The BUGS Lecture II
Avoiding, Finding and Eliminating Bugs
Saman Amarasinghe
Spring 2014
Boundary Testing

/* @param a  
 * @return |a|  
 */
    
    public static int abs(int a) {
        ...
    }

Tests for abs

➤ what are some values or ranges of x that might be worth probing?
  • x < 0 (flips sign) or x ≥ 0 (returns unchanged)
  • around x = 0 (boundary condition)
  • Specific tests: say x = -1, 0, 1

How about...

    int x = -2147483648; // this is Integer.MIN_VALUE
    System.out.println(x<0); // true
    System.out.println(Math.abs(x)<0); // also true!

From Javadoc for Math.abs:

➤ Note that if the argument is equal to the value of Integer.MIN_VALUE, the most negative representable int value, the result is that same value, which is negative

© Robert Miller 2009
In-class Exercise

/** @param p is a array of integers
 * @return the mean value of the integers as an integer
 */

public static int mean(int[] p) {
    ...
}

© Robert Miller, Saman Amarasinghe, Max Goldman 2009-2014
What to test in Mean

**Simple correct**
- One element list
- More than one element list

**Boundary cases**
- Zero element list

**Integer conversion**
- Average of a list of same numbers
  - Average > xxx.5
  - Average < xxx.5
  - Average == xxx.5

**Overflow**
- Average of maxint-4, maxint-5, minint+6, minint+7
COVERAGE
public class Coordinate {
    private int x;
    private int y;
    Coordinate(int x, int y) {
        this.x = x;
        this.y = y;
    }

    int diffX(Coordinate a) {
        return (a.x > x)?(a.x - x):(x - a.x);
    }

    int diffY(Coordinate a) {
        return (a.y > y)?(a.y - y):(y - a.y);
    }

    int lengthSq(Coordinate a) {
        return diffX(a)*diffX(a) + diffY(a)*diffY(a);
    }
}
Coverage

how good are my tests?

➢ measure extent to which tests ‘cover’ the specification or code

➢ What coverage do you get?

Coordinate c1(1, 1);
Coordinate c2(2, 2);
c1.lengthSq(c2);
public class Coordinate {
    private int x;
    private int y;
    Coordinate(int x, int y) {
        this.x = x;
        this.y = y;
    }
    int diffX(Coordinate a) {
        return (a.x > x)?(a.x - x):(x - a.x);
    }
    int diffY(Coordinate a) {
        return (a.y > y)?(a.y - y):(y - a.y);
    }
    int lengthSq(Coordinate a) {
        return diffX(a)*diffX(a) + diffY(a)*diffY(a);
    }
}

Coordinate c1(1, 1);
Coordinate c2(2, 2);
c1.lengthSq(c2);
Coordinate c1(1, 1);
Coordinate c2(2, 2);
c1.lengthSq(c2);
Coverage

how good are my tests?

- measure extent to which tests ‘cover’ the specification or code

- What coverage do you get?

Coordinate c1(1, 1);
Coordinate c2(2, 2);
c1.lengthSq(c2);
c2.lengthSq(c1);
Coordinate c1(1, 1);
Coordinate c2(2, 2);
c1.lengthSq(c2);  c1.lengthSq(c2);
how good are my tests?

- measure extent to which tests ‘cover’ the specification or code

- What coverage do you get?

Coordinate c1(1, 1);
Coordinate c2(2, 2);
Coordinate c3(0, 3);
c1.lengthSq(c2);
c2.lengthSq(c1);
c1.lengthSq(c3);
c3.lengthSq(c1);
Coordinate c1(1, 1);
Coordinate c2(2, 2);
Coordinate c3(0, 3);

lengthSq

diffX

diffX

diffY

diffY

\[ c_1.lengthSq(c_2); \]
\[ c_1.lengthSq(c_3); \]
\[ c_3.lengthSq(c_1); \]
Coverage

how good are my tests?

- measure extent to which tests ‘cover’ the specification or code

- What coverage do you get?
  - All Statements
  - All Branches
  - All Paths
How Far Should You Go?

for code coverage
- all-statements: should be the goal
- all-branches: if possible
- all-paths: infeasible

industry practice
- all-statements is common goal, rarely achieved (due to unreachable code)
- safety critical industry has more arduous criteria (eg, “MCDC”, modified decision/condition coverage)
A Typical Statement Coverage Tool

- EclEmma Eclipse plugin

Coverage statistics for packages and classes.
Black Box vs. Glass Box Testing

**black box testing**
- choosing test data only from spec, without looking at implementation

**glass box (white box) testing**
- choosing test data with knowledge of implementation
  - e.g. if implementation does caching, then should test repeated inputs
  - if implementation selects different algorithms depending on the input, should choose inputs that exercise all the algorithms
- must take care that tests don’t depend on implementation details
  - e.g. if spec says “throws Exception if the input is poorly formatted”, your test shouldn’t check specifically for a NullPtrException just because that’s what the current implementation does
- good tests should be **modular** -- depending only on the spec, not on the implementation

© Robert Miller 2009
Black Box vs. Glass Box Testing

**best practice**

- generate black-box test cases until code coverage is sufficient

![Diagram](attachment:diagram.png)
Testing Framework

**driver**
- just runs the tests
- must design unit to be drivable!
- eg: program with GUI should have API

**stub**
- replaces other system components
- allows reproducible behaviours (esp. failures)

**oracle**
- determines if result meets spec
- preferably automatic and fast
- varieties: computable predicate (e.g. is the result odd?), comparison with literal (e.g. must be 5), manual examination (by a human)
- in regression testing, can use previous results as “gold standard”
Regression Testing

whenever you find and fix a bug

➢ store the input that elicited the bug
➢ store the correct output
➢ add it to your test suite

why regression tests help

➢ helps to populate test suite with good test cases
  • remember that a test is good if it elicits a bug – and every regression test did in one version of your code
➢ protects against reversions that reintroduce bug
➢ the bug may be an easy error to make (since it happened once already)

test-first debugging

➢ when a bug arises, immediately write a test case for it that elicits it
➢ once you find and fix the bug, the test case will pass, and you’ll be done
DEBUGGING
How to Debug

1) reproduce the bug with a small test case
   - find a small, repeatable test case that produces the failure (may take effort, but helps clarify the bug, and also gives you something for regression)
   - don't move on to next step until you have a repeatable test

2) find the cause
   - narrow down location and proximate cause
   - study the data / hypothesize / experiment / repeat
   - may change code to get more information
   - don't move on to next step until you understand the cause

3) fix the bug
   - is it a simple typo, or is it a design flaw? does it occur elsewhere?

4) add test case to regression tests
   - then run regression tests to ensure that the bug appears to be fixed, and no new bugs have been introduced by the fix
Reducing to a Simple Test Case

find simplest input that will provoke bug
- usually not the input that originally revealed existence of the bug
- start with data that revealed bug
- keep paring it down (binary search can help)
- often leads directly to an understanding of the cause

same idea is useful at many levels of a system
- method arguments
- input files
- keystrokes and mouse clicks in a GUI
Example

```java
/**
 * @param s
 * @param t
 * @return true if and only if s contains t as a substring, e.g.
 *         contains("hello world", "world") == true.
 */

public static boolean contains(String s, String t) { ... }
```

- A user discovers that
  
  contains("Life is wonderful! I am so very very happy all the time“, 
  "very happy")
  
  incorrectly returns false

**Wrong approach:**

- Try to trace the execution of contains() for this test case

**Right approach:**

- First try to reduce the size of the test case
- Even better: bracket the bug with a test case that fails and similar test cases that succeed

© Robert Miller 2009
Code for contains()

/**
 * @param s
 * @param t
 * @return true if and only if s contains t as a substring, e.g.
 *         contains("hello world", "world") == true.
 */

public static boolean contains(String s, String t) {
    search:
    for (int i = 0; i < s.length(); ++i) {
        for (int j = 0; j < t.length(); ++j, ++i) {
            if (s.charAt(i) != t.charAt(j))
                continue search;
        }
        return true;
    }
    return false;
}
Finding the Cause

exploit modularity
- start with everything, take away pieces until bug goes
- start with nothing, add pieces back in until bug appears

take advantage of modular reasoning
- trace through program, viewing intermediate results
- insert assertions targeted at the bug
- design all data structures to be printable (i.e., implement toString())
  - Java arrays are not printable; Java collections (e.g. ArrayList) are
- println is a surprisingly useful and universal tool
  - in large systems, use a logging infrastructure instead of println

use binary search to speed things up
- bug happens somewhere between first and last statement
- so do binary search on the ordered set of statements
Example: Finding a Web Browser Bug

suppose a web browser displays the wrong output

URL
Input Stream

Web browser

Welcome to MT!

Input Stream Reader

HTML parser

HTML renderer

This diagram shows data flow through the system, which is often useful for thinking about bugs.
THE BAD AND THE UGLY
The Ugliest Bugs

we’ve had it easy so far
- sequential, deterministic programs have repeatable bugs

but the real world is not that nice…
- timing dependencies
- unpredictable network delays
- varying processor loads
- concurrent programming with threads

heisenbugs
- nondeterministic, hard to reproduce
- may even disappear when you try to look at it with println or debugger!

one approach
- build a lightweight event log (circular buffer)
- log events during execution of program as it runs at speed
- when you detect the error, stop program and examine logs
TEST FIRST PROGRAMMING
Test-First Development

write tests before coding

- specifically, for every method or class:
  1) write specification
  2) write test cases that cover the spec
  3) implement the method or class
  4) once the tests pass (and code coverage is sufficient), you’re done

writing tests first is a good way to understand the spec

- think about partitioning and boundary cases
- if the spec is confusing, write more tests
- spec can be buggy too
  - incorrect, incomplete, ambiguous, missing corner cases
  - trying to write tests can uncover these problems
Example: Binary Search

Binary search is a very simple algorithm that is notoriously hard to implement correctly, even in pseudocode.

“While the first binary search was published in 1946, the first binary search that works correctly for all values of n did not appear until 1962.”

--- Jon Bentley

/**
* Find the first occurrence of x in sorted array a.
* @param x value to find
* @param a array sorted in increasing order
*        (a[0] <= a[1] <= ... <= a[n-1])
* @return lowest i such that a[i]==x, or -1 if x not found in a.
*/
public static int find(int x, int[] a) { ... }
/**
 * Find the first occurrence of x in sorted array a.
 *
 * @param x value to find
 * @param a array sorted in increasing order (a[0] <= a[1] <= ... <= a[n-1])
 * @return lowest i such that a[i]==x, or -1 if x not found in a.
 */

public static int find(int x, int[] a) {
    for (int i = 0; i < a.length; ++i) {
        if (x == a[i]) {
            return i;
        }
    }
    return -1;
}
public class BinarySearchTest {

    @Test
    public void findEmptyListTest() {
        int[] a = { };        
        assertEquals(-1, BinarySearch.find(2, a));
    }

    @Test
    public void findOneElemTest() {
        int[] a = { 23 };        
        assertEquals(-1, BinarySearch.find(20, a));
        assertEquals(0, BinarySearch.find(23, a));
        assertEquals(-1, BinarySearch.find(33, a));
    }

    @Test
    public void findMultiElemTest() {
        int[] a = { -1, 10, 23, 50 };        
        assertEquals(-1, BinarySearch.find(-12, a));
        assertEquals(0, BinarySearch.find(-1, a));
        assertEquals(-1, BinarySearch.find(6, a));
        assertEquals(1, BinarySearch.find(10, a));
        assertEquals(-1, BinarySearch.find(22, a));
        assertEquals(2, BinarySearch.find(23, a));
        assertEquals(-1, BinarySearch.find(24, a));
        assertEquals(3, BinarySearch.find(50, a));
        assertEquals(-1, BinarySearch.find(100, a));
    }

    @Test
    public void findRepeatedTest() {
        int[] a1 = { 1, 1, 1, 1, 1 };        
        int[] a2 = { 1, 2, 3, 3, 3 };        
        assertEquals(0, BinarySearch.find(1, a1));
        assertEquals(0, BinarySearch.find(1, a2));
        assertEquals(1, BinarySearch.find(2, a2));
        assertEquals(-1, BinarySearch.find(3, a2));
    }
}
/**
 * Find the first occurrence of x in sorted array a.
 *
 * @param x
 *   value to find
 * @param a
 *   array sorted in increasing order (a[0] <= a[1] <= ... <= a[n-1])
 * @return
 *   lowest i such that a[i]==x, or -1 if x not found in a.
 */
public static int find(int x, int[] a) {
    return binarySearchInRange(x, a, 0, a.length-1);
}

/**
 * Find the first occurrence of x in sorted array a[first..last].
 *
 * @param x
 *   value to find
 * @param a
 *   array sorted in increasing order (a[0] <= a[1] <= ... <= a[n-1])
 * @param first
 *   first index of range. Requires 0 <= first <= a.length.
 * @param last
 *   last index of range. Requires 0 <= last <= a.length, and first <= last.
 * @return lowest i such that first<=i<=last and a[i]==x, or -1 if there's no such i.
 */
private static int binarySearchInRange(int x, int[] a, int first, int last) {
    int mid = (first + last) / 2;
    if (x < a[mid]) {
        return binarySearchInRange(x, a, first, mid);
    } else if (x > a[mid]) {
        return binarySearchInRange(x, a, mid + 1, last);
    } else {
        // x == a[mid]... we found it!
        return mid;
    }
}
/**
 * Find the first occurrence of x in sorted array a.
 *
 * @param x   value to find
 * @param a   array sorted in increasing order (a[0] <= a[1] <= ... <= a[n-1])
 * @return   lowest i such that a[i]==x, or -1 if x not found in a.
 */
public static int find(int x, int[] a) {
    return binarySearchInRange(x, a, 0, a.length-1);
}

/**
 * Find the first occurrence of x in sorted array a[first..last].
 *
 * @param x   value to find
 * @param a   array sorted in increasing order (a[0] <= a[1] <= ... <= a[n-1])
 * @param first   first index of range. Requires 0 <= first <= a.length.
 * @param last   last index of range. Requires 0 <= last <= a.length, and first <= last.
 * @return lowest i such that first<=i<=last and a[i]==x, or -1 if there's no such i.
 */
private static int binarySearchInRange(int x, int[] a, int first, int max) {
    int mid = (first + max) / 2;
    if (first >= max) {
        return -1; // range has dwindled to nothingness
    }
    if (x < a[mid]) {
        return binarySearchInRange(x, a, first, mid);
    } else if (x > a[mid]) {
        return binarySearchInRange(x, a, mid + 1, max);
    } else { // x == a[mid]... we found it!
        return mid;
    }
}
Binary Search – Take III

```java
/**
 * Find the first occurrence of x in sorted array a.
 * @param x value to find
 * @param a array sorted in increasing order (a[0] <= a[1] <= ... <= a[n-1])
 * @return lowest i such that a[i]==x, or -1 if x not found in a.
 */
public static int find(int x, int[] a) {
    return binarySearchInRange(x, a, 0, a.length);
}
```

```java
private static int binarySearchInRange(int x, int[] a, int first, int max) {
    if (first >= max) {
        return -1; // range has dwindled to nothingness
    }
    int mid = (first + max) / 2;
    if (x < a[mid]) {
        return binarySearchInRange(x, a, first, mid);
    } else if (x > a[mid]) {
        return binarySearchInRange(x, a, mid + 1, max);
    } else {
        // x == a[mid]... we found it!
        return mid;
    }
}
```

© Robert Miller, Saman Amarasinghe, Max Goldman 2009-2014
/**
 * Find the first occurrence of x in sorted array a.
 *
 * @param x value to find
 * @param a array sorted in increasing order (a[0] <= a[1] <= ... <= a[n-1])
 * @return lowest i such that a[i]==x, or -1 if x not found in a.
 */
public static int find(int x, int[] a) {
    return binarySearchInRange(x, a, 0, a.length);
}

private static int binarySearchInRange(int x, int[] a, int first, int max) {
    if (first >= max) {
        return -1; // range has dwindled to nothingness
    }
    int mid = (first + max) / 2;
    if (x < a[mid]) {
        return binarySearchInRange(x, a, first, mid);
    } else if (x == a[mid]) {
        if (x == a[mid - 1]) {
            return binarySearchInRange(x, a, first, mid);
        } else {
            return mid; // it's the first
        }
    } else { // x == a[mid] we found it, but check if it's the first
        if (x == a[mid - 1]) {
            return binarySearchInRange(x, a, first, mid);
        } else {
            return mid; // it's the first
        }
    }
}
/**
 * Find the first occurrence of x in sorted array a.
 *
 * @param x value to find
 * @param a array sorted in increasing order (a[0] <= a[1] <= ... <= a[n-1])
 * @return lowest i such that a[i]==x, or -1 if x not found in a.
 */

public static int find(int x, int[] a) {
    return binarySearchInRange(x, a, 0, a.length);
}

/**
 ... .*/

private static int binarySearchInRange(int x, int[] a, int first, int max) {
    if (first >= max) {
        return -1; // range has dwindled to nothingness
    }
    int mid = (first + max) / 2;
    if (x < a[mid]) {
        return binarySearchInRange(x, a, first, mid);
    } else if (x > a[mid]) {
        return binarySearchInRange(x, a, mid + 1, max);
    } else {
        // x == a[mid] we found it, but check if it's the first
        if (mid > 0 && x == a[mid - 1]) {
            // not the first! search lower half
            return binarySearchInRange(x, a, first, mid);
        } else {
            return mid; // it's the first
        }
    }
}