# ICS 52: Introduction to Software Engineering

Fall Quarter 2004

Professor Richard N. Taylor

Lecture Notes:  Testing

http://www.ics.uci.edu/~taylor/ICS_52_FQ04/syllabus.html



vine

# Two Approaches

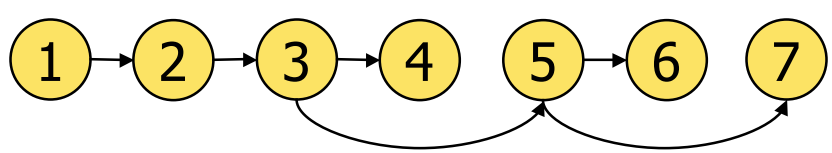
- White box testing
  - <u>Structural</u> testing
  - Test cases designed, selected, and ran based on structure of the source code
  - Scale: tests the nitty-gritty
  - Drawbacks: need access to source; thought-process driven by what you've got
- Black box testing
  - <u>Specification-based</u> testing
  - Test cases designed, selected, and ran based on specifications
  - Scale: tests the overall system behavior
  - Drawback: less thorough?  Thought-process limited to what you're supposed to have.

# Structural Testing

- Use source code to derive test cases
  - Build a graph model of the system
    - » Control flow
    - » Data flow
  - State test cases in terms of graph coverage
- Choose test cases that guarantee different types of coverage
  - Node coverage
  - Edge coverage
  - Loop coverage
  - Condition coverage
  - Path coverage

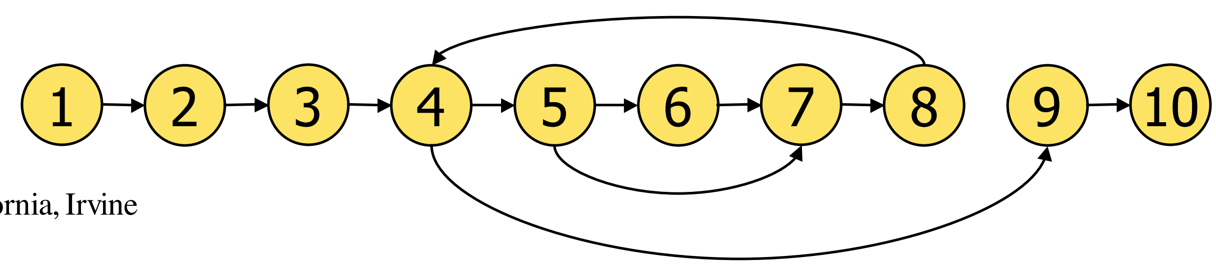
# Example

```
Node getSecondElement() {
    Node head = getHead();
    if (head == null)
        return null;
    if (head.next == null)
        return null;
    return head.next.node;
}
```



# Example

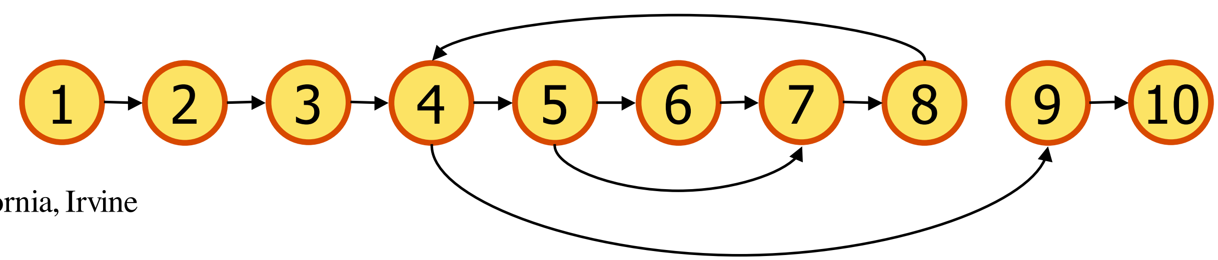
```
float homeworkAverage(float[] scores) {
    float min = 99999;
    float total = 0;
    for (int i = 0 ; i < scores.length ; i++) {
        if (scores[i] < min)
            min = scores[i];
        total += scores[i];
    }
    total = total – min;
    return total / (scores.length – 1);
}
```



# Node Coverage

- Select test cases such that every <u>node</u> in the graph is visited
  - Also called statement coverage
    - » Guarantees that every statement in the source code is executed at least once
- Selects minimal number of test cases

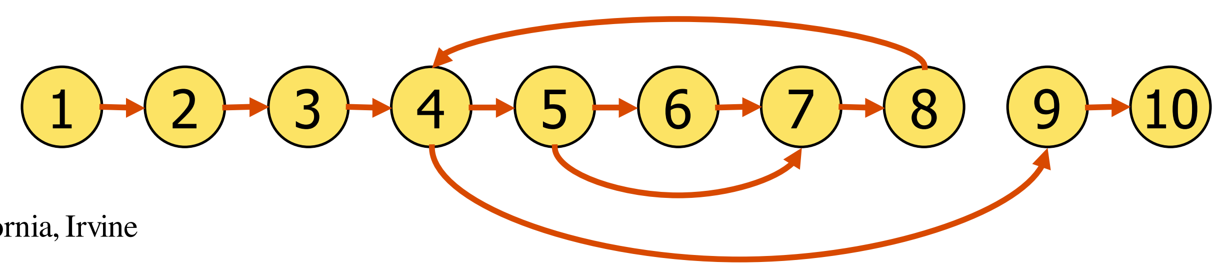
Test case: { 2 }



# Edge Coverage

- Select test cases such that every edge in the graph is visited
  - Also called branch coverage
    - » Guarantees that every branch in the source code is executed at least once
- More thorough than node coverage
  - More likely to reveal logical errors

Test case: { 1, 2 }



# Other Coverage Criteria

- Loop coverage
  - Select test cases such that every loop *boundary* and *interior* is tested
    - » Boundary: 0 iterations
    - » Interior: 1 iteration <u>and</u> > 1 iterations
  - Watch out for nested loops
  - Less precise than edge coverage
- Condition coverage
  - Select test cases such that all conditions are tested
    - » if (a > b || c > d) …
  - More precise than edge coverage

# Other Coverage Criteria

- Path coverage
  - Select test cases such that every <u>path</u> in the graph is visited
  - Loops are a problem
    - » 0, 1, average, max iterations
- Most thorough…
- …but is it feasible?

# Challenges

- Structural testing can cover all nodes or edges without revealing obvious faults
  - No matter what input, program always returns 0
- Some nodes, edges, or loop combinations may be infeasible
  - Unreachable/unexecutable code
- “Thoroughness”
  - A test suite that guarantees edge coverage also guarantees node coverage…
  - …but it may not find as many faults as a different test suite that only guarantees node coverage

# More Challenges

- Interactive programs
- Listeners or event-driven programs
- Concurrent programs
- Exceptions
- Self-modifying programs
- Mobile code
- Constructors/destructors
- Garbage collection

# Specification-Based Testing

- Use specifications to derive test cases
  - Requirements
  - Design
  - Function signature
- Based on some kind of input domain
- Choose test cases that guarantee a wide range of coverage
  - Typical values
  - Boundary values
  - Special cases
  - Invalid input values

# “Some Kind of Input Domain”

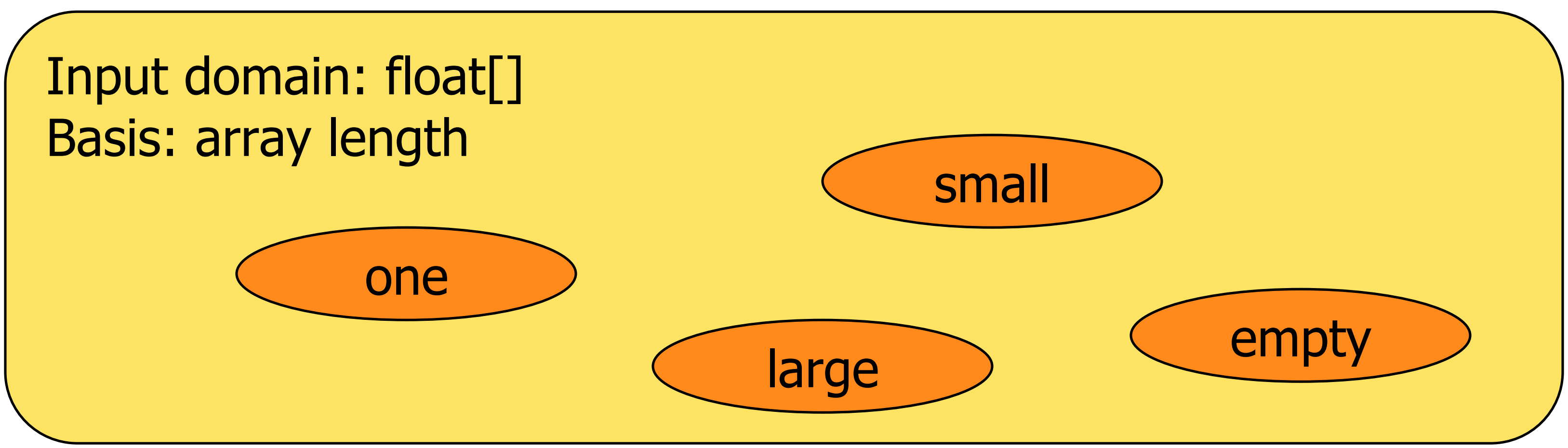
- Determine a basis for dividing the input domain into subdomains
    - » Subdomains may overlap
  - – Possible bases
    - » Size
    - » Order
    - » Structure
    - » Correctness
    - » Your creative thinking
- Select test cases from each subdomain
  - – One test case may suffice

# Example

```
float homeworkAverage(float[] scores) {
    float min = 99999;
    float total = 0;
    for (int i = 0 ; i < scores.length ; i++) {
        if (scores[i] < min)
            min = scores[i];
        total += scores[i];
    }
    total = total – min;
    return total / (scores.length – 1);
}
```

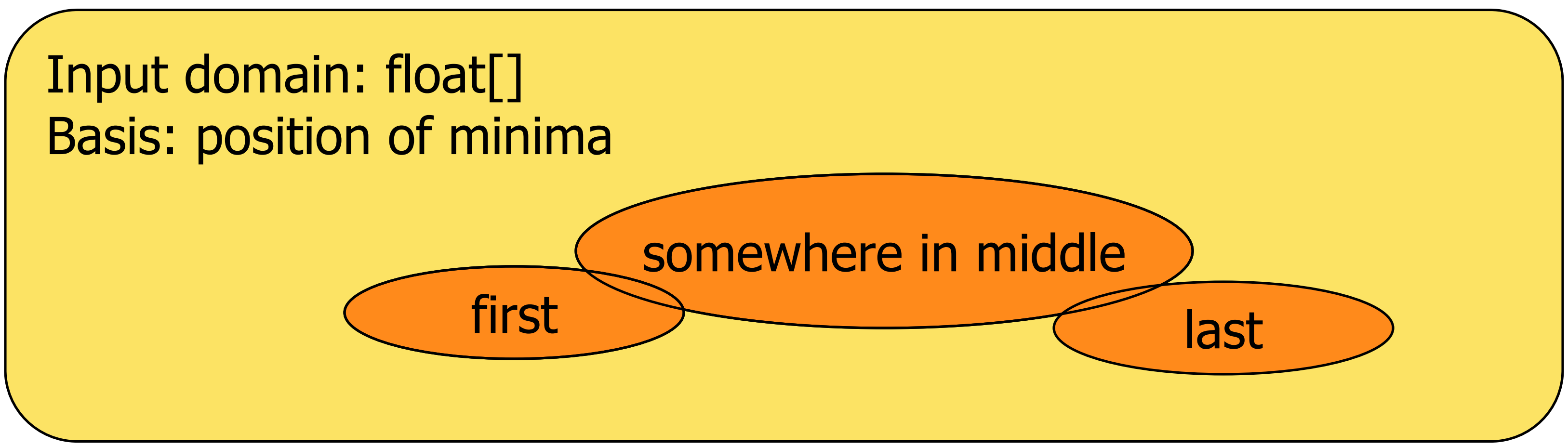
# Possible Bases

- Array length
  - Empty array
  - One element
  - Two or three elements
  - Lots of elements



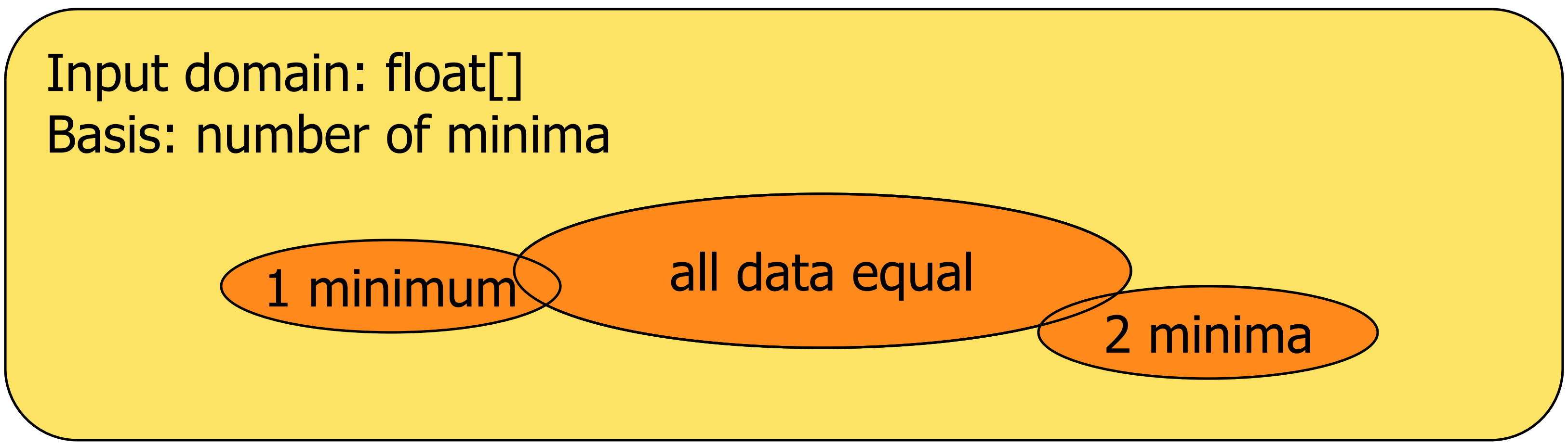
# Possible Bases

- Position of minimum score
  - Smallest element first
  - Smallest element in middle
  - Smallest element last



# Possible Bases

- Number of minima
  - Unique minimum
  - A few minima
  - All minima



# Testing Matrix

| Test case (input) | Basis (subdomain) | Expected output | Notes |
|---|---|---|---|
| | | | |

# homeworkAverage 1

| Test case (input) | Basis: Array length | | | | Expected output | Notes |
|---|---|---|---|---|---|---|
| | Empty | One | Small | Large | | |
| () | x | | | | 0.0 | |
| (87.3) | | x | | | 87.3 | crashes! |
| (90,95,85) | | | x | | 92.5 | |
| (80,81,82,83, 84,85,86,87, 88,89,90,91) | | | | x | 86.0 | |

# homeworkAverage 2

| Test case (input) | Basis: Position of minimum | | | Expected output | Notes |
|---|---|---|---|---|---|
| | First | Middle | Last | | |
| (80,87,88,89) | x | | | 88.0 | |
| (87,88,80,89) | | x | | 88.0 | |
| (99,98,0,97,96) | | x | | 97.5 | |
| (87,88,89,80) | | | x | 88.0 | |

# homeworkAverage 3

| Test case (input) | Basis: Number of minima | | | Expected output | Notes |
|---|---|---|---|---|---|
| | One | Several | All | | |
| (80,87,88,89) | x | | | 88.0 | |
| (87,86,86,88) | | x | | 87.0 | |
| (99,98,0,97,0) | | x | | 73.5 | |
| (88,88,88,88) | | | x | 88.0 | |