Program Testing and Analysis: Manual Testing

Dr. Michael Pradel

Software Lab, TU Darmstadt

Partly based on slides from Peter Müller, ETH Zurich

Warm-up Quiz

What does the following code print?

```
function Person(name) { this.name = name; };
Person.prototype = {
  isJoe: function() {
    return this.name === "Joe";
  }
};
var p1 = new Person("Peter");
var p2 = p1;
p1.name = "Joe";
console.log(p2.isJoe());
```

false **true** **Something else**

Warm-up Quiz

What does the following code print?

```
function Person(name) { this.name = name; };
Person.prototype = {
  isJoe: function() {
    return this.name === "Joe";
  }
};
var p1 = new Person("Peter");
var p2 = p1;
p1.name = "Joe";
console.log(p2.isJoe());
```

false **true** **Something else**

Warm-up Quiz

What does the following code print?

```
function Person(name) { this.name = name; };
Person.prototype = {
  isJoe: function() {
    return this.name === "Joe";
  }
};
var p1 = new Person("Peter");
var p2 = p1;
p1.name = "Joe";
console.log(p2.isJoe());
```

Constructor and its prototype

false **true** **Something else**

Warm-up Quiz

What does the following code print?

```
function Person(name) { this.name = name; };
Person.prototype = {
  isJoe: function() {
    return this.name === "Joe";
  }
};
var p1 = new Person("Peter");
var p2 = p1;
p1.name = "Joe";
console.log(p2.isJoe());
```

Two references to same object

false | **true** | **Something else**

Outline: Operational Semantics

- **Motivation & preliminaries**
- **Abstract syntax of SIMP**
- **An abstract machine for SIMP**
- **Structural operation semantics for SIMP**
  - Small-step semantics
  - Big-step semantics ←

Outline: Manual Testing

- **Overview**
- **Control flow testing**
  - Statement coverage
  - Branch coverage
  - Path coverage
  - Loop coverage
- **Data flow testing**
  - DU-pair coverage
- **Interpretation of coverage**

Goal of Testing

- **Error: Deviation of observed behavior from required/desired behavior**
  - Functional requirements (what the system is supposed to accomplish)
  - Non-functional requirements (performance, usability, etc.)
- **Testing: Process of executing a program to find errors**
- **Successful test: Test that finds errors**

Limitations of Testing

Testing can only show the presence of bugs, never their absence.

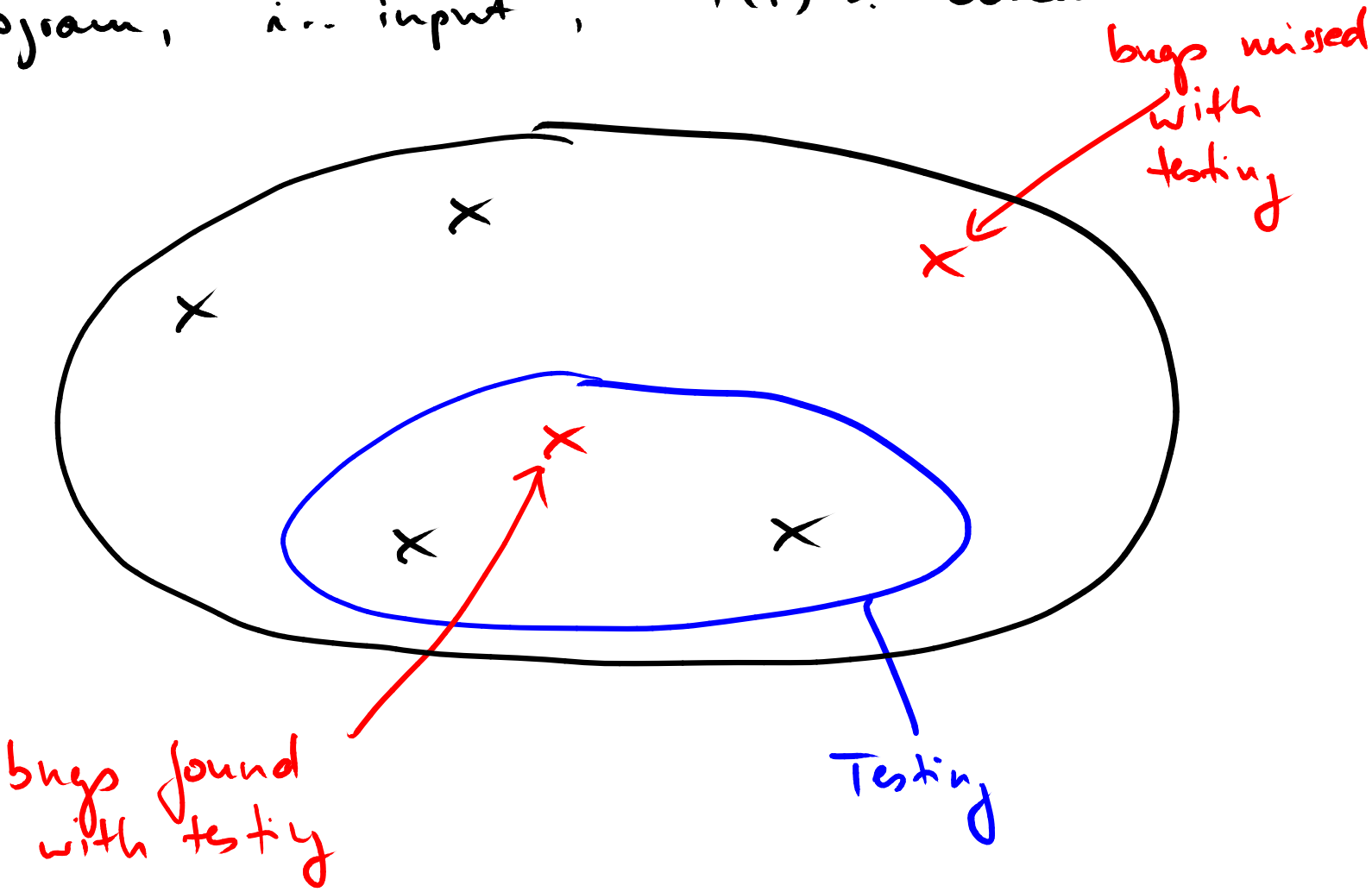
E. W. Dijkstra

Impossible to completely test any non-trivial module or system

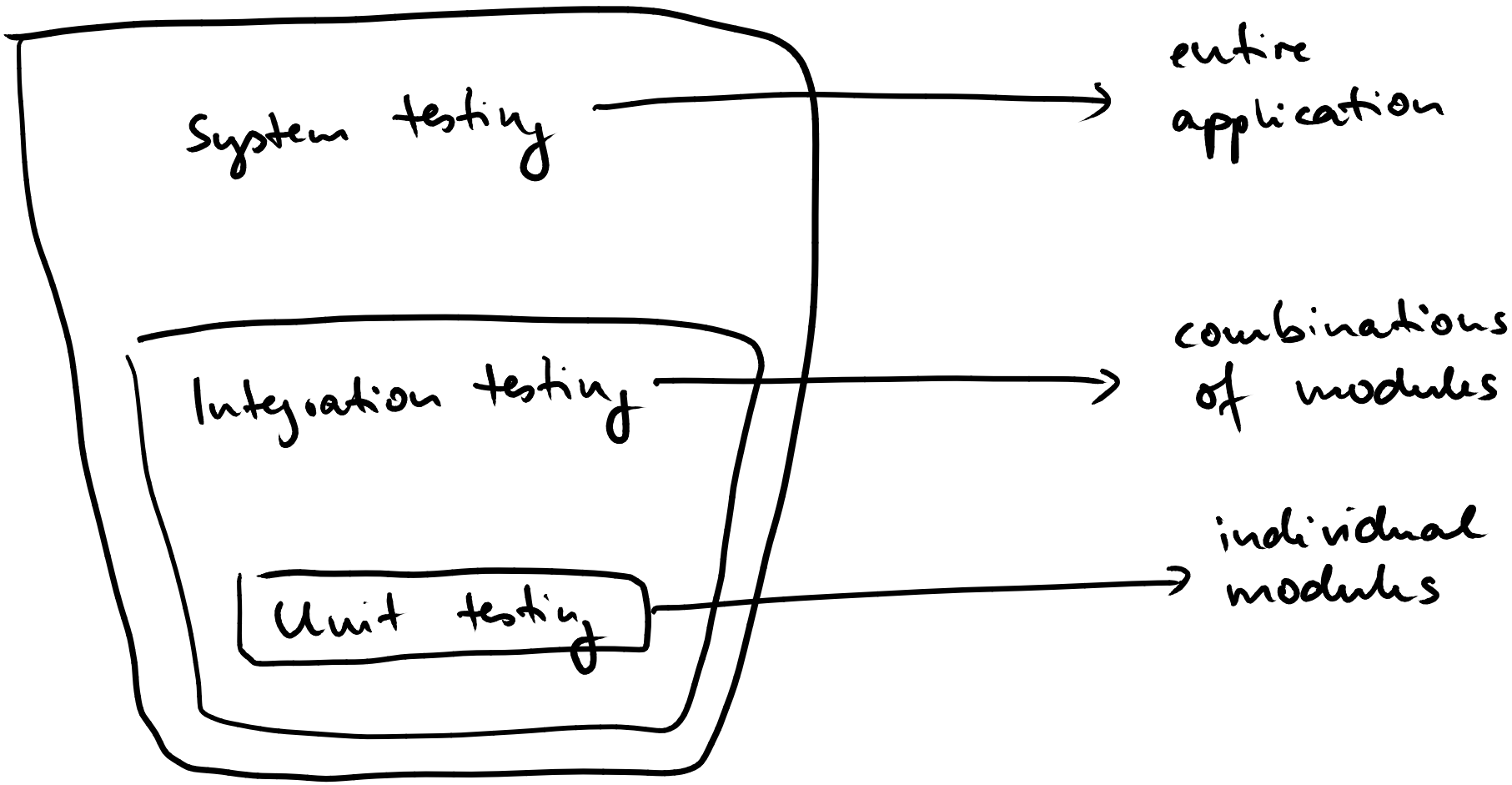
- Theoretical limitations: Termination (halting problem)
- Practical limitations: Prohibitive in time and cost

Over- & underapproximation

P .. program, i .. input, P(i) .. behavior



Level of testing



Testing Approaches

Exhaustive testing

- Goal: Cover all input
- Not feasible for real programs

Random testing

- Goal: Automation
- Black-box testing
- Suitable for all stages

Functional testing

- Goal: Cover all requirements
- Black-box testing
- Suitable for all stages

Structural testing

- Goal: Cover all code
- White-box testing
- Suitable for unit testing

Example: Quadratic Equation

```
// writes result to global variables:
//   x1, x2, and numRoots
function roots(a, b, c /*numbers*/) {
  var q = b * b - 4 * a * c;
  if (q > 0 && a != 0) {
    numRoots = 2;
    var r = Math.sqrt(q);
    x1 = (-b + r) / (2 * a);
    x2 = (-b - r) / (2 * a);
  } else if (q == 0) {
    numRoots = 1;
    x1 = -b / (2 * a);
  } else {
    numRoots = 0;
  }
}
```

$$ax^2 + bx + c = 0$$

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Example: Quadratic Equation

```
// writes result to global variables:
//   x1, x2, and numRoots
function roots(a, b, c /*numbers*/) {
  var q = b * b - 4 * a * c;
  if (q > 0 && a != 0) {
    numRoots = 2;
    var r = Math.sqrt(q);
    x1 = (-b + r) / (2 * a);
    x2 = (-b - r) / (2 * a);
  } else if (q == 0) {
    numRoots = 1;
    x1 = -b / (2 * a);
  } else {
    numRoots = 0;
  }
}
```

Fails if `a==0` and `b*b-4*a*c==0` (division by zero)

$$ax^2 + bx + c = 0$$

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Approach 1: Exhaustive Testing

- Check function for all possible inputs
  $\rightarrow$ Not feasible, even for trivial programs

```
function roots(a, b, c /*numbers*/) {
  ...
}
```

- Assuming that numbers are 64-bit values, we get $(2^{64})^3 \approx 10^{58}$ possible values
- Programs with heap structures:
  Much larger state space

Approach 2: Random Testing

Naive approach:

Select test data uniformly

```
function roots(a, b, c /*doubles*/) {
  ...
}
```

- Fails if `a==0` and `b*b-4*a*c==0` (division by zero)
- Probability of selecting `a==0` and `b==0` is $\frac{1}{10^{38}}$

Random Testing: Discussion

- **Main advantage: Generates test data fully automatically**
  - Avoids programmer/tester bias
  - Tests robustness, especially handling of invalid and unusual input
- **If all inputs considered equally valuable:**
  Small chance to hit a particular bug
- **We'll see more clever approaches later**

Approach 3: Functional Testing

Use specification:

Given three values a, b, c, compute all solutions of $ax^2 + bx + c = 0$.

Two solutions:	One solution:	No solution:
■ $a \neq 0$ and $b^2 - 4ac > 0$	■ $a = 0$ and $b \neq 0$, or ■ $a \neq 0$ and $b^2 - 4ac = 0$	■ $a = 0$, $b = 0$, and $c \neq 0$, or ■ $a \neq 0$ and $b^2 - 4ac < 0$

Construct test case for each case of the specification

Functional Testing: Discussion

- **Focuses in input/output behavior**
- **Attemps to find**
  - incorrect or missing functionalities
  - interface errors
- **Limitations:**
  - Not effective for finding coding errors, e.g., buffer overflow
  - Does not reveal errors in specification, e.g., missing use case

Approach 4: Structural Testing

Use knowledge about code:

```
function roots(a, b, c /*doubles*/) {
  var q = b * b - 4 * a * c;
  if (q > 0 && a != 0) {
    ...                    ← Test this case
  } else if (q == 0) {
    ...                    ← Test this case
  } else {
    ...                    ← Test this case
  }
}
```

Approach 4: Structural Testing

Use knowledge about code:

```
function roots(a, b, c /*doubles*/) {
  var q = b * b - 4 * a * c;
  if (q > 0 && a != 0) {
    ...
  } else if (q == 0) {
    ...
  } else {
    ...
  }
}
```

Test this case

Test this case

Test this case

May still miss error, e.g., when testing this case with `a==1`, `b==2`, `c==1`.

Structural Testing: Discussion

- **Goal: Cover all code**
- **Not well suited for system testing**
  - Does not detect missing functionality
  - Requires knowledge about code, which external testers and clients may not have (and do not care about)
  - Covering all code leads to highly redundant tests

Testing Approaches: Summary

Exhaustive testing

- Goal: Cover all input
- Not feasible for real-world programs

Random testing

- Goal: Automation
- Black-box testing
- Suitable for all levels

Functional testing

- Goal: Cover all requirements
- Black-box testing
- Suitable for all levels

Structural testing

- Goal: Cover all code
- White-box testing
- Suitable for unit testing

Testing Approaches: Summary

Exhaustive testing

- Goal: Cover all input
- Not feasible for real-world programs

Random testing

- Goal: Automation
- Black-box testing
- Suitable for all levels

Focus of rest of today's lecture

Functional testing

- Goal: Cover all requirements
- Black-box testing
- Suitable for all levels

Structural testing

- Goal: Cover all code
- White-box testing
- Suitable for unit testing

Outline

- **Overview**
- **Control flow testing**
  - Statement coverage
  - Branch coverage
  - Path coverage
  - Loop coverage
- **Data flow testing**
  - DU-pair coverage
- **Interpretation of coverage**

Motivating Example

Given an array of numbers, sort the array in-place in ascending order.

```
function sort(a) {
  if (!a || a.length < 2)
    return; // array is trivially sorted
  for (var i = 0; i < a.length - 1; i++) {
    if (a[i] < a[i+1]) {
      break;
    }
  }
  if (i >= a.length - 1)
    return; // array is sorted
  qsort(a); // use quicksort to sort array
}
```

Motivating Example

Given an array of numbers, sort the array in-place in ascending order.

```
function sort(a) {
  if (!a || a.length < 2)
    return; // array is trivially sorted
  for (var i = 0; i < a.length - 1; i++) {
    if (a[i] < a[i+1]) {
      break;
    }
  }
  if (i >= a.length - 1)
    return; // array is sorted
  qsort(a); // use quicksort to sort array
}
```

Error: Check for sortedness should use >

Example Test Suite

```
sort(null);
sort(undefined);
sort([]);
sort([1]);
sort([1,2,3]);
```

Test invalid input

Test valid input

Example Test Suite

```
sort(null);
sort(undefined);
sort([]);
sort([1]);
sort([1,2,3]);
```

Test invalid input

Test valid input

How effective is this test suite?

Example Test Suite

```
sort(null);
sort(undefined);
sort([]);
sort([1]);
sort([1,2,3]);
```

Test invalid input

Test valid input

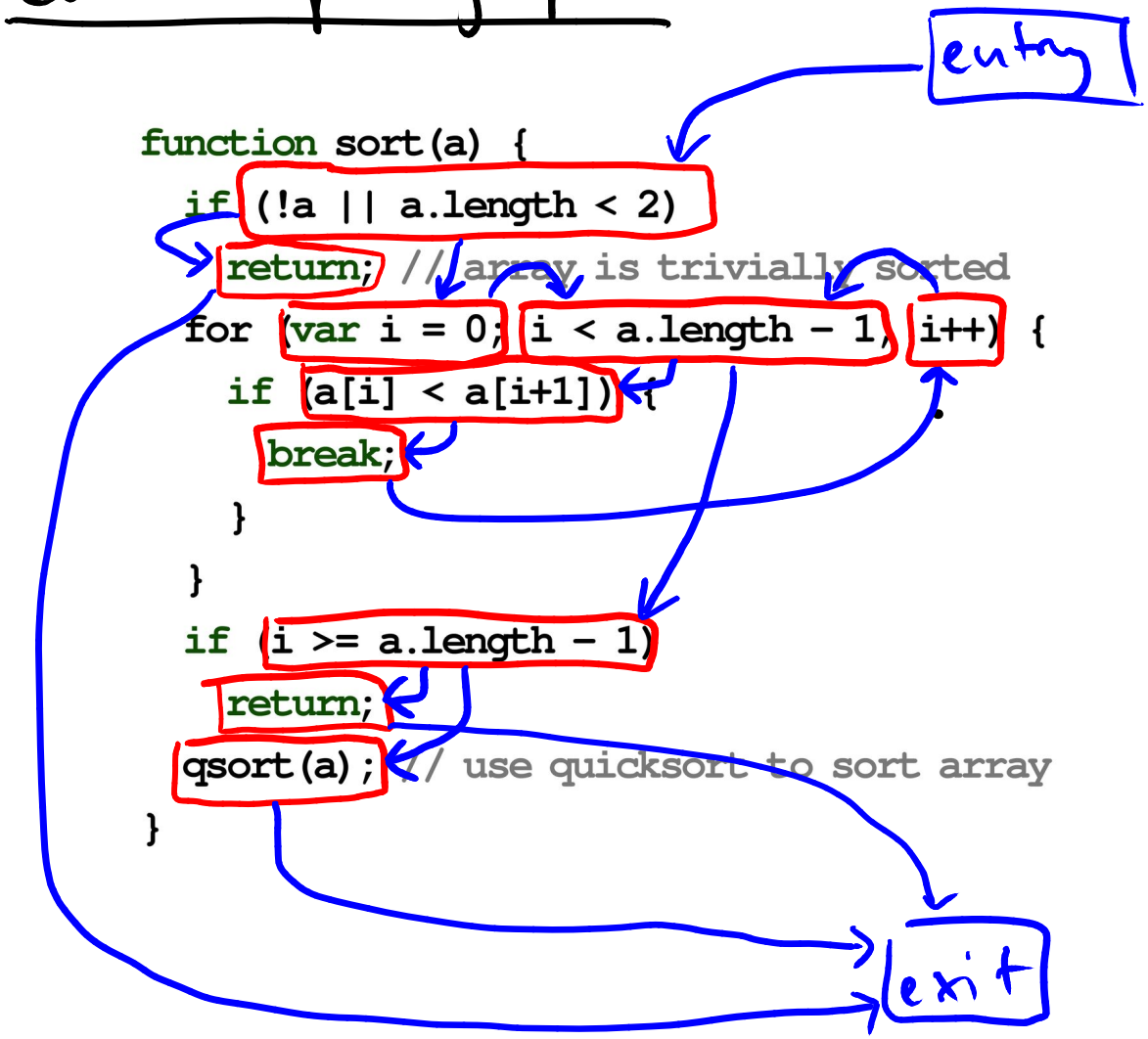
How effective is this test suite?

Coverage: Measure for degree to which code is tested by a test suite

Control Flow Testing

- **Idea: Cover as many different flows of control as possible**
- **Based on control flow graph**
  - Nodes: Basic blocks (sequence of statements that are always executed together) or statements
  - Edges: Flow of control
- **Idea: More parts of CFG executed gives higher chance to uncover bug**

Control flow graph

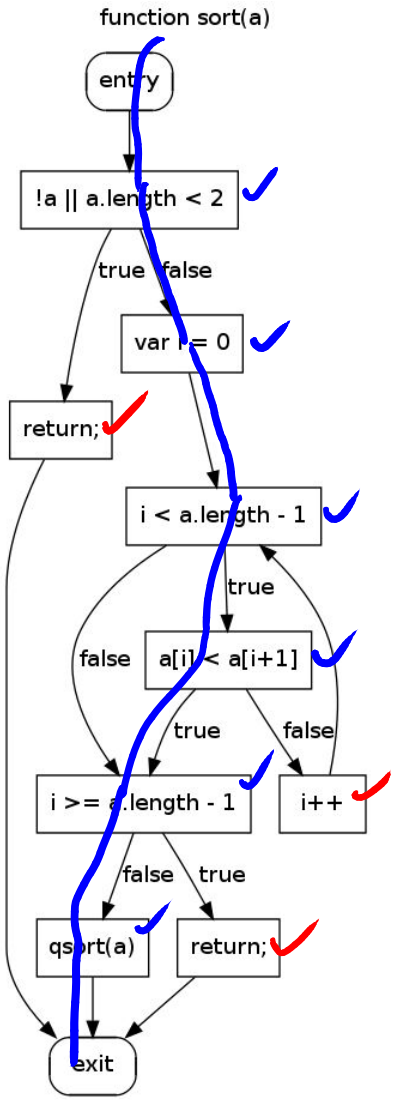


Statement Coverage

Assess effectiveness of test suite by measuring how many statements of the tested program it executes

$$cov_{stmt} = \frac{nb.\ of\ executed\ statements}{total\ nb.\ of\ statements}$$

- Intuition: Can detect bug in a statement only by executing the statement
- Can also be defined on basic blocks



Input: a = [3, 7, 5]

Statement coverage:

6 executed stmts.

3 statements not covered.

→ 67 % statement cov.

Statement Coverage: Discussion

Is full statement coverage enough?

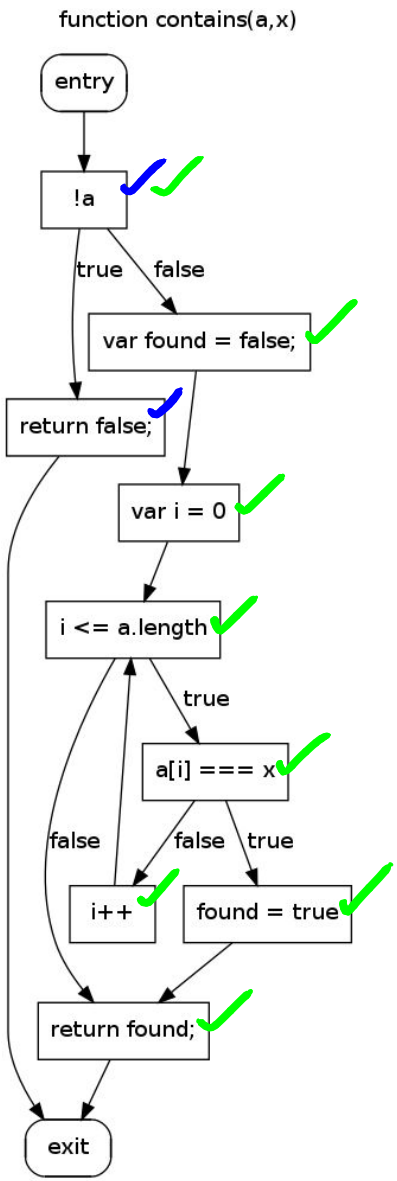
```
function contains(a, x) {
  if (!a) return false;
  var found = false;
  for (var i = 0; i <= a.length; i++) {
    if (a[i] === x) {
      found = true;
      break;
    }
  }
  return found;
}
```

Statement Coverage: Discussion

Is full statement coverage enough?

```
function contains(a, x) {
  if (!a) return false;
  var found = false;
  for (var i = 0; i <= a.length; i++) {
    if (a[i] === x) {
      found = true;
      break;
    }
  }
  return found;
}
```

Should be <, otherwise goes out of bounds.



Two tests:

- a = undefined, x = 23
- a = [1, 2], x = 2

→ 100% statement coverage

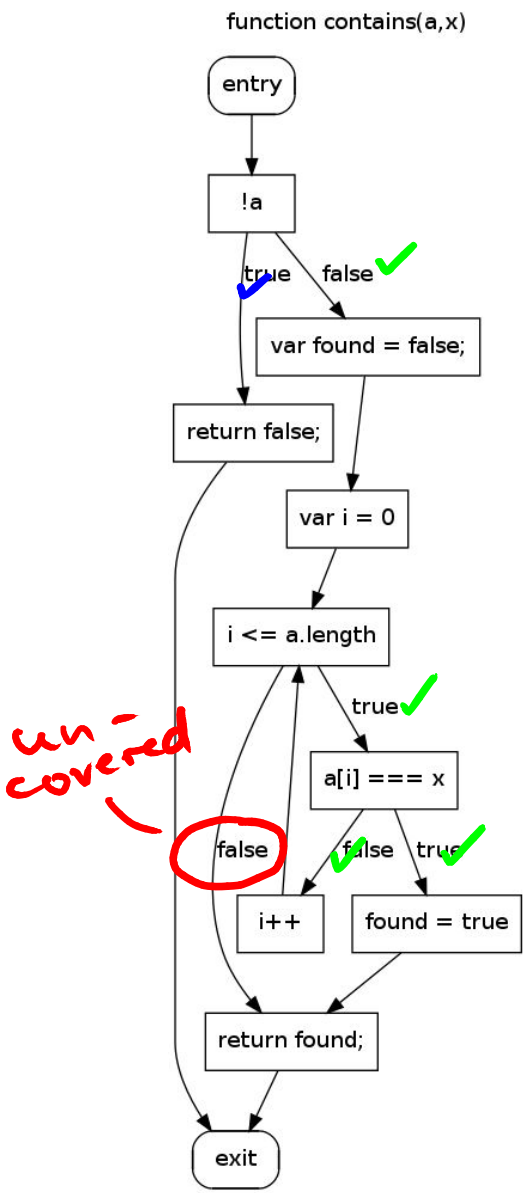
Bug not found!

Branch Coverage

Idea: Test all possible branches in the CFG

$$cov_{branch} = \frac{nb.\ of\ executed\ branches}{total\ nb.\ of\ branches}$$

- An edge $a \xrightarrow{b} c$ in the CFG is a branch iff there is another edge $a \xrightarrow{b'} c'$ with $c \neq c'$
- If code contains no branches, coverage defined as 100%



Example for branch coverage

Tests:

- a = undefined, x = 23
- a = [1, 2], x = 2

Branch coverage:

→ execute 5/6 branches

→ 83% coverage

Branch Coverage: Discussion

- **Leads to more thorough testing than statement coverage**
  - Complete branch coverage implies complete statement coverage
  - But: ”at least n% branch coverage” does not imply ”at least n% statement coverage”
- **Most widely used criterion in industry**