double avg(double[] nums) {
2    int n = nums.length;
3    double sum = 0;
4
5    int i = 0;
6    while (i<n) {
7      sum = sum + nums[i];
8      i = i + 1;
9    }
10
11   double avg = sum * n;
12   return avg;
13 }
```

Testing: is there a failure?

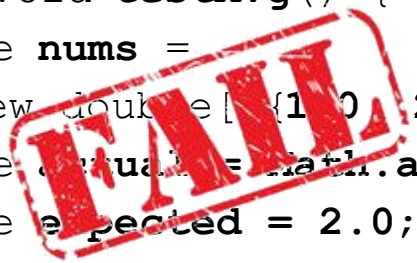
```
@Test
public void testAvg() {
  double nums =
      new double[]{1.0, 2.0, 3.0});
  double actual = Math.avg(nums);
  double expected = 2.0;
  assertEquals(expected,actual,EPS);
}
```

Testing vs. debugging

```
1  double avg(double[] nums) {
2    int n = nums.length;
3    double sum = 0;
4
5    int i = 0;
6    while (i<n) {
7      sum = sum + nums[i];
8      i = i + 1;
9    }
10
11   double avg = sum * n;
12   return avg;
13 }
```

Testing: is there a failure?

```
@Test
public void testAvg() {
  double nums =
      new double[]{1.0, 2.0, 3.0});
  double actual = Math.avg(nums);
  double expected = 2.0;
  assertEquals(expected,actual,EPS);
}
```



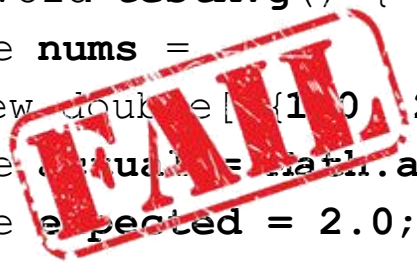

```
testAvg failed: 2.0 != 18.0
```

Testing vs. debugging

```
1  double avg(double[] nums) {
2    int n = nums.length;
3    double sum = 0;
4
5    int i = 0;
6    while (i<n) {
7      sum = sum + nums[i];
8      i = i + 1;
9    }
10
11   double avg = sum * n;
12   return avg;
13 }
```

Testing: is there a failure?

```
@Test
public void testAvg() {
  double nums =
      new double[]{1.0, 2.0, 3.0});
  double actual = MathL.avg(nums);
  double expected = 2.0;
  assertEquals(expected,actual,EPS);
}
```



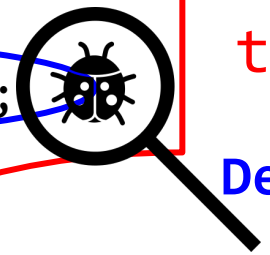

```
testAvg failed: 2.0 != 18.0
```

Debugging: where is the defect?

how to fix the defect?

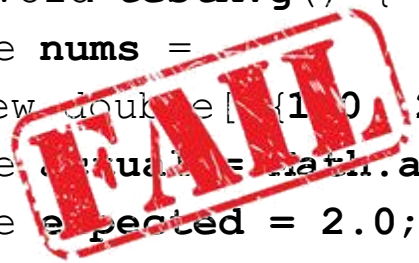
Testing vs. debugging

```
1  double avg(double[] nums) {
2    int n = nums.length;
3    double sum = 0;
4
5    int i = 0;
6    while (i<n) {
7      sum = sum + nums[i];
8      i = i + 1;
9    }
10
11   double avg = sum * n;
12   return avg;
13 }
```



Testing: is there a failure?

```
@Test
public void testAvg() {
  double nums =
      new double[]{1.0, 2.0, 3.0});
  double actual = MathLib.avg(nums);
  double expected = 2.0;
  assertEquals(expected,actual,EPS);
}
```




```
testAvg failed: 2.0 != 18.0
```

Debugging: where is the defect?

how to fix the defect?

Software testing is unsound

“Program testing can … show the presence of bugs but never … show their absence.”
(Edsger W. Dijkstra)

Software testing is unsound

"Program testing can … show the presence of bugs but never … show their absence."
(Edsger W. Dijkstra)

- A good test is one that fails because of a defect.

How do we come up with good tests?

Devising good tests

- Partition input into different behaviors
- One test for each behavior
- Use heuristics for partitioning (= for choosing tests)
  - Take inspiration from the theory of revealing subdomains

Two strategies: black box vs. clear box

Black box testing

- The system is a black box (can’t see inside).
- No knowledge about the internals of a system.
- Create tests solely based on the specification (e.g., input/output behavior).

Clear box testing

- Knowledge about the internals of a system.
- Create tests based on these internals (e.g., exercise a particular part or path of the system).

Clear box testing example

Some subdomains are not evident from the specification

```
boolean[] primeTable = new boolean[CACHE_SIZE];

boolean isPrime(int x) {
  if (x < CACHE_SIZE) {
    return primeTable[x];
  } else {
    for (int i = 2; i < x/2; i++) {
      if (x%i == 0)
        return false;
      }
      return true;
    }
  }
}
```

Subdomain boundary (execution difference)
at: *x* = CACHE_SIZE

Clear box testing tradeoffs

Advantages

- Provides an important source of boundaries
- Has an objective test quality metric: coverage

Disadvantages

- Tests may have same bugs as implementation
  - Buggy code tricks you into complacency as soon as you read it

Unit testing, integration testing, system testing

Unit testing

- Does each unit work as specified?

Integration testing

- Do the units work when put together?

System testing

- Does the system work as a whole?

Unit testing, integration testing, system testing

Unit testing

- Does each unit work as specified?

Integration testing

- Do the units work when put together?

System testing

- Does the system work as a whole?

Our focus: unit testing

Unit testing

- A **unit** is the **smallest testable part** of the software system (e.g., a method or a function).
- **Goal**: Verify that each software unit performs as specified.
- **Focus**:
  - Individual unit
    - Not the interactions between units
    - Not dependences of the unit
  - Usually input/output relationships

Software testing is unsound

“Program testing can … show the presence of bugs but never … show their absence.”
(Edsger W. Dijkstra)

- A good test is one that fails because of a defect.

When should we stop testing if no (new) test fails?

Test effectiveness

Ratio of **detected defects** is all that matters!

Problem

- The set of defects is unknowable.

Solution

- Use a proxy metric, for example code coverage.

Adequacy = score according to the proxy metric
Adequate = 100% score

Test adequacy: structural code coverage

Structural code coverage: motivating example

Average of 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("Array numbers must not be 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;
}
```

What tests should we write for this method?

Cobertura's line coverage report

Classes in this File	Line Coverage	Branch Coverage	Complexity
Avg	100% 10/10	100% 8/8	6

```
 1    package avg;
 2
 3 4  public class Avg {
 4
 5        /*
 6         * Compute the average of the absolute values of an array of doubles
 7         */
 8        public double avgAbs(double ... numbers) {
 9            // We expect the array to be non-null and non-empty
10 4        if (numbers == null || numbers.length == 0) {
11 2            throw new IllegalArgumentException("Array numbers must not be null or empty!");
12          }
13
14 2        double sum = 0;
15 8        for (int i=0; i<numbers.length; ++i) {
16 6            double d = numbers[i];
17 6            if (d < 0) {
18 2                sum -= d;
19              } else {
20 4                sum += d;
21              }
22          }
23 2        return sum/numbers.length;
24       }
25   }
```

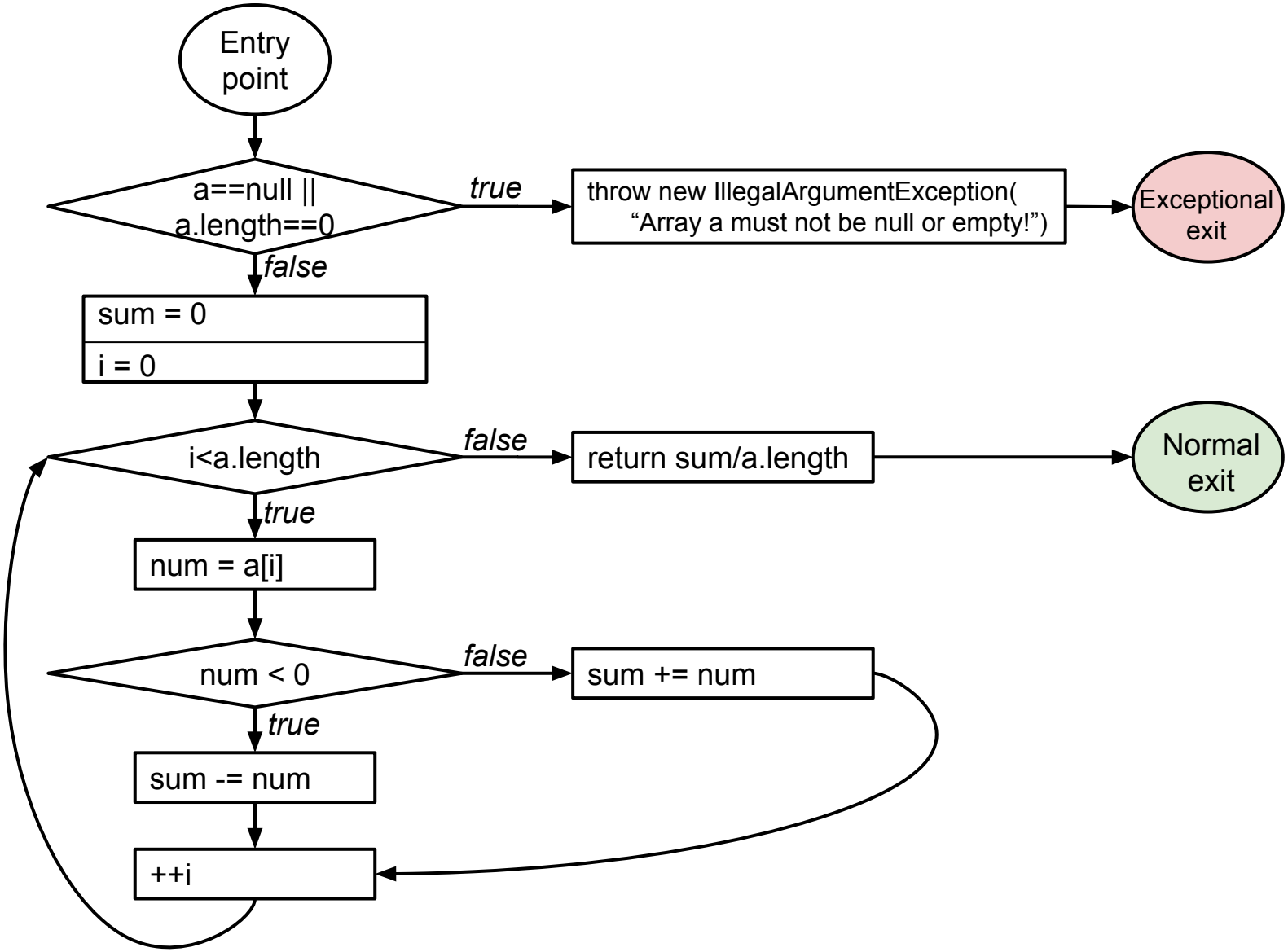
Structural code coverage: visualization

Average of 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("Array numbers must not be 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;
}
```

What's the control flow graph (CFG) for this method?

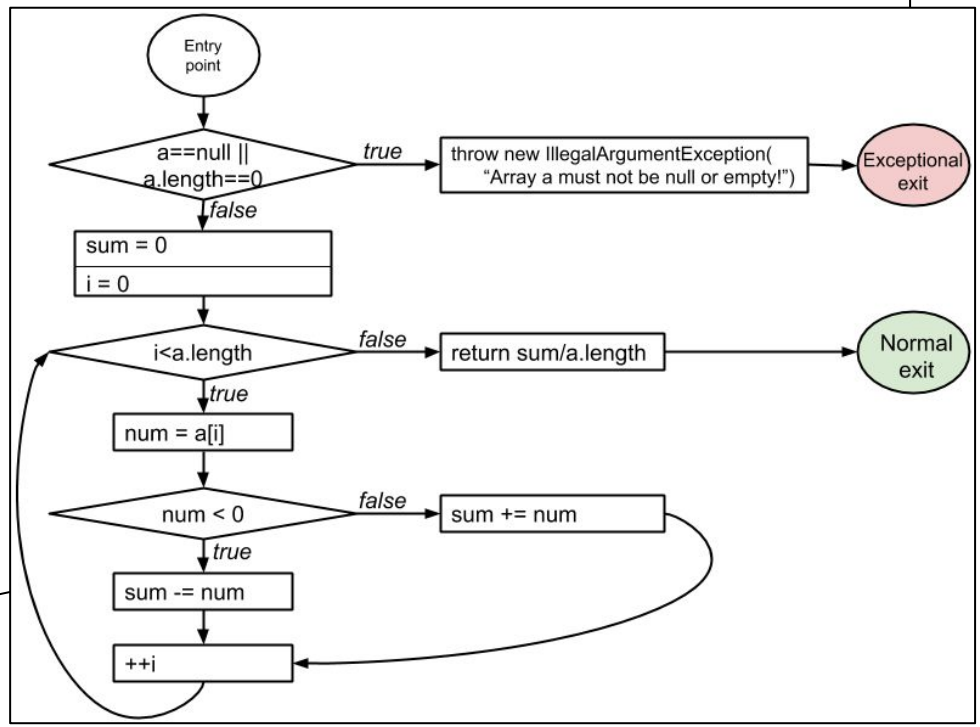
Control flow graph



Control flow graph

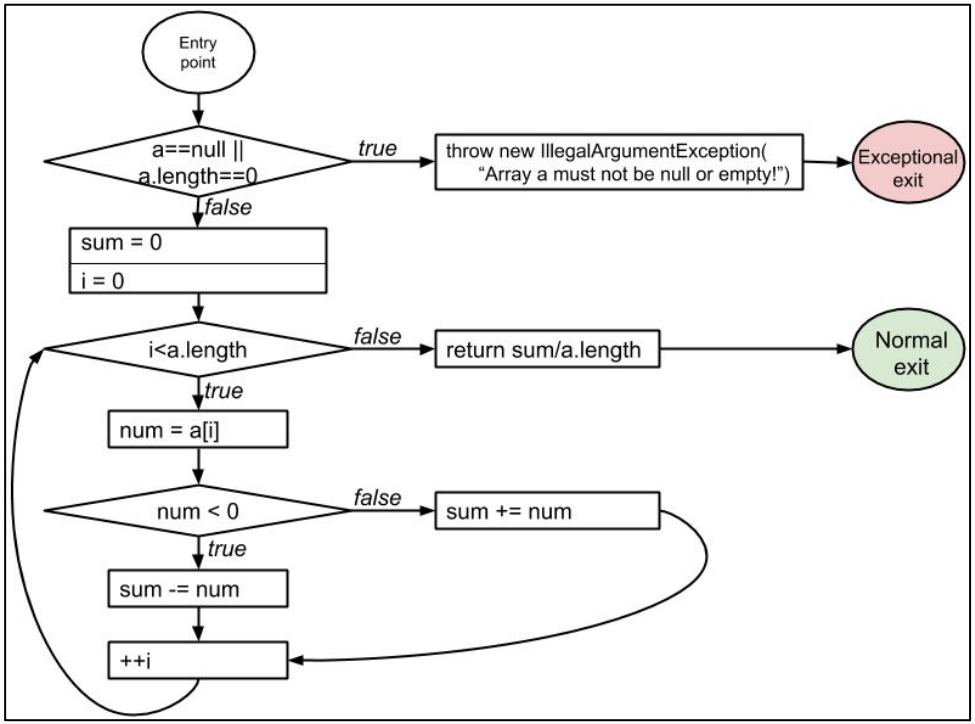
Average of the absolute values of an array of doubles

```
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("Array numbers must not be 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;
}
```

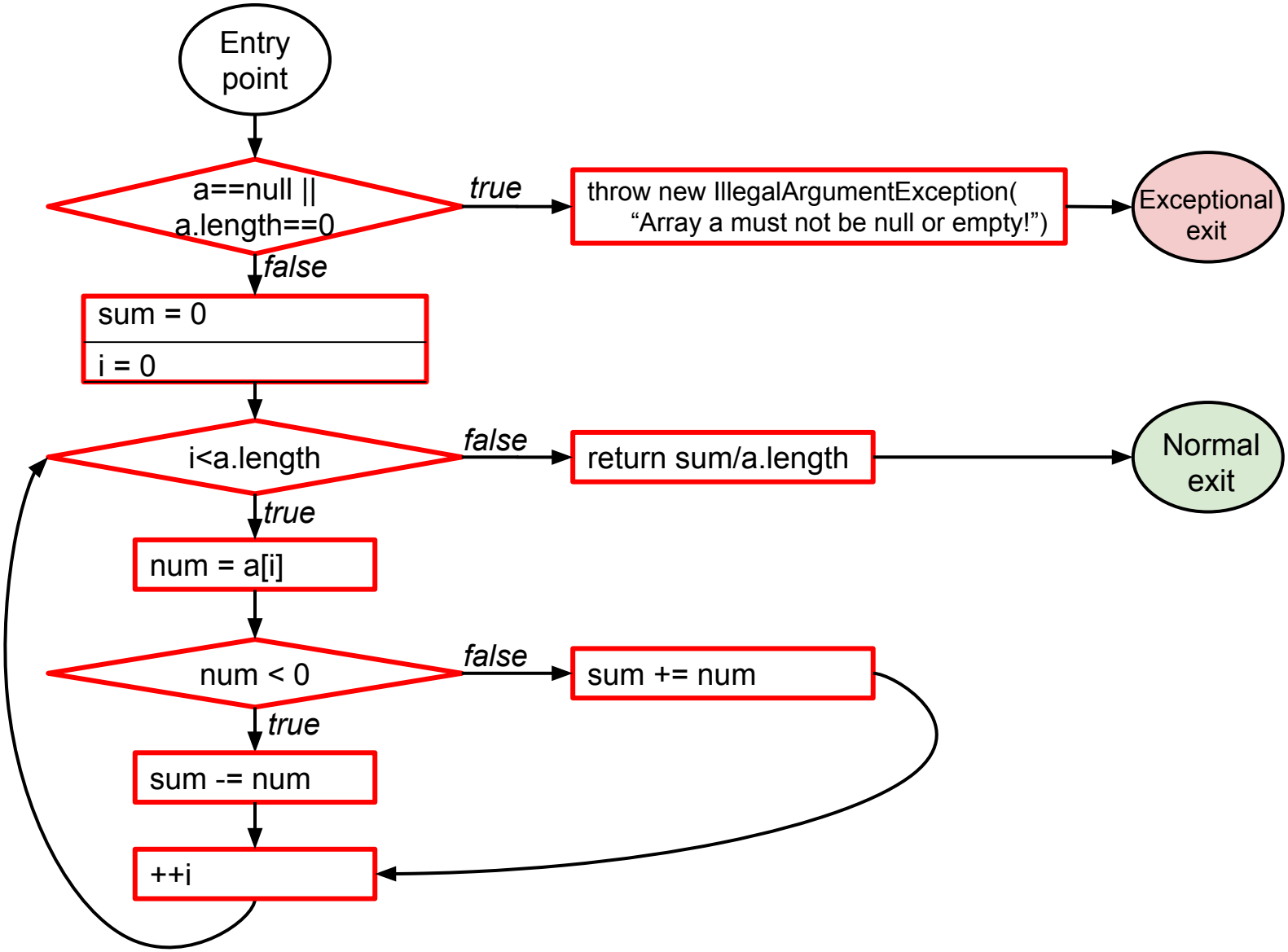


Statement coverage

- Every **statement** in the program must be executed at least once.

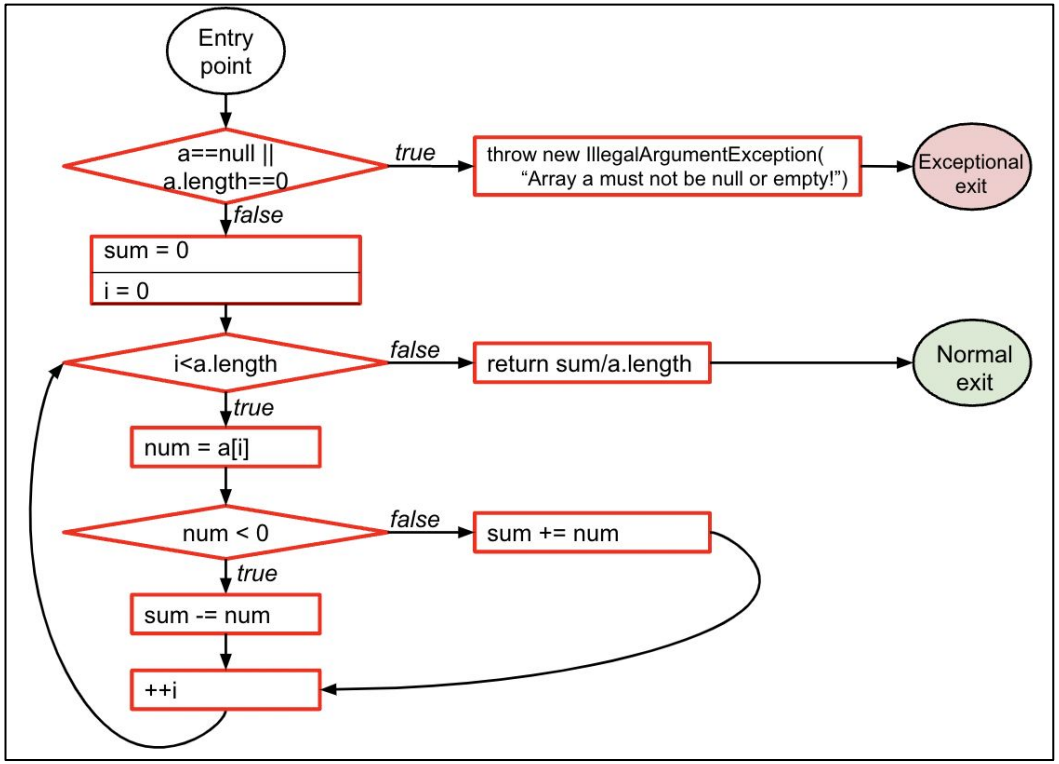


Statement coverage



Statement coverage

- Every **statement** in the program must be executed at least once.
- This is **node coverage** in the control-flow graph (CFG).



100% coverage

- Usually unachievable (dead code)
- Prohibitively expensive

Statement coverage is not enough

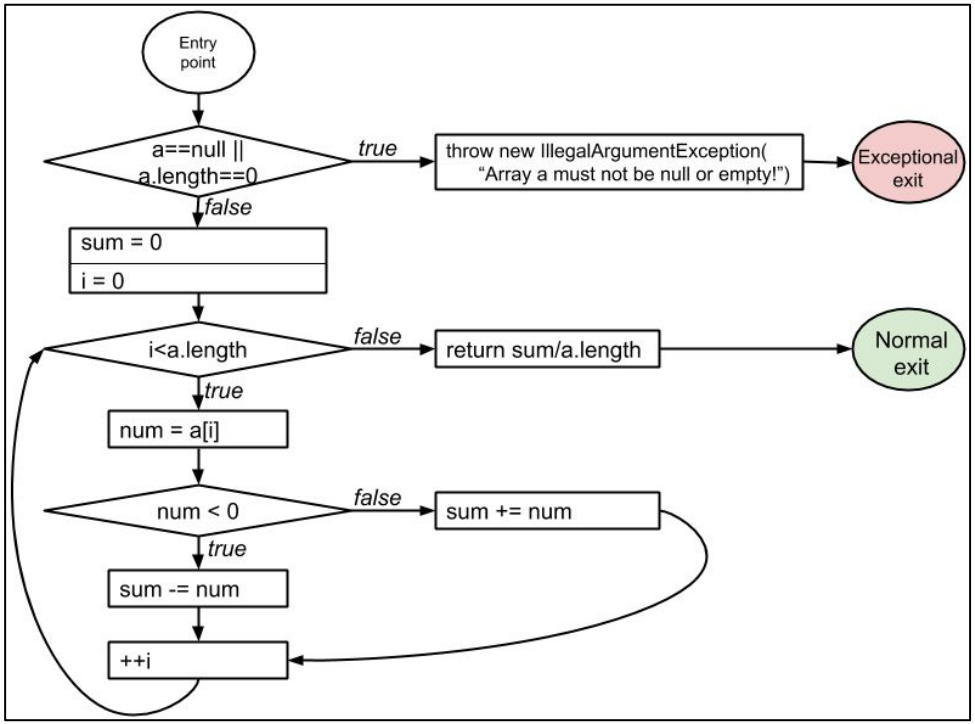
```
int min(int a, int b) {
    int result = a;
    if (a <= b) {
       result = a;
    }
    return result;
}
```

Consider any test with a ≤ b, e.g., **`min(1,2)`**

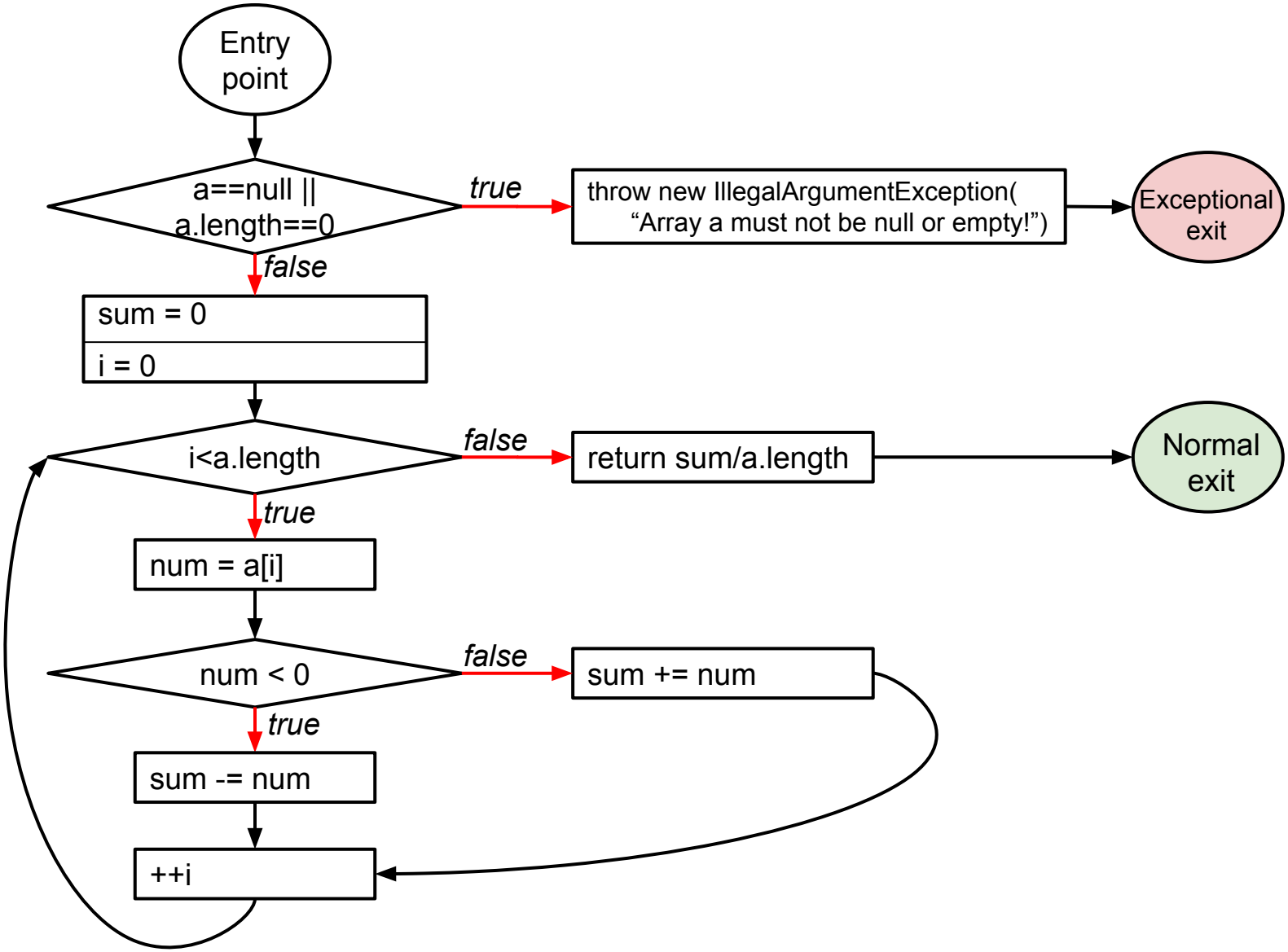
- It executes every instruction
- It misses the defect

Branch coverage

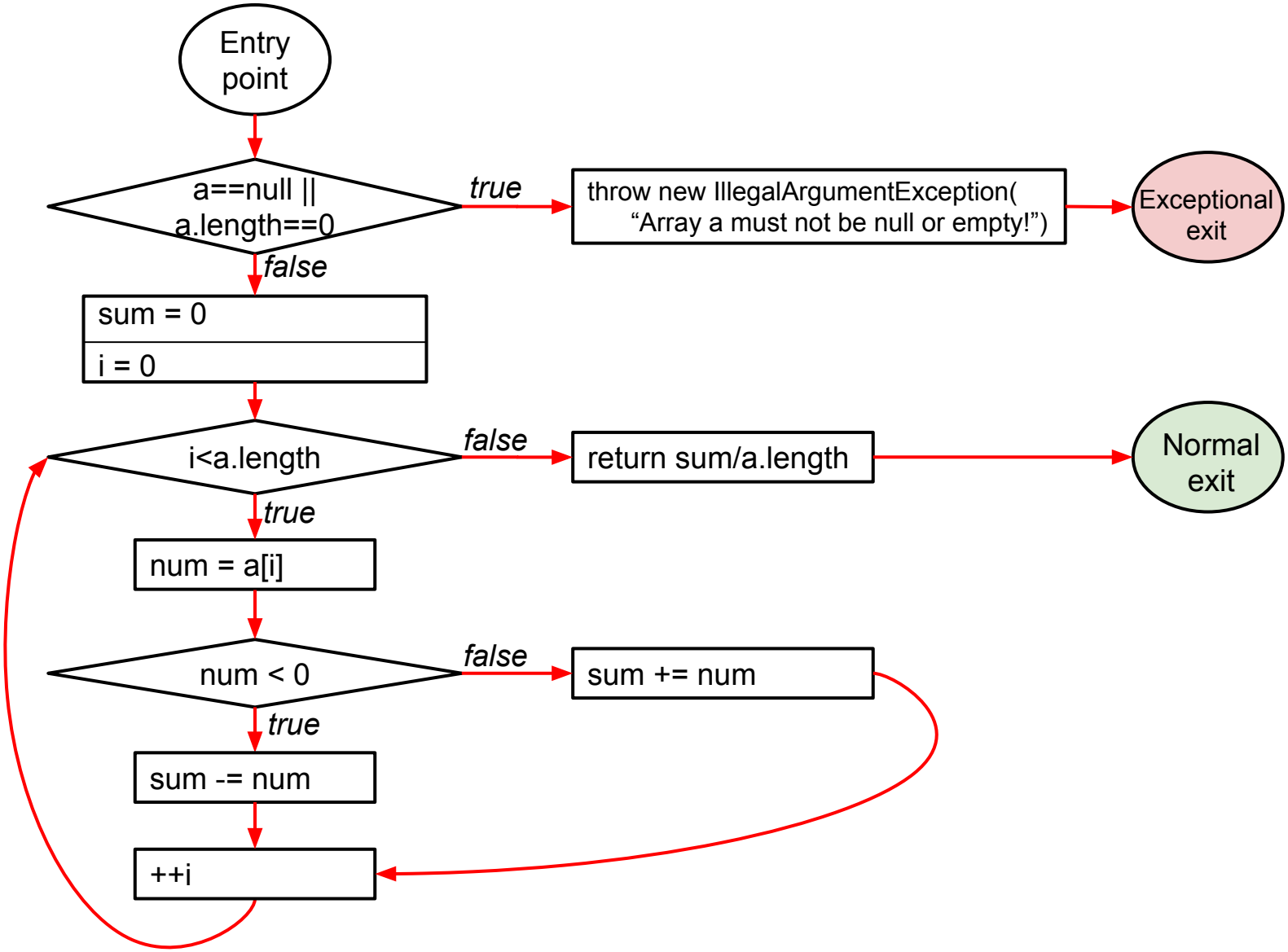
- Every **branch** in the program must be executed at least once.
- A branch is each outcome of a conditional statement's test: `if`, `for`, `while`, ...



Branch coverage

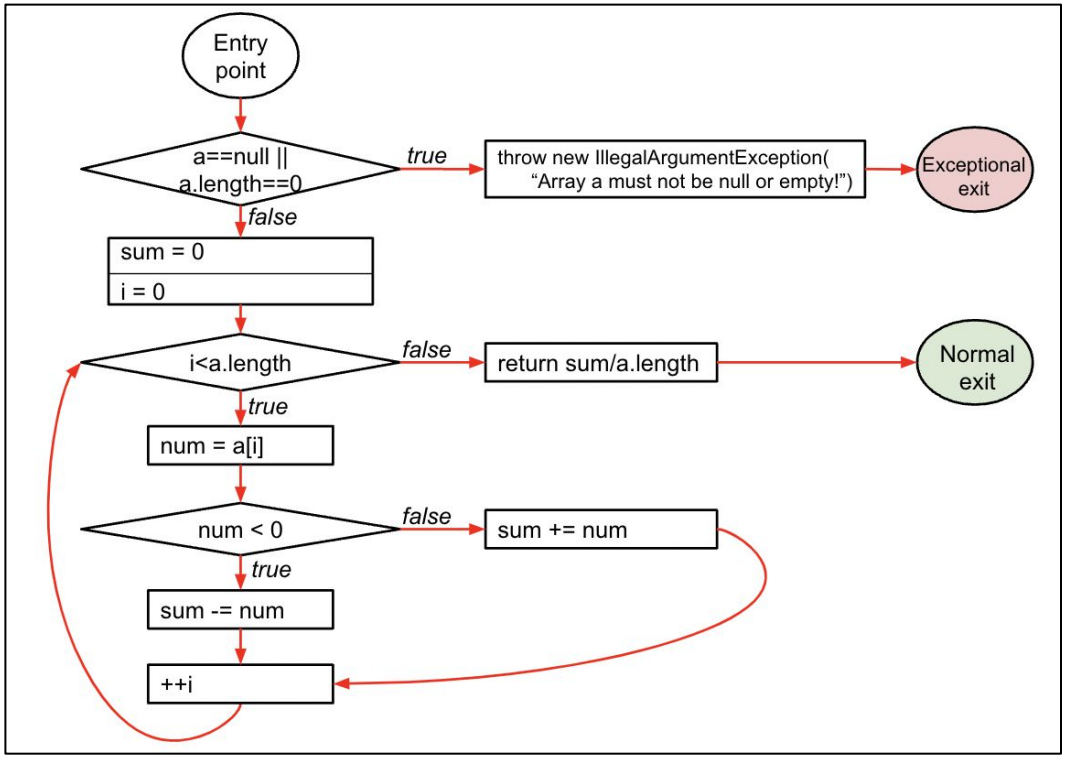


Branch coverage



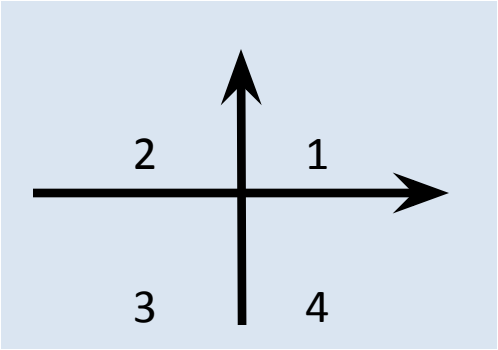
Branch coverage

- Every **branch** in the program must be executed at least once.
- This is **edge coverage** in the CFG.



Branch coverage is not enough

```
int quadrant(int x, int y) {
  int answer;
  if (x >= 0)
    answer = 1;
  else
    answer = 2;
  if (y < 0)
    answer = 4;
  return answer;
}
```

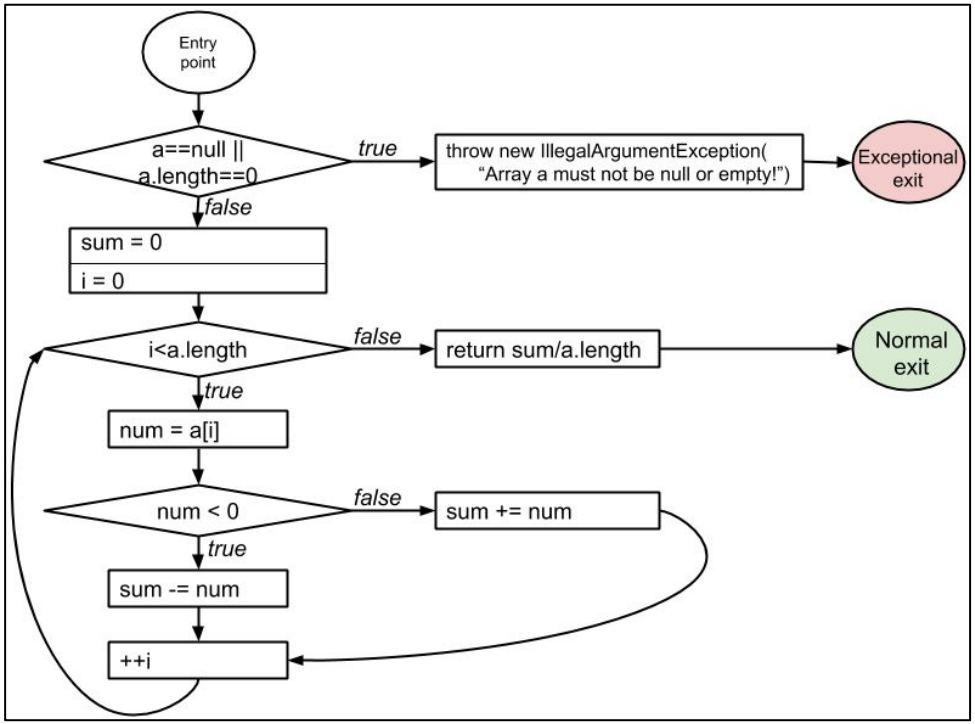


Consider a suite with two test inputs: (2,-2) and (-2,2)

- Achieves 100% branch coverage
- Misses the bug

Branch coverage = decision coverage

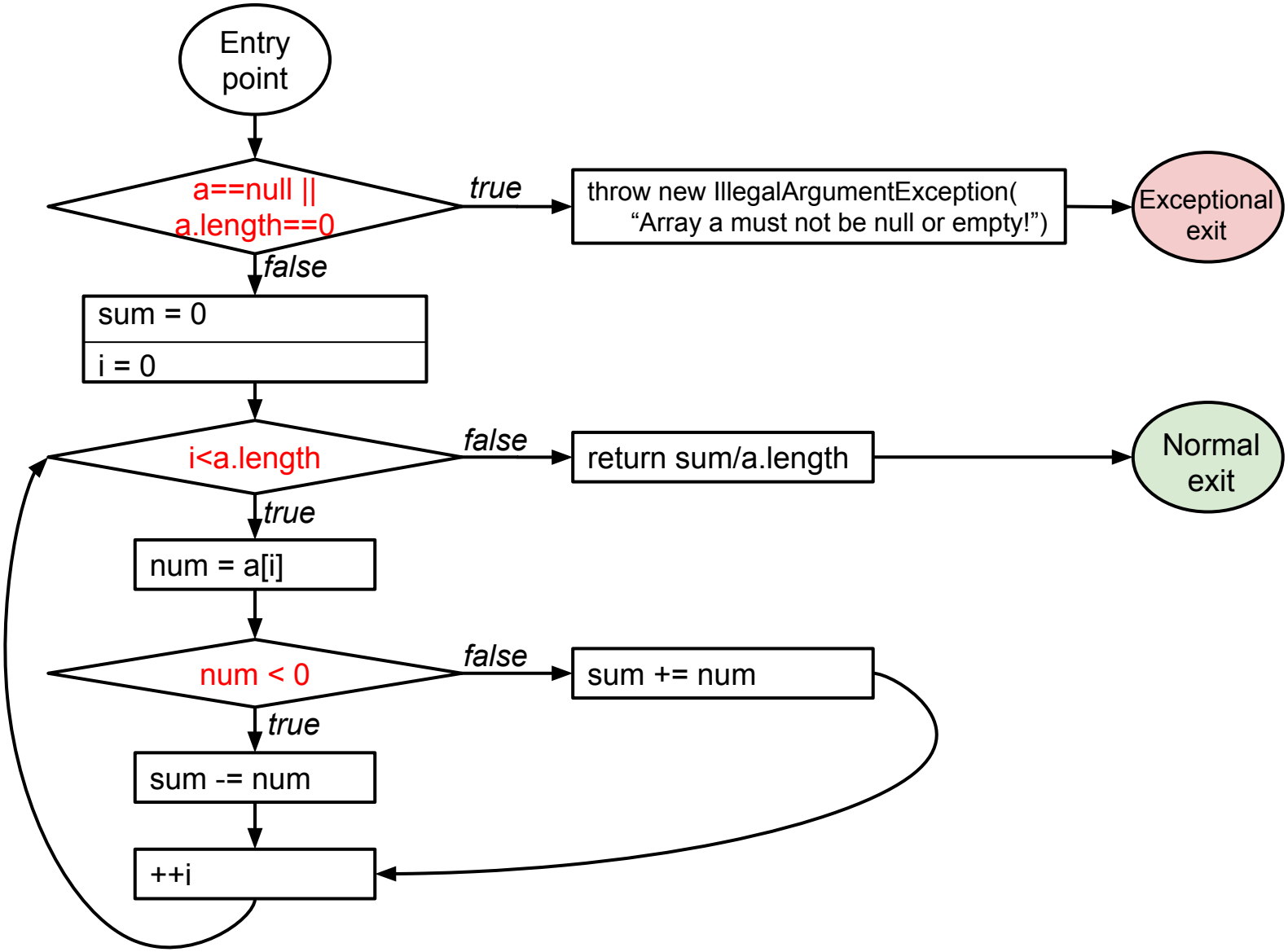
- Every **decision** in the program must take on **all possible outcomes** (true/false) at least once.



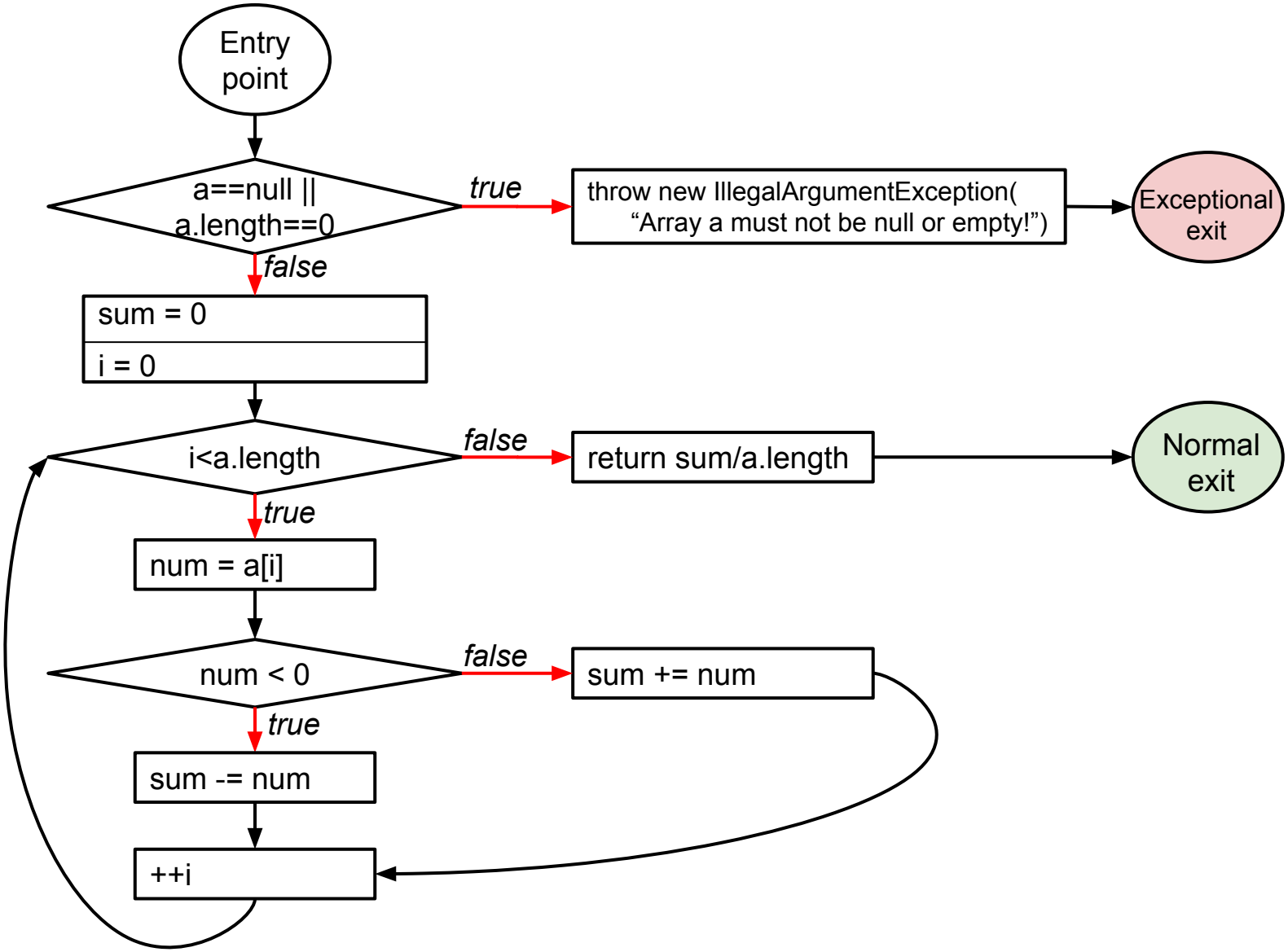
Terminology: conditions and decisions

- **Condition**: an atomic boolean expression
  - contains no smaller boolean expressions
- **Decision**: a maximal boolean expression in the source code
  - decision = one or more conditions joined with logical connectors
- **Example:** if (*a* || *b*) { … }
  - "*a*" and "*b*" are *conditions.*
  - "*a* || *b*" is a *decision.*

Decision coverage

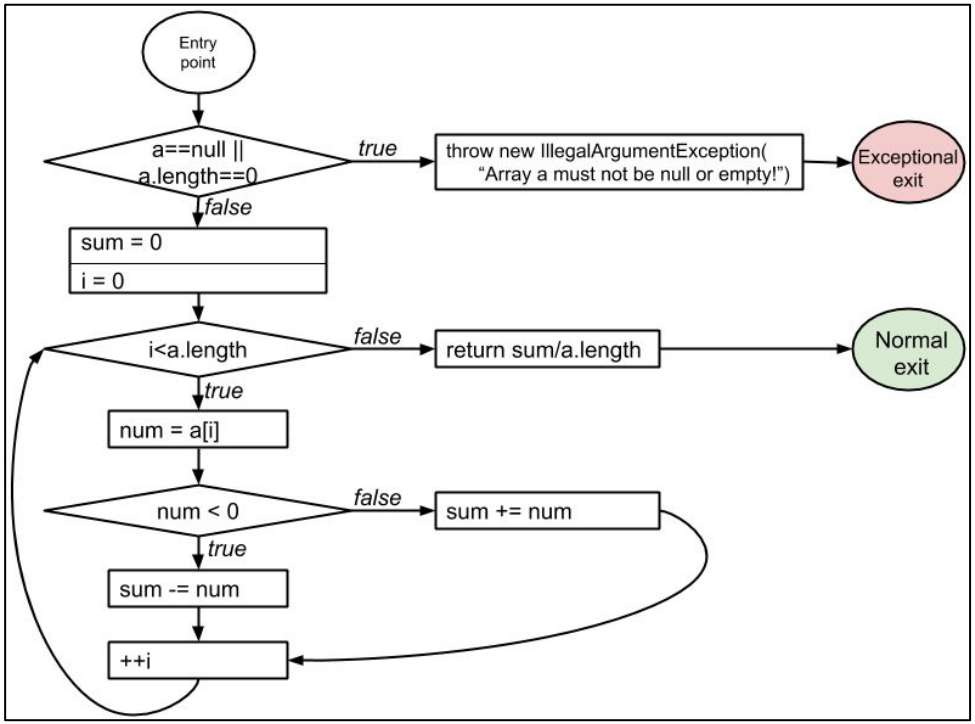


Decision coverage = branch coverage

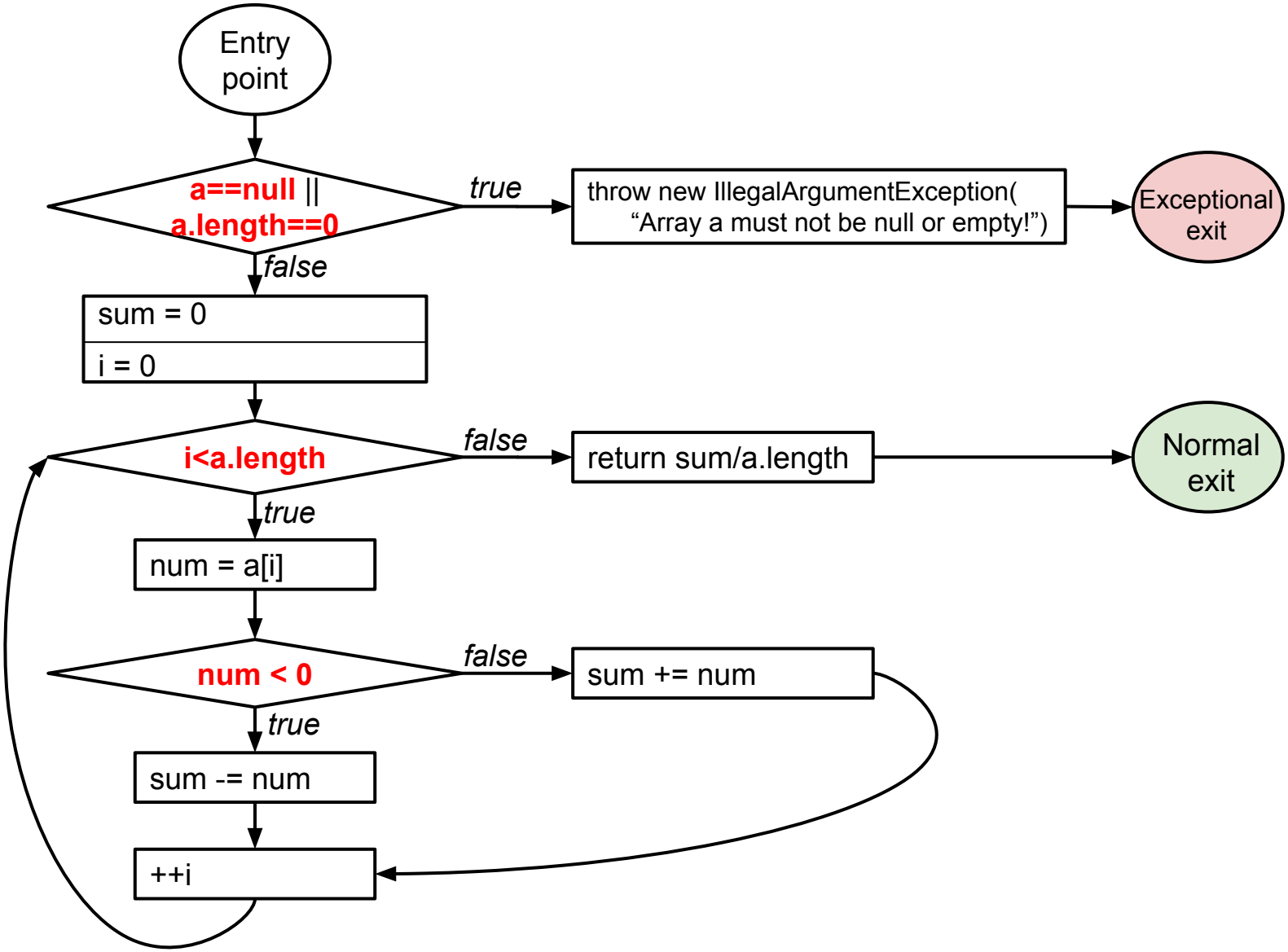


Condition coverage

- Every **condition** in the program must take on **all possible outcomes** (true/false) at least once.

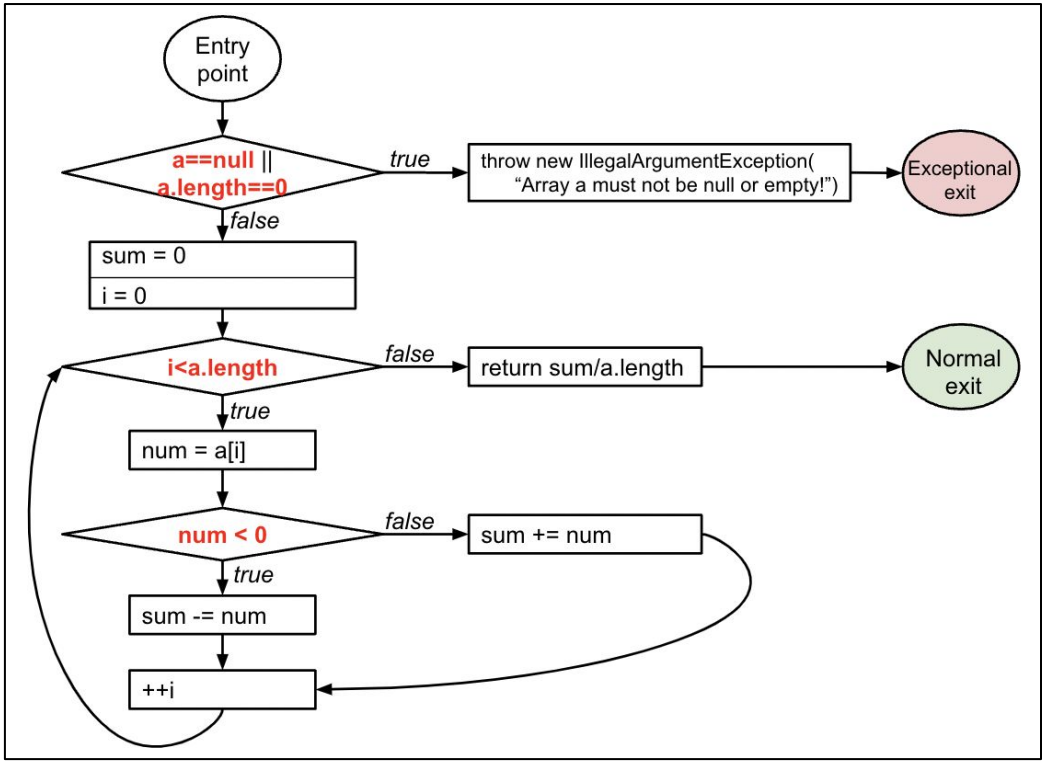


Condition coverage



Condition coverage

- Every **condition** in the program must take on **all possible outcomes** (true/false) at least once.
- Think of another explanation for condition coverage, in terms of **the binary**.



Structural code coverage: subsumption

Given two coverage criteria A and B,

A subsumes B iff **satisfying A implies satisfying B**

Subsumption relationships:

1. Does statement coverage subsume branch (decision) coverage?
2. Does branch (decision) coverage subsume statement coverage?

If so, give a brief argument.

If not, give a counterexample.

In: if (*a* || *b*) { … },
"*a* || *b*" is a *decision.*

Structural code coverage: subsumption

Given two coverage criteria A and B,
A subsumes B iff **satisfying A implies satisfying B**

Subsumption relationships:

1. **Statement** coverage **does not subsume decision** coverage
2. **Decision** coverage **subsumes statement** coverage

In: if (*a* || *b*) { … },
"*a* || *b*" is a *decision.*

Structural code coverage: subsumption

Given two coverage criteria A and B,

A subsumes B iff **satisfying A implies satisfying B**

Subsumption relationships:

1. Does decision coverage subsume condition coverage?
2. Does condition coverage subsume decision coverage?

If so, give a brief argument.

If not, give a counterexample.

In: if (*a* || *b*) { … }

- "*a* || *b*" is a *decision.*
- "*a*" and "*b*" are *conditions.*

Structural code coverage: subsumption

Given two coverage criteria A and B,
A subsumes B iff **satisfying A implies satisfying B**

Subsumption relationships:

1. **Decision** coverage **does not subsume condition** coverage
2. **Condition** coverage **does not subsume decision** coverage

In: if (*a* || *b*) { … }

- "*a* || *b*" is a *decision.*
- "*a*" and "*b*" are *conditions.*

Decision coverage vs. condition coverage

4 possible tests for the decision *a* | *b*:

1. *a* = 0, *b* = 0
2. *a* = 0, *b* = 1
3. *a* = 1, *b* = 0
4. *a* = 1, *b* = 1

a	*b*	*a* | *b*
0	0	0
0	**1**	**1**
1	**0**	**1**
1	1	1

Satisfies **condition coverage** but **not decision coverage**

a	*b*	*a* | *b*
0	**0**	**0**
0	**1**	**1**
1	0	1
1	1	1

Does **not** satisfy **condition coverage** but **decision coverage**

Neither coverage criterion subsumes the other!

MCDC: Modified condition and decision coverage

- **Decision coverage** (every decision is both true & false)
- **Condition coverage** (every condition is both true & false)
- Each **condition independently affects** its decision's outcome.
  Hold other conditions fixed, vary that condition, decision changes.

Required for safety critical systems (DO-178B/C)

MCDC: an example

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

a	b	Outcome
0	0	0
0	1	1
1	0	1
1	1	1

MCDC

- **Decision** coverage
- **Condition** coverage
- Each **condition independently affects** its decision

Which tests (combinations of a and b) satisfy MCDC?

MCDC: an example

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

a	b	Outcome
0	**0**	**0**
0	**1**	**1**
1	**0**	**1**
1	1	1

MCDC

- **Decision** coverage
- **Condition** coverage
- Each **condition independently affects** its decision

MCDC is still cheaper than testing all possible combinations.

MCDC: another example

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

a	b	Outcome
0	0	0
0	1	1
1	0	1
1	1	1

MCDC

- **Decision** coverage
- **Condition** coverage
- Each **condition independently affects** its decision

Why is this example different?

MCDC: another example

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

a	b	Outcome
0	0	0
0	1	1
1	--	1
1	--	1

MCDC

- **Decision** coverage
- **Condition** coverage
- Each **condition independently affects** its decision

Short-circuiting operators may not evaluate all conditions.

MCDC: yet another example

```
if (!a) { ... if (a || b) ... }
```

a	b	Outcome
0	0	0
0	1	1
1	0	1
1	1	1

MCDC

- **Decision** coverage
- **Condition** coverage
- Each **condition independently affects** its decision

What about this example?

MCDC: yet another example

```
if (!a) { ... if (a || b) ... }
```

a	b	Outcome
0	0	0
0	1	1
X	X	X
X	X	X

MCDC

- **Decision** coverage
- **Condition** coverage
- Each **condition independently affects** its decision

Not all combinations of conditions may be possible.

MCDC: complex expressions

Provide an MCDC-adequate test suite for:

1. `a | b | c`
2. `a & b & c`

MCDC

- **Decision** coverage
- **Condition** coverage
- Each **condition independently affects** its decision

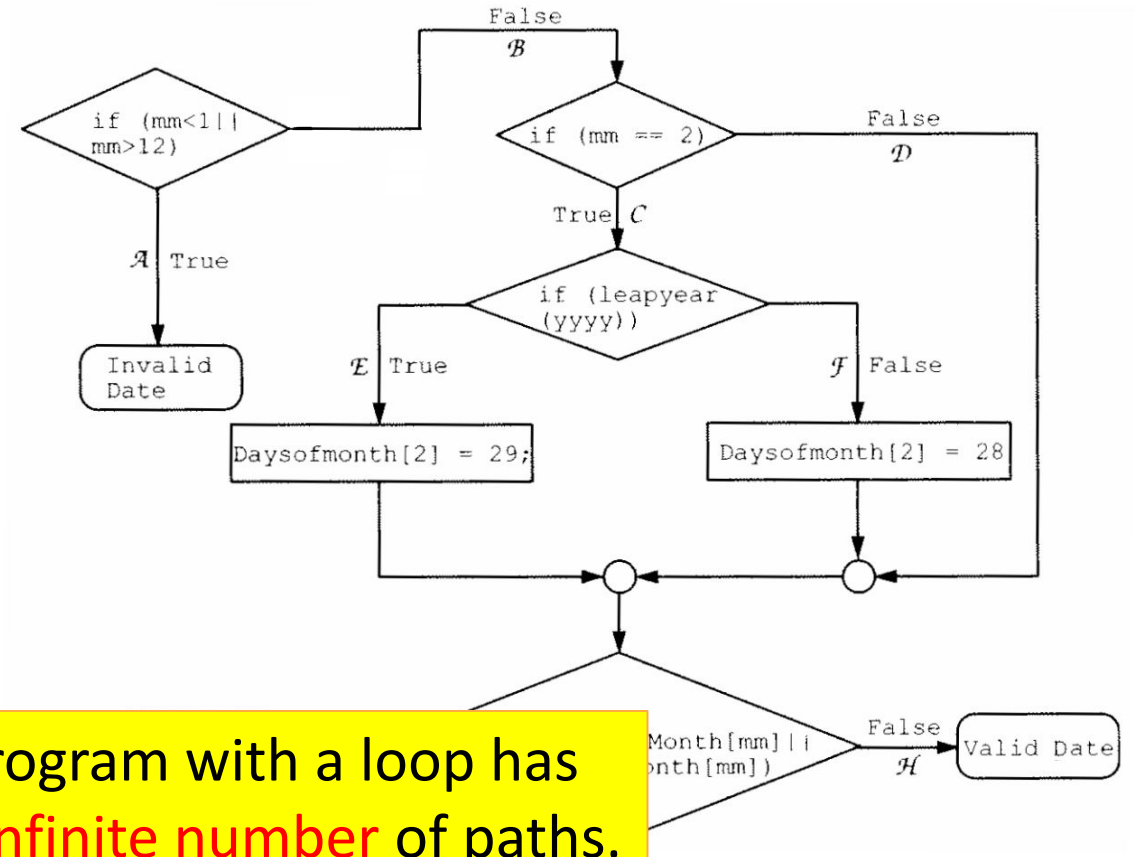
a | b | c

a	b	c
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

a & b & c

a	b	c
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

Path coverage example



A program with a loop has an infinite number of paths.

Other notions of coverage (“program spectra”)

Example use: run a program with dates before & after Y2K

Structural code coverage: summary

Classes in this File	Line Coverage	Branch Coverage	Complexity
Avg	100% 10/10	100% 8/8	6

```
 1    package avg;
 2
 3 4  public class Avg {
 4
 5        /*
 6         * Compute the average of the absolute values of an array of doubles
 7         */
 8        public double avgAbs(double ... numbers) {
 9            // We expect the array to be non-null and non-empty
10 4        if (numbers == null || numbers.length == 0) {
11 2            throw new IllegalArgumentException("Array numbers must not be null or empty!");
12          }
13
14 2        double sum = 0;
15 8        for (int i=0; i<numbers.length; ++i) {
16 6            double d = numbers[i];
17 6            if (d < 0) {
18 2                sum -= d;
19              } else {
20 4                sum += d;
21              }
22          }
23 2        return sum/numbers.length;
24       }
25   }
```

- Code coverage is easy to compute.
- Code coverage has an intuitive interpretation.
- Code coverage is used in industry: Code coverage at Google
- Code coverage itself is not sufficient!

Mutation-based testing: teaser

A better test suite detects more real defects

- One proxy metric: code coverage
- Another proxy metric: detection of fake defects (“mutants”)

Mutant = a small program change

```
a = b + c;   ⟹   a = b - c;
a = b + c;   ⟹   a = b * c;
a = b + c;   ⟹   a = b + 0;
a = b + c;   ⟹   a = 1 + c;
```

Intuitive argument about mutants and real defects:

a mutant might be defective

∧ detecting more mutants means detecting more fake defects

∴ detecting more mutants means the test suite is better

Measuring mutation score

Input: program and test suite

1. Apply *mutation operators* to the program to create many mutants
   Each mutant is a variation of the program
2. For each mutant, run the test suite
3. Count the number of test runs that fail
   If the test run failed, then the test suite "detected" or "killed" the mutant.
4. Mutation score = $\frac{\text{detected mutants}}{\text{all mutants}}$

Discussion of papers

Ice cream causes murder

Murder causes ice cream sales

ChatGPT explanation

1. Stress and Emotional Response: During times of increased violent crime, individuals within a community may experience heightened levels of stress, fear, or anxiety due to concerns for their safety or the safety of their loved ones. This elevated emotional state might lead some people to seek out comfort foods as a way to cope with their emotions.

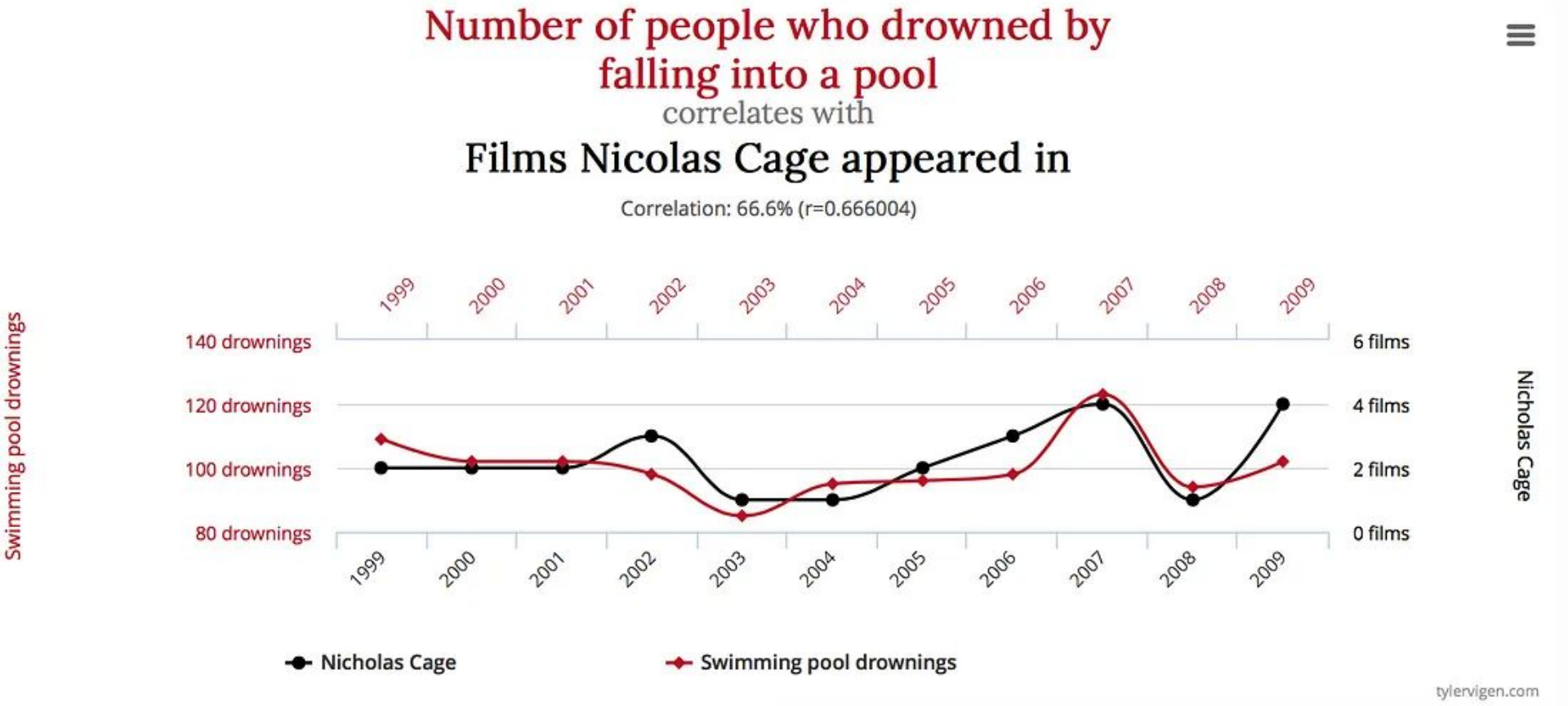
2. Psychological Response: Ice cream, being a popular comfort food, could serve as a psychological coping mechanism for individuals experiencing stress. The act of consuming ice cream may trigger the release of neurotransmitters such as serotonin and endorphins, which are associated with feelings of pleasure and relaxation. As a result, some individuals may be more inclined to purchase and consume ice cream during periods of heightened stress or emotional distress.

3. Behavioral Patterns: An increase in violent crime may lead to changes in individuals' behavior, including their consumption habits. People may be more likely to indulge in comfort foods like ice cream as a way to temporarily alleviate their stress or anxiety. This could result in a noticeable uptick in ice cream sales within communities experiencing higher levels of violent crime.

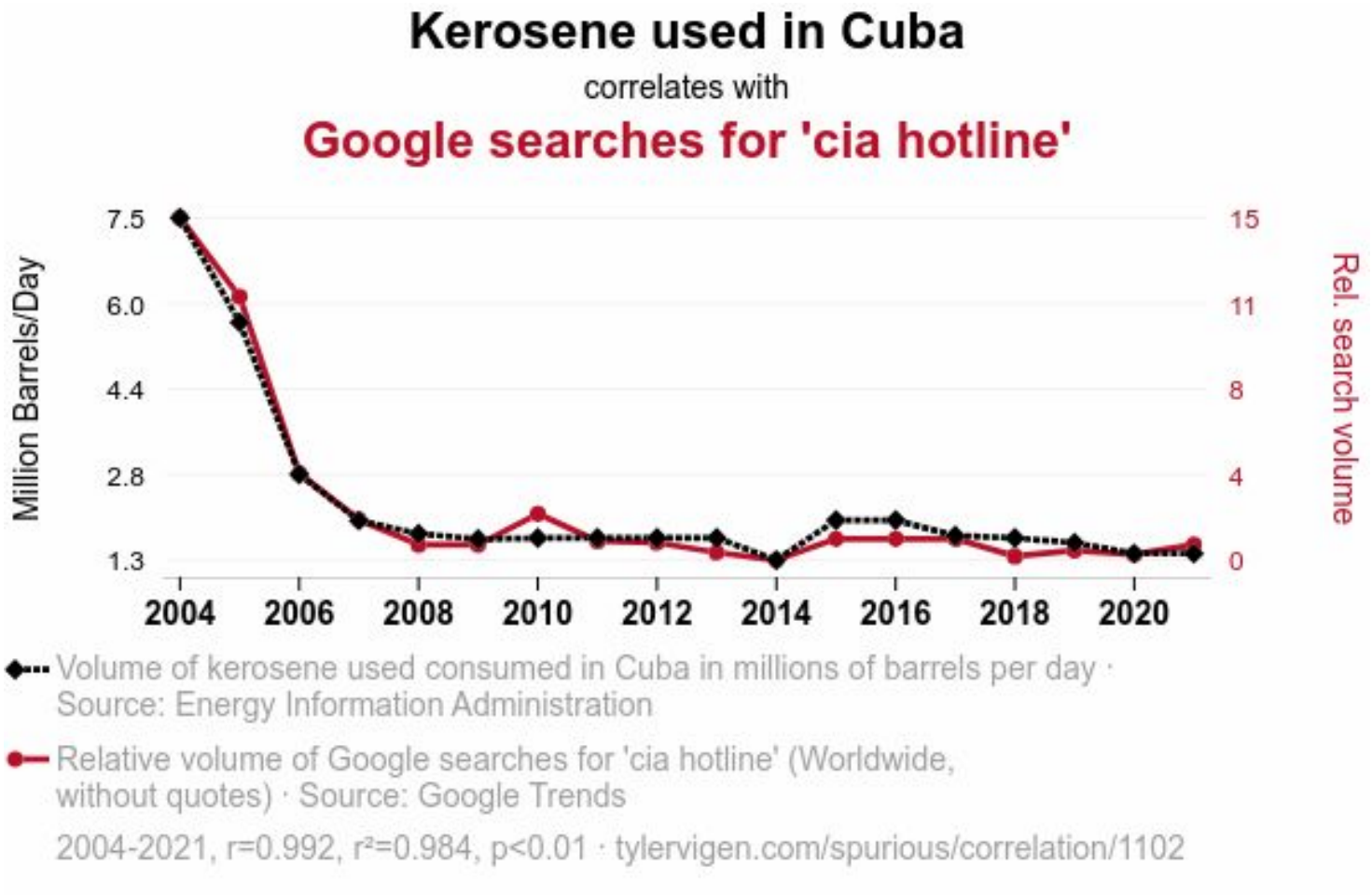
4. Cultural Factors: Cultural norms and societal responses to stress or trauma may play a role in shaping individuals' behaviors. In some communities, the consumption of certain foods, such as ice cream, may be ingrained as a socially acceptable way to cope with difficult emotions or situations.

Ice cream and murder

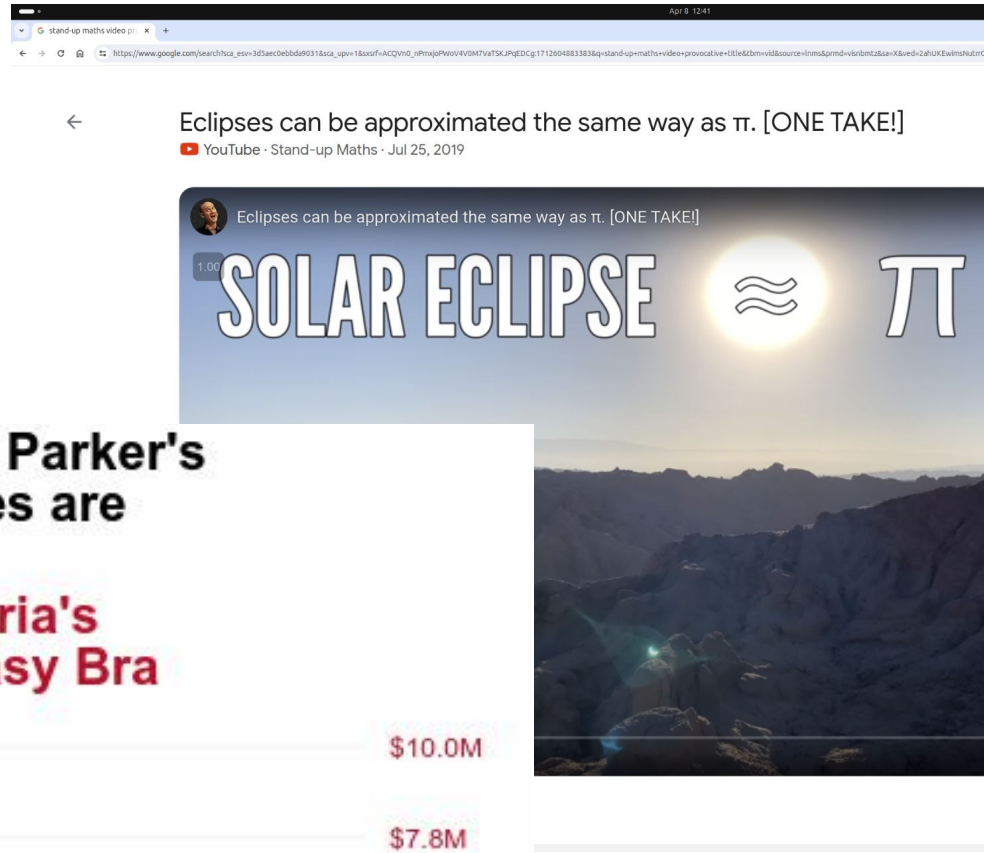
Correlation or causation?

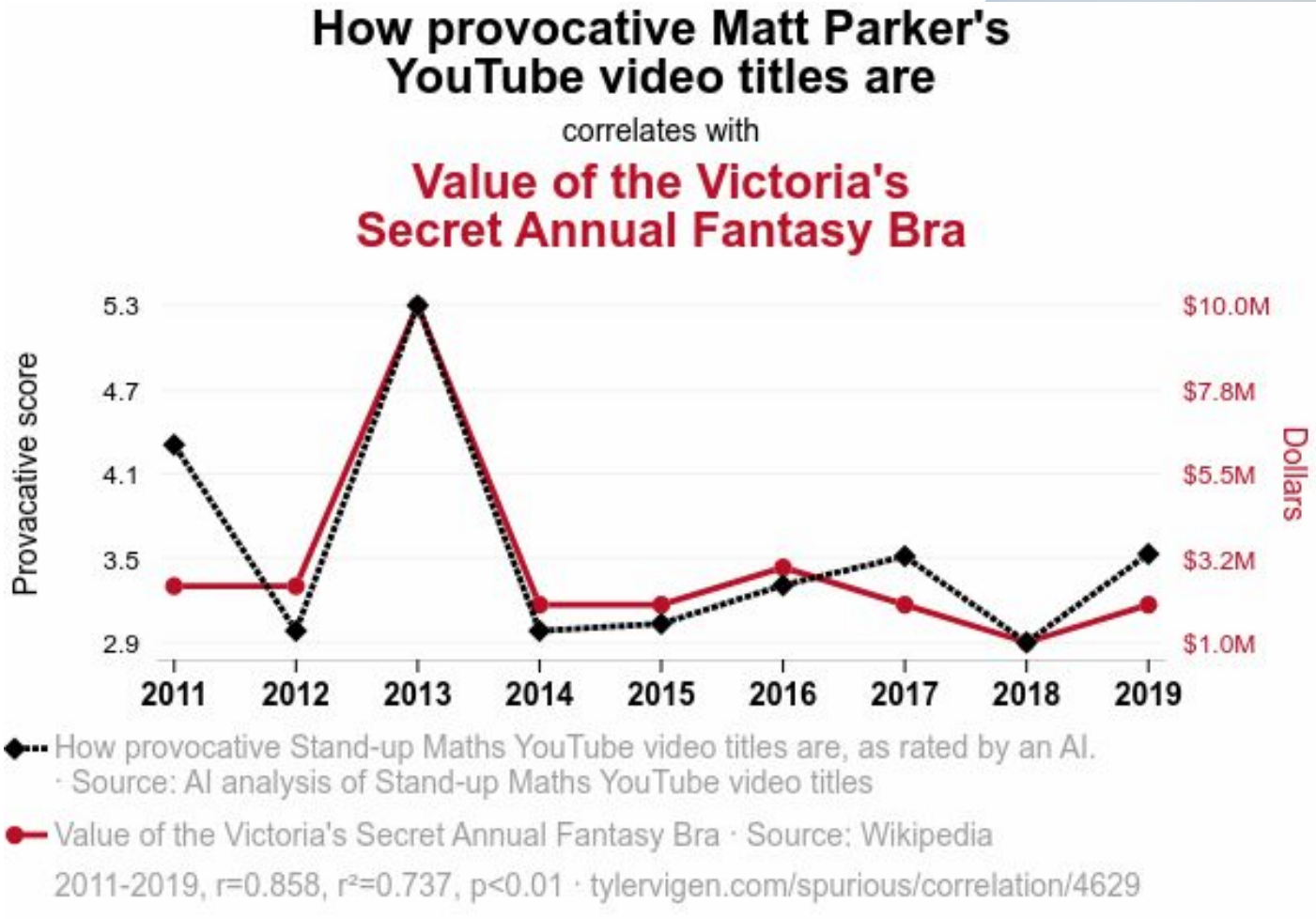


Correlation or causation?

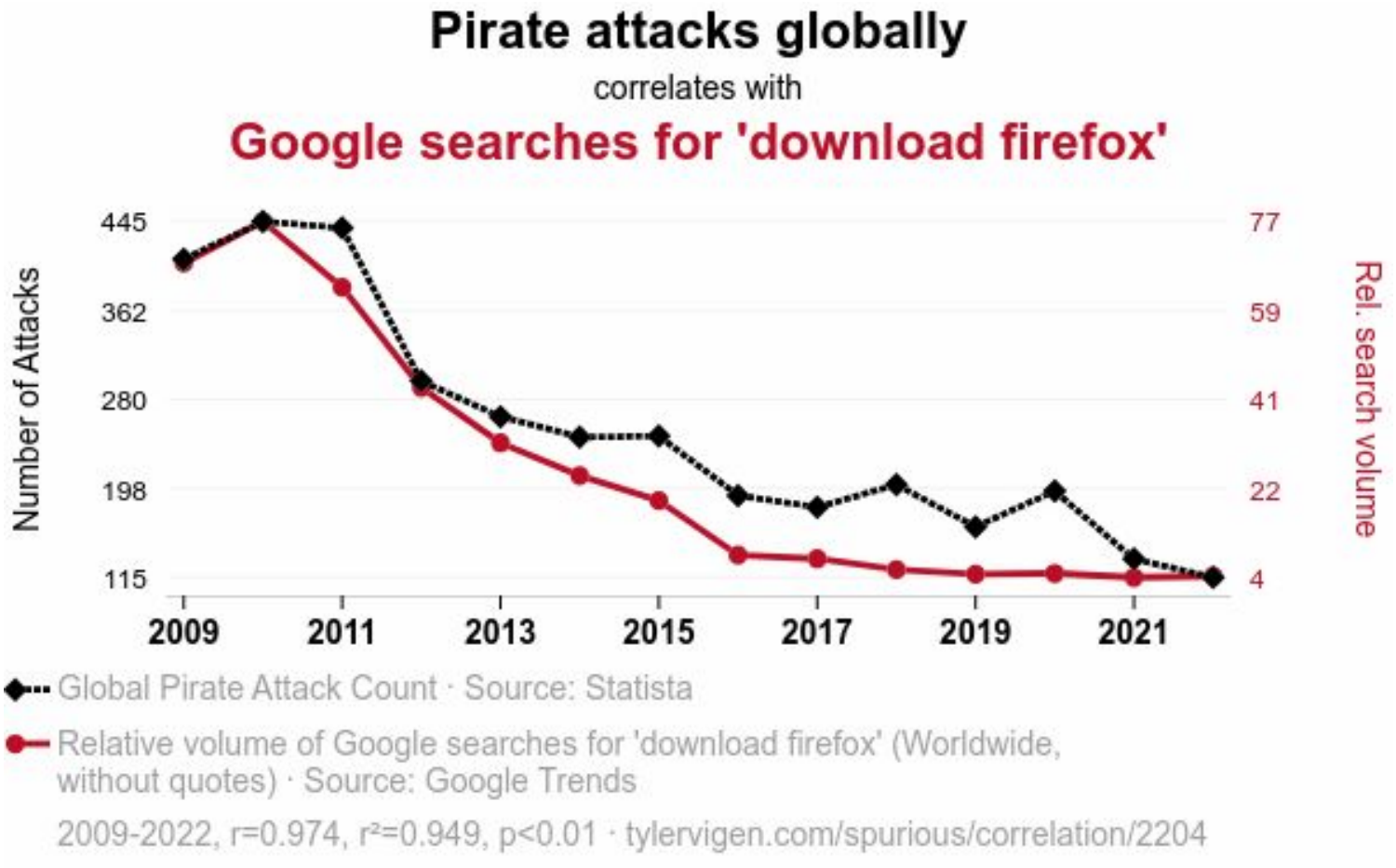


Correlation or causation?

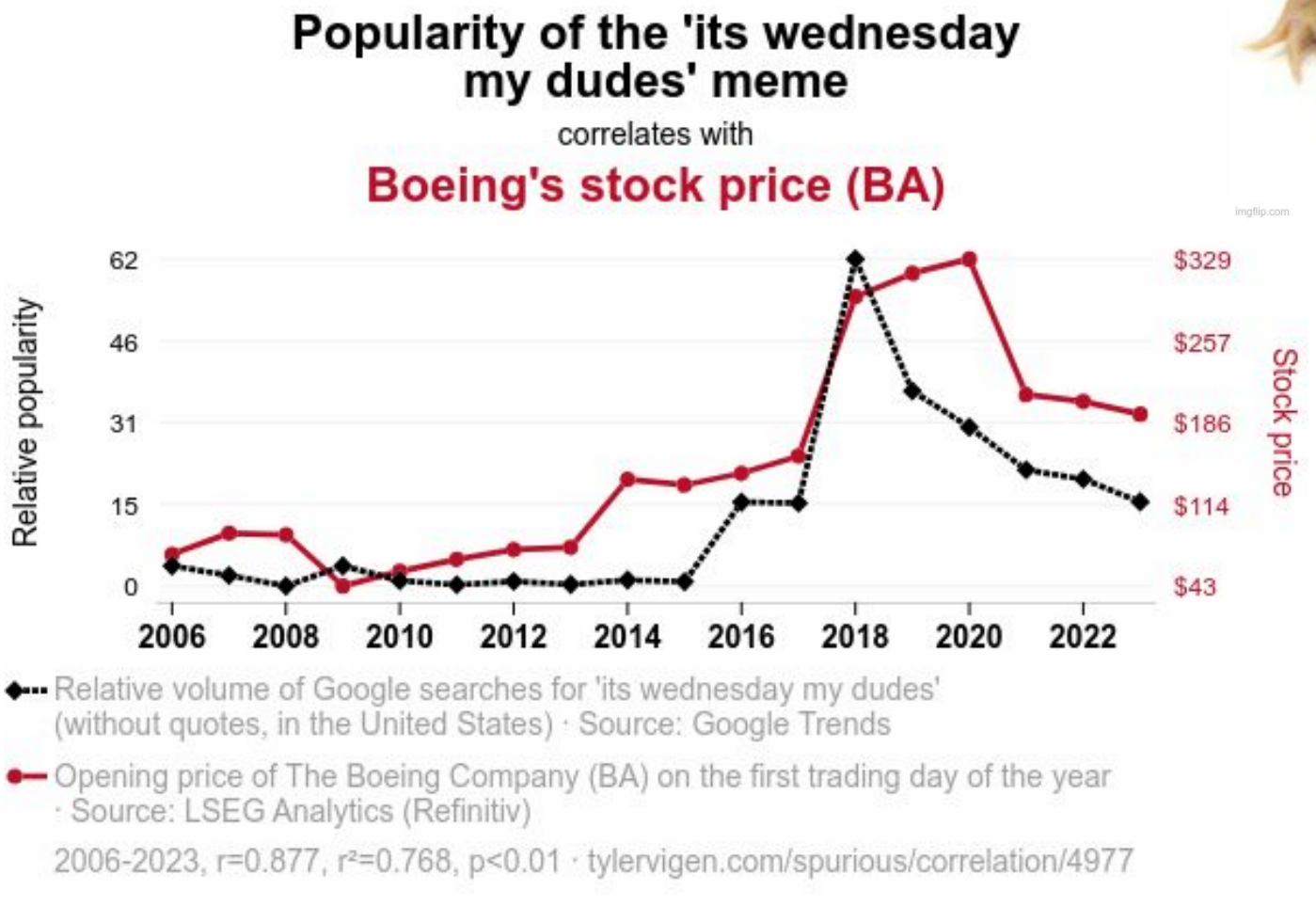




Correlation or causation?



Correlation or causation?



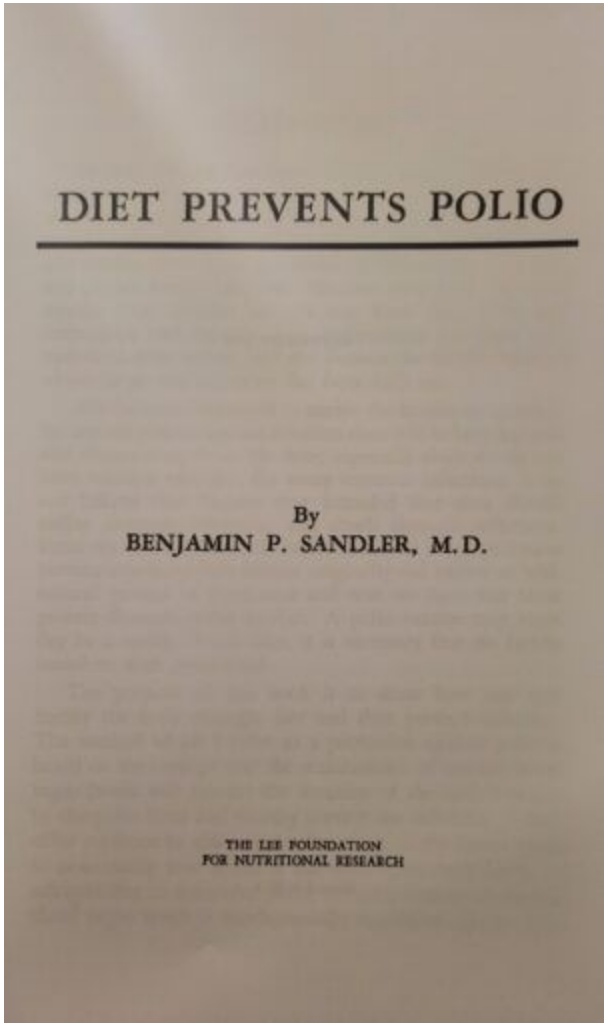
Ice cream causes polio

Benjamin Sandler showed a correlation between sugar consumption and polio

- By country
- By month
  - Polio is at its height in summer when sugar intake is highest (ice cream, soft drinks, ...)

Articles in the 1940s; 1951 book

Reduced ice cream sales by over 1 million gallons per week



Discussion of papers