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: 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

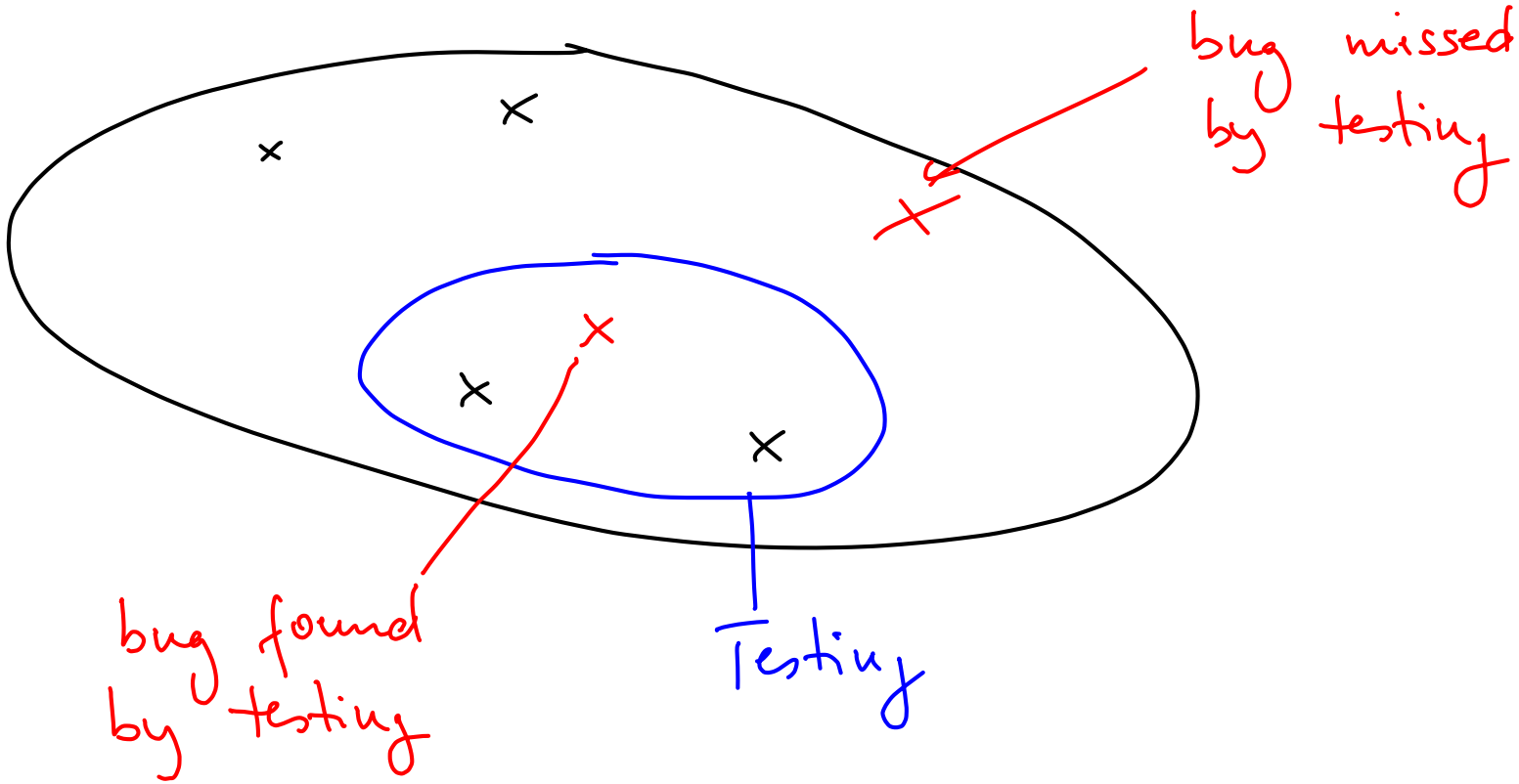
Testing can only show the presence of bugs, never their absence.

E. W. Dijkstra

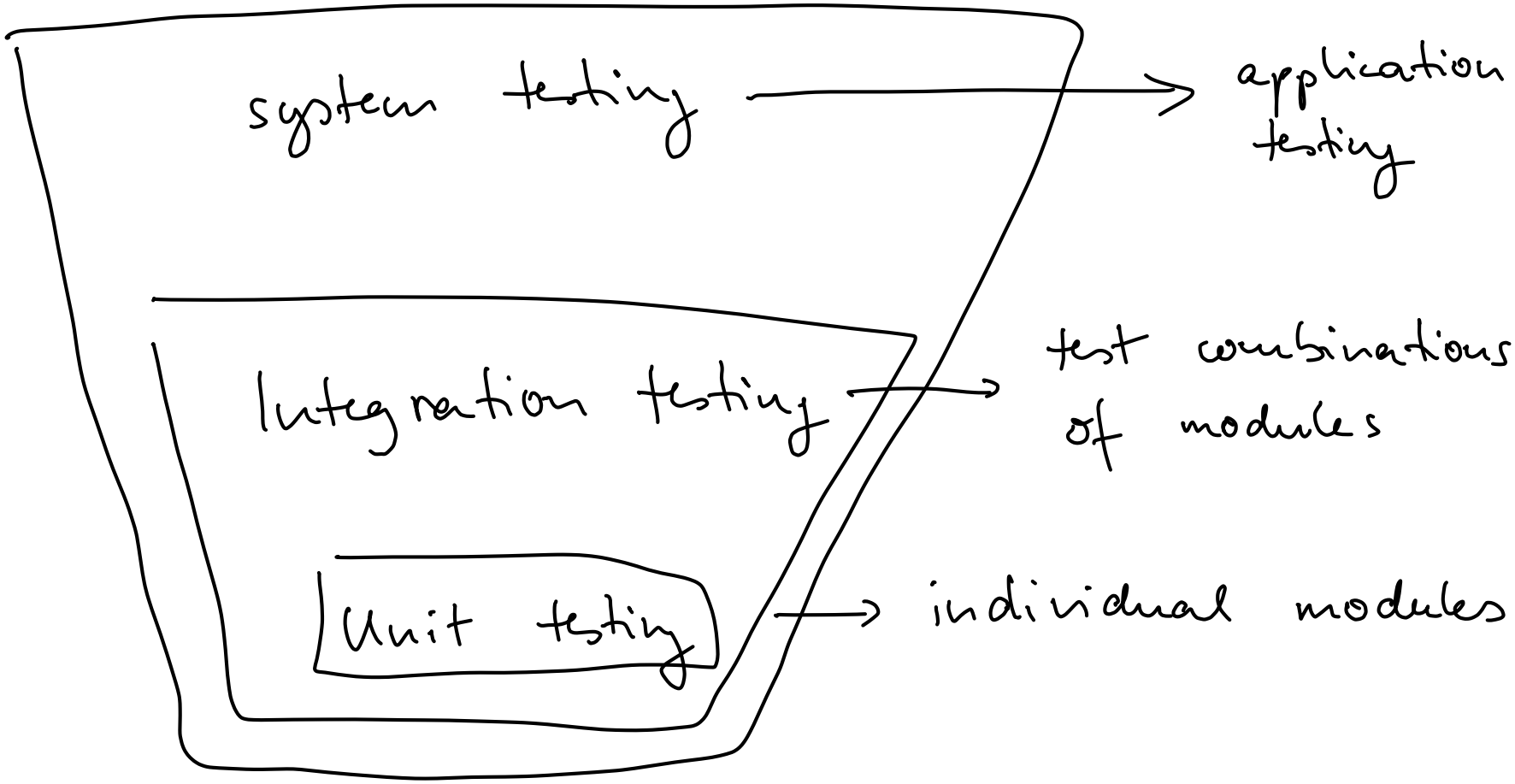
Impossible to completely test any non-trivial module or system

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

P i P(i) .. behavior



Testing Levels



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

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

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

- Assuming that doubles 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;
  // ... (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;
  // ... (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

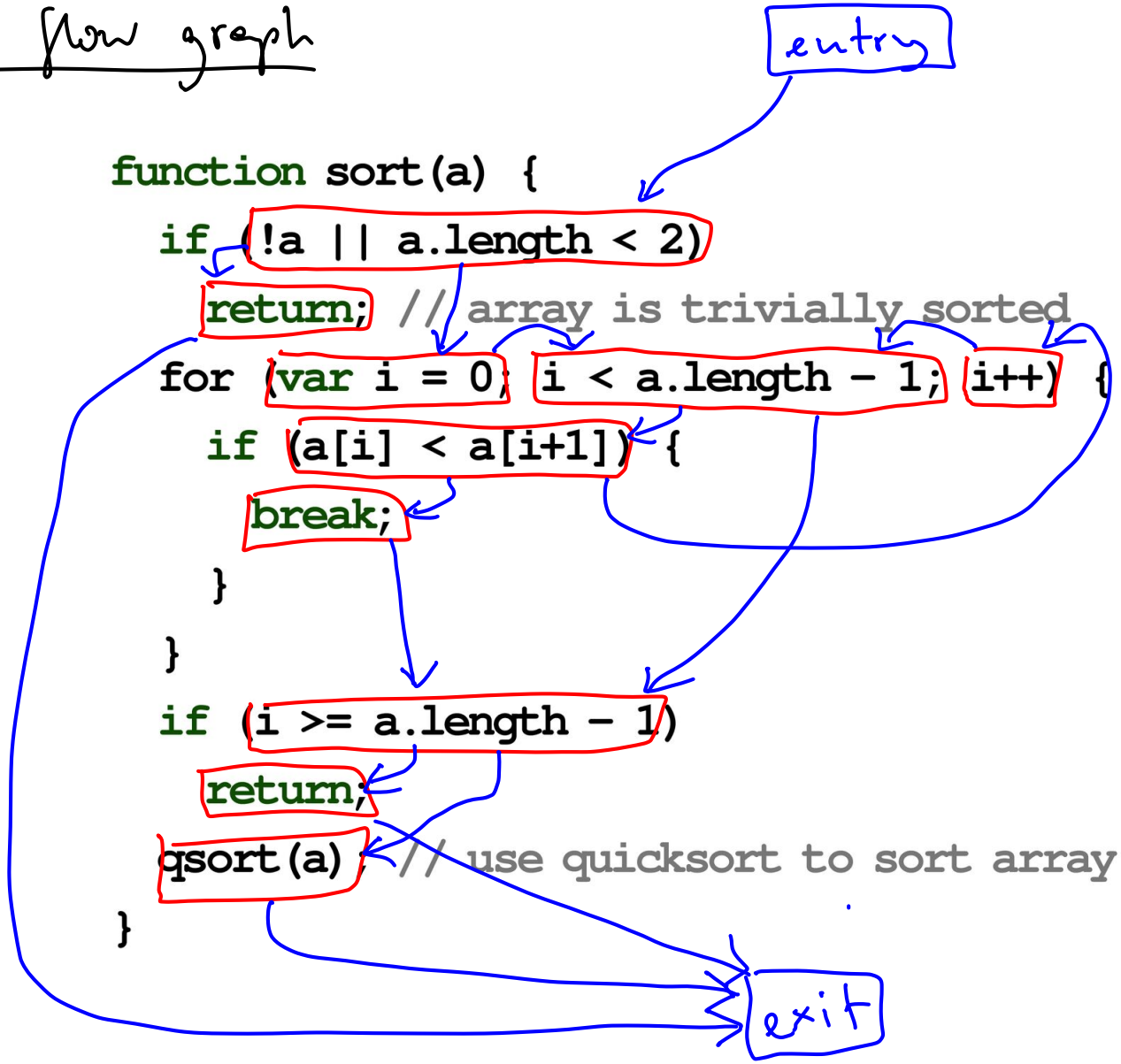
How effective is this test suite?

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

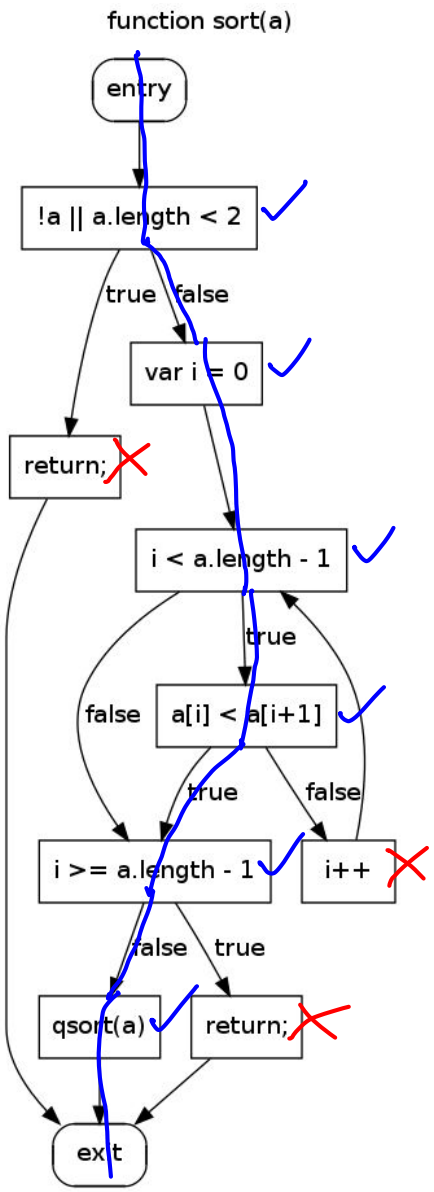


Statement Coverage

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 = [3, 7, 5]

Statement coverage :

6 statements covered

3 statements not covered

→ 67% statement coverage

Statement Coverage: Discussion

Is full statement coverage enough?

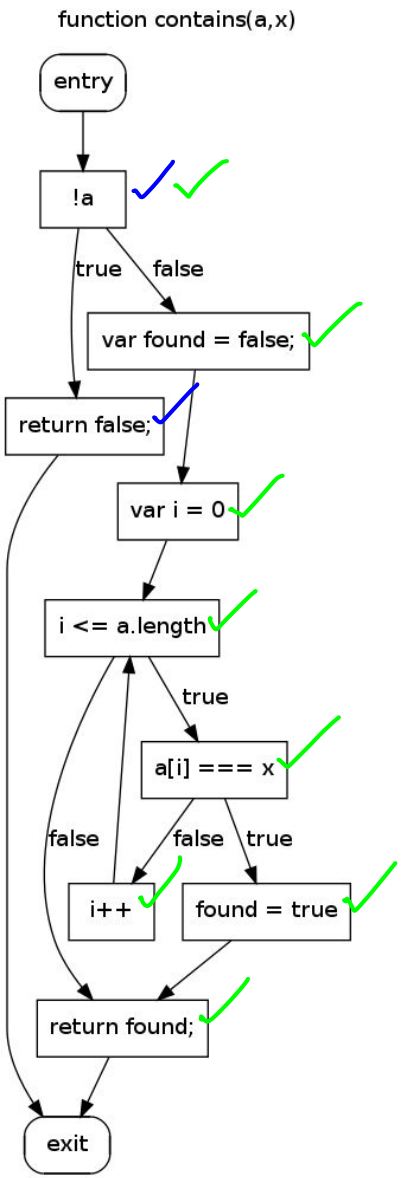
```
function contains(a, x) {
  if (!a) return false;
  var found = false;
  for (var i = 0; i <= a.length; i++) {
    if (a[i] === x) {
      found = true;
      break;
    }
  }
  return found;
}
```

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

But not found!

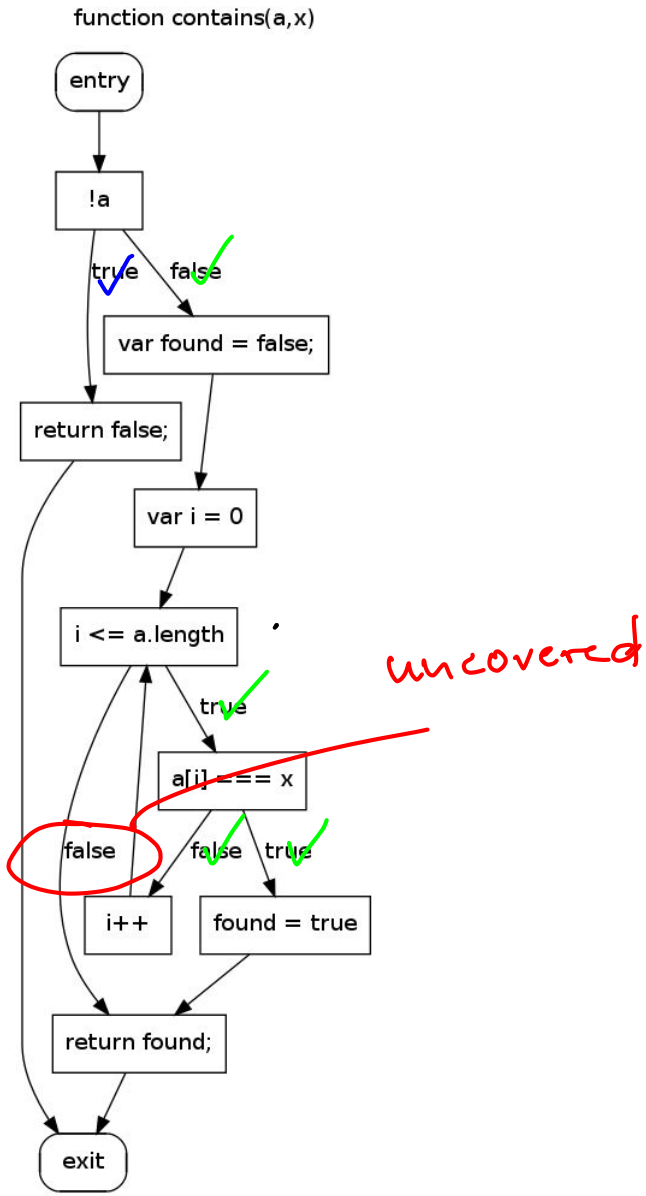
Branch Coverage

Idea: Test all possible branches in the CFG

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

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

Branch coverage example:



uncovered

Tests:

a = undefined, x = 23

a = [1, 2], x = 2

Branch coverage?

→ execute 5/6 branches

→ 83% coverage

Add 3rd test:

a = [1], x = undefined

→ 100% 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**

Branch Coverage: Discussion (2)

Is full branch coverage enough?

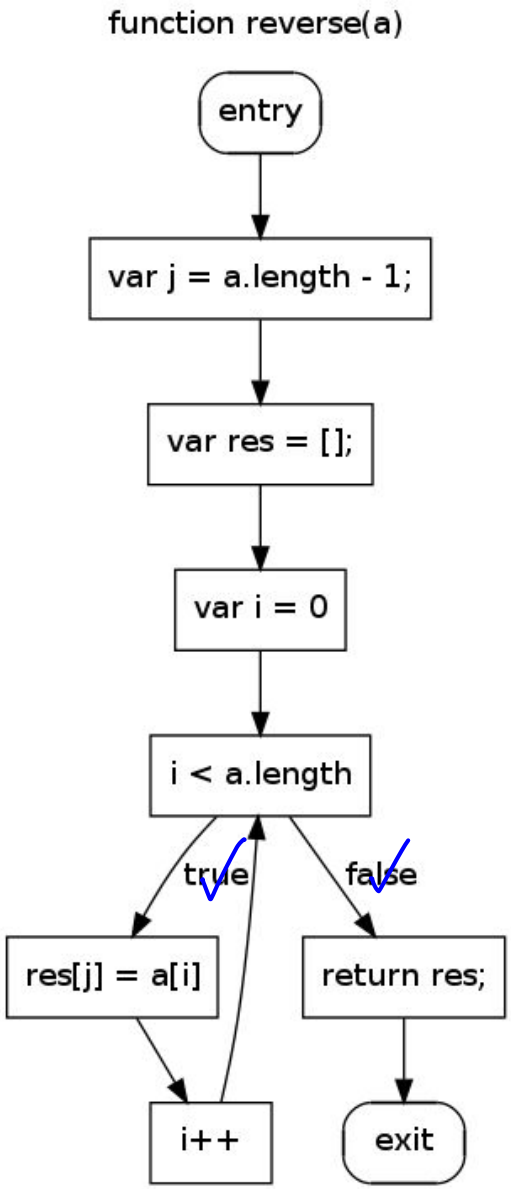
```
function reverse(a) {
  var j = a.length - 1;
  var res = [];
  for (var i = 0; i < a.length; i++) {
    res[j] = a[i];
  }
  return res;
}
```

Branch Coverage: Discussion (2)

Is full branch coverage enough?

```
function reverse(a) {
  var j = a.length - 1;
  var res = [];
  for (var i = 0; i < a.length; i++) {
    res[j] = a[i];
  }
  return res;
}
```

Must decrement `j`



A single test can achieve 100% branch coverage:

a = [1] ✓

But: Bug missed!

Branch Coverage: Discussion (3)

Another example:

```
function fourPaths(a, b) {
  var x = 1;
  var y = 1;
  if (a)
    x = 0;
  else
    y = 0;
  if (b)
    return 5 / x;
  else
    return 5 / y;
}
```

Branch Coverage: Discussion (3)

Another example:

```
function fourPaths(a, b) {
  var x = 1;
  var y = 1;
  if (a)
    x = 0;
  else
    y = 0;
  if (b)
    return 5 / x;
  else
    return 5 / y;
}
```

Division by zero

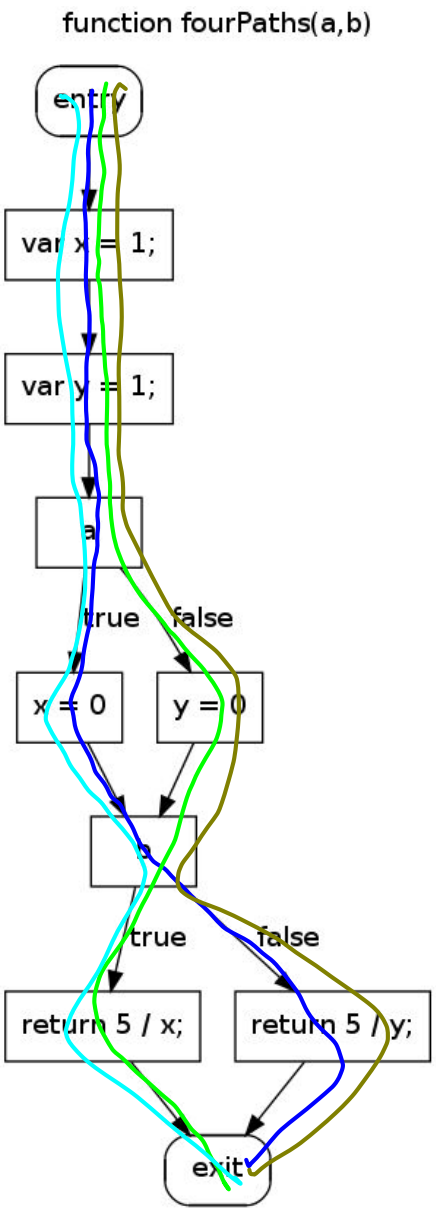
Path Coverage

Idea:

Test all possible paths through the CFG

$$cov_{path} = \frac{Nb.\ of\ executed\ paths}{Total\ nb.\ of\ paths}$$

- A path is a sequence $n_1, .., n_k$ where
  - $n_1 = entry\ node$
  - $n_k = exit\ node$
  - There is an edge (n_i, n_{i+1}, c) for all $1 \leq i < k$



Two tests:

- a = true, b = false ✓
- a = false, b = true ✓

→ 100% branch coverage

Bug missed!

Path coverage?

→ 2/4 = 50%

More tests: a = true, b = true ✓

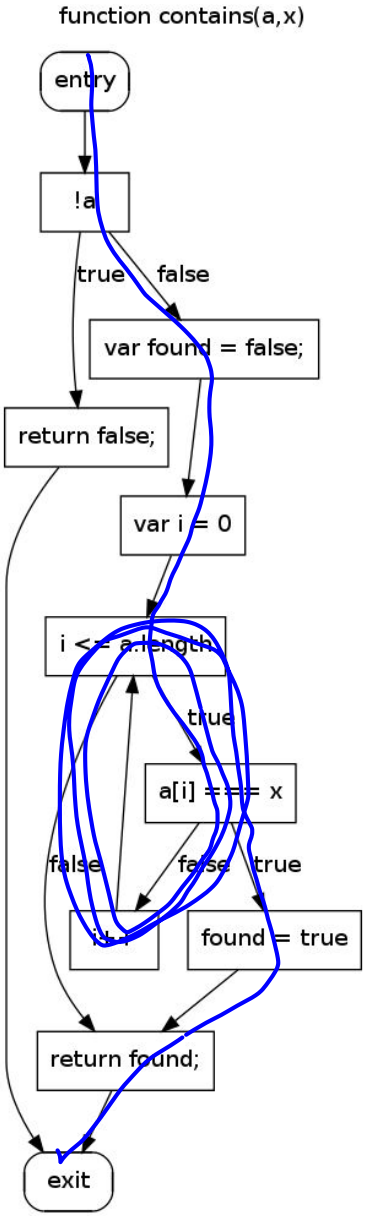
a = false, b = false ✓

→ 100% coverage

Path Coverage: Discussion

Is path coverage the solution?

```
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;
}
```


function contains(a,x)
entry
!a
true
false
var found = false;
return false;
var i = 0
i <= a.length
true
a[i] === x
false
false
true
i++
found = true
return found;
exit

Path Coverage: Discussion (2)

- **Leads to more thorough testing than both statement and branch coverage**
  - Complete path coverage implies complete statement coverage and branch coverage
  - But: ”at least n% path coverage” implies neither ”at least n% statement coverage” nor ”at least n% branch coverage”
- **Complete path coverage is not feasible for loops: Unbounded number of paths**

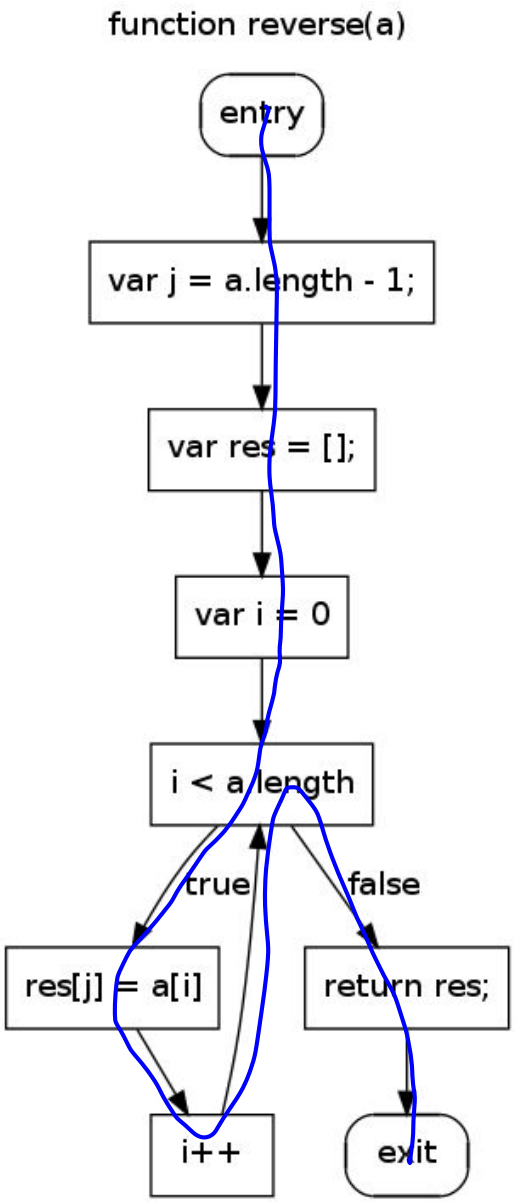
Loop Coverage

Idea: For each loop, test 0, 1, or more than 1 consecutive iterations

Let n_k be the number of loops with exactly k consecutively executed iterations

$$cov_{loop} = \frac{n_0 + n_1 + n_{>1}}{Total\ nb.\ of\ loops * 3}$$

- Typically combined with other criteria, such as statement or branch coverage



Test:

a = [1] ✓

Loop coverage?

→ 1/3 possible cases

33% loop coverage

More tests:

a = [ ]

a = [1, 2]

→ 100% loop coverage

Measuring Coverage

- **Coverage information is collected while the test suite executes**
- **Typically implemented via code instrumentation or debug interface**
  - Count executed basic blocks, branches taken, etc.

Measuring Coverage: Example

```
function fourPaths(a, b) {
  var x = 1;
  var y = 1;
  if (a) {
    x = 0;
  } else {
    y = 0;
  }
  if (b) {
    return 5 / x;
  } else {
    return 5 / y;
  }
}
```

Measuring Coverage: Example

```
function fourPaths(a, b) {
  var x = 1;
  var y = 1;
  if (a) {
    executedBranches[0]++; x = 0;
  } else {
    executedBranches[1]++;  y = 0;
  }
  if (b) {
    executedBranches[2]++; return 5 / x;
  } else {
    executedBranches[3]++; return 5 / y;
  }
}
```

Coverage measurement described with operational semantics

- extend configuration : $\langle P, s \rangle \rightsquigarrow \langle P, s, b \rangle$

  branch cov. map : $Loc \to \{T, F\}$

- extend transition rules :
  assume that each command is annotated with
  its code location : e.g. , $x := 5^{\text{line } 23}$

$$\frac{}{\langle \text{if True then } c_1^{\ell_1} \text{ else } c_2^{\ell_2}, s, b \rangle \to \langle c_1, s, b[\ell_1 \mapsto T] \rangle} \; (\text{if}_T)$$

Outline

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