Testing and Debugging

Tips & Tricks

Ibrahim Albluwi
Repeat Until Deadline:
Repeat Until Deadline:

Hack!
Repeat Until Deadline:

Hack!

Click Check All Submitted Files
Repeat Until Deadline:

Hack!

Click Check All Submitted Files

If all correctness tests pass:

Celebrate

Break
Repeat Until Deadline:

Hack!

Click Check All Submitted Files

If all correctness tests pass:

   Celebrate
   Break

Fake some test cases

If in the mood:

   Choke off CheckStyle
Repeat Until Deadline:

- Hack!
- Click Check All Submitted Files
- If all correctness tests pass:
  - Celebrate
  - Break

Fake some test cases

If in the mood:

- Choke off CheckStyle

Not Realistic!
Repeat Until Deadline:

- Hack!
- Click Check All Submitted Files

If all correctness tests pass:

- Celebrate
- Break

Fake some test cases

If in the mood:

- Choke off CheckStyle

Not Realistic!

Doesn’t work in 226!
COS 126 Unofficial Coding “Strategy”

Repeat 10 times:

Repeat Until Deadline:

Hack!

Click Check All Submitted Files

If all correctness tests pass:

Celebrate

Break

Fake some test cases

If in the mood:

Choke off CheckStyle

Not Realistic!

Doesn’t work in 226!
Intended Coding Strategy

Repeat:
Intended Coding Strategy

Repeat:

Repeat:
Repeat:

Repeat:

Hack thoughtfully and with style!
Repeat:

- Repeat:
  - Hack thoughtfully and with style!
  - Test
Repeat:

Repeat:

Hack thoughtfully and with style!
Test

If all tests pass:

Break
Intended Coding Strategy

Repeat:

Repeat:

Hack thoughtfully and with style!
Test
If all tests pass:

Break

Click Check All Submitted Files
Intended Coding Strategy

Repeat:

Repeat:

Hack thoughtfully and with style!
Test

If all tests pass:

Break

Click Check All Submitted Files

If all tests pass:

Break
Intended Coding Strategy

Repeat:

Repeat:

Hack thoughtfully and with style!
Test

If all tests pass:

Break

Click Check All Submitted Files

If all tests pass:

Break

Celebrate!
Today’s Class Meeting

Goals:

» Testing and debugging tips and tricks.
» Help you succeed in 226.
» Develop healthy programming habits.

Not Goals:

» Rigorous introduction to testing.
» Prepare you for a SW Testing Engineer job.
Got to:

http://etc.ch/i7VR
Which of the following best describes you as you work on programming assignments?

(A) **Idealist**: Codes very carefully. Usually gets it right from the first shot. Doesn’t need to test much.

(B) **Pragmatist**: Let’s get something up and running quickly. Careful testing will let us know if there is an issue.

(C) **Submissionist**: Why code too carefully? Why test carefully? **KEEP CALM AND CHECK ALL SUBMITTED FILES.**
Which of the following tests are *necessary* and *sufficient* for testing a method that *returns the maximum of three integers*.

A.  (1, 2, 3)

B.  (1, 2, 3) (3, 2, 1) (1, 3, 2)

C.  (1, 2, 3) (1, 3, 2) (2, 1, 3) (2, 3, 1) (3, 1, 2) (3, 2, 1)

D.  None of the above.
Quiz # 1

Which of the following tests are necessary and sufficient for testing a method that returns the maximum of three integers.

A. (1, 2, 3)
B. (1, 2, 3) (3, 2, 1) (1, 3, 2)
C. (1, 2, 3) (1, 3, 2) (2, 1, 3) (2, 3, 1) (3, 1, 2) (3, 2, 1)
D. None of the above.

Tip # 1

Tests can be written before the program is implemented. Blackbox Testing: Test based on problem description.
Which of the following tests are necessary and sufficient for testing a method that returns the maximum of three integers.

A. (1, 2, 3)
B. (1, 2, 3) (3, 2, 1) (1, 3, 2)
C. (1, 2, 3) (1, 3, 2) (2, 1, 3) (2, 3, 1) (3, 1, 2) (3, 2, 1)
D. None of the above.

Tip # 1 Tests can be written before the program is implemented.
Blackbox Testing: Test based on problem description.
Which of the following tests are necessary and sufficient for testing a method that returns the maximum of three integers.

A. (1, 2, 3)
B. (1, 2, 3) (3, 2, 1) (1, 3, 2)
C. (1, 2, 3) (1, 3, 2) (2, 1, 3) (2, 3, 1) (3, 1, 2) (3, 2, 1)
D. None of the above.

Tip # 1 Tests can be written before the program is implemented.

Blackbox Testing: Test based on problem description.

Tip # 2 Think carefully about the domain of the inputs.
Quiz # 1

Which of the following tests are necessary and sufficient for testing a method that returns the maximum of three integers. integers can be negative!

A. (1, 2, 3)

B. (1, 2, 3) (3, 2, 1) (1, 3, 2)

C. (1, 2, 3) (1, 3, 2) (2, 1, 3) (2, 3, 1) (3, 1, 2) (3, 2, 1)

D. None of the above.

Tip # 1
Tests can be written before the program is implemented. Blackbox Testing: Test based on problem description.

Tip # 2
Think carefully about the domain of the inputs.
Example

The following code passes all test cases with positive integers but fails all test cases with *negative* integers!

```c
int max(int a, int b, int c) {
    int max = 0;
    if (a > max)
        max = a;
    if (b > max)
        max = b;
    if (c > max)
        max = c;
    return max;
}
```
Which of the following tests are *necessary* and *sufficient* for testing a method that *returns the maximum of three integers*.

A. \((1, 2, 3) (3, 2, 1) (1, 3, 2) (-1, -2, -3) (-3, -2, -1) (-1, -3, -2)\)

B. All 3-permutations of \(-3, -2, -1, 1, 2, 3\).

C. Thousands of randomly generated positive and negative integers.

D. None of the above.
Which of the following tests are necessary and sufficient for testing a method that returns the maximum of three integers.

A. (1, 2, 3) (3, 2, 1) (1, 3, 2) (-1, -2, -3) (-3, -2, -1) (-1, -3, -2)
B. All 3-permutations of -3, -2, -1, 1, 2, 3.
C. Thousands of randomly generated positive and negative integers.
D. None of the above.
Which of the following tests are *necessary* and *sufficient* for testing a method that *returns the maximum of three integers*.

A.  \((1, 2, 3) (3, 2, 1) (1, 3, 2) (-1, -2, -3) (-3, -2, -1) (-1, -3, -2)\)

B.  All 3-permutations of -3, -2, -1, 1, 2, 3.

C.  Thousands of randomly generated positive and negative integers.

D.  None of the above.
Which of the following tests are necessary and sufficient for testing a method that returns the maximum of three integers.

A. \((1, 2, 3) (3, 2, 1) (1, 3, 2) (-1, -2, -3) (-3, -2, -1) (-1, -3, -2)\)

B. All 3-permutations of \(-3, -2, -1, 1, 2, 3\).

C. Thousands of randomly generated positive and negative integers.

D. None of the above.

Tip # 3 Think of input equivalence classes.

\((1, 2, 3)\) is equivalent to \((2, 3, 4) = (\text{min, mid, max})\)

\((3, 2, 1)\) is equivalent to \((30, 5, 4) = (\text{max, mid, min})\)
Example

The following code passes all test cases with 3-permutations of -3, -2, -1, 1, 2, 3. However, it could fail if the input has duplicates!

```c
int max(int a, int b, int c) {
    int max = 0;
    if (a > b && a > c)
        max = a;
    if (b > a && b > c)
        max = b;
    if (c > a && c > b)
        max = c;
    return max;
}
```
Using \{-3, -2, -1, 1, 2, 3\}, is it enough to test all possible 3-tuples (permutations with repetition)?

Not necessarily!
Example
int max(int a, int b, int c) {
    int max = 0;
    if (a - b >= 0 && a - c >= 0)
        max = a;
    if (b - a >= 0 && b - c >= 0)
        max = b;
    if (c - a >= 0 && c - b >= 0)
        max = c;
    return max;
}
The following code passes all test cases with 3-tuples from \{-3, -2, -1, 1, 2, 3\}. However, it could fail when the used numbers are too small or too large, like:

\[
a = 2147483647 \quad b = 2147483647 \quad c = -2147483647
\]

```c
int max(int a, int b, int c) {
    int max = 0;
    if (a - b >= 0 && a - c >= 0)
        max = a;
    if (b - a >= 0 && b - c >= 0)
        max = b;
    if (c - a >= 0 && c - b >= 0)
        max = c;
    return max;
}
```
The following code passes all test cases with 3-tuples from \{-3, -2, -1, 1, 2, 3\}. However, it could fail when the used numbers are too small or too large, like:

\[
\begin{align*}
a &= 2147483647 & b &= 2147483647 & c &= -2147483647 \\
\end{align*}
\]

```c
int max(int a, int b, int c) {
    int max = 0;
    if (a - b >= 0 && a - c >= 0)
        max = a;
    if (b - a >= 0 && b - c >= 0)
        max = b;
    if (c - a >= 0 && c - b >= 0)
        max = c;
    return max;
}
```

Overflow!

\[
\begin{align*}
2147483647 + 2147483647 &= -2 \\
\end{align*}
\]
Example

The following code passes all test cases with 3-tuples from \{-3, -2, -1, 1, 2, 3\}. However, it could fail when the used numbers are too small or too large, like:

\[
\begin{align*}
a &= 2147483647 & b &= 2147483647 & c &= -2147483647
\end{align*}
\]

```c
int max(int a, int b, int c) {
    int max = 0;
    if (a - b >= 0 && a - c >= 0)
        max = a;
    if (b - a >= 0 && b - c >= 0)
        max = b;
    if (c - a >= 0 && c - b >= 0)
        max = c;
    return max;
}
```

Overflow!

\[
\begin{align*}
2147483647 + 2147483647 &= -2
\end{align*}
\]

Tip # 4 Always test boundary inputs and corner cases.
Tip # 5

Blackbox testing may not be enough.

**Whitebox Testing**: Generate tests based on code.

Examine code and make sure there are test cases that cover all possible program flow paths.
Blackbox testing may not be enough.

**Whitebox Testing**: Generate tests based on code.

Examine code and make sure there are test cases that cover all possible program flow paths.

Example:

```java
if (a == true)
    doSomething();
else
    doSomethingElse();

for (int i = n; i > 0; i--)
    doAnotherThing();

doAFinalThing();
```
Blackbox testing may not be enough.

**Whitebox Testing**: Generate tests based on code.

Examine code and make sure there are test cases that cover all possible program flow paths.

**Example**

```java
if (a == true)
    doSomething();
else
    doSomethingElse();

for (int i = n; i > 0; i--)
    doAnotherThing();

doAFinalThing();
```
Blackbox testing may not be enough. **Whitebox Testing**: Generate tests based on code.

Examine code and make sure there are test cases that cover all possible program flow paths.

**Example**

```java
if (a == true)
    doSomething();
else
    doSomethingElse();

for (int i = n; i > 0; i--)
    doAnotherThing();

doAFinalThing();
```

- Test both branches
- Test entering the loop and not entering the loop
Which of the following tests could reveal the bug in the following stack code?

**Hint**: Is size always correctly updated?

A. Calling `push` then `toString` then `pop`.

B. Calling `push` then `pop` then `toString`.

C. Calling `push` (many times) then `pop` (many times) then `push`.

D. All of the above.
push(x):
  if (size == cap)
    Error
  last++
  data[last] = x
  size++

pop():
  if (size == 0)
    Error
  x = data[last]
  last--
  return x

toString():
  s = ""
  for (i=0 -> size-1)
    s += data[i]
  return s

Which of the following tests could reveal the bug in the following stack code?

**Hint**: Is size always correctly updated?

A. Calling `push` then `toString` then `pop`.
B. Calling `push` then `pop` then `toString`.
C. Calling `push` (many times) then `pop` (many times) then `push`.
D. All of the above.
Which of the following tests could reveal the bug in the following stack code?

**Hint**: Is size always correctly updated?

A. Calling `push` then `toString` then `pop`.
B. Calling `push` then `pop` then `toString`.
C. Calling `push` (many times) then `pop` (many times) then `push`.
D. All of the above.
Tip # 6  Test different *orderings* of method calls.
Tip # 6  Test different *orderings* of method calls.

Tip # 7  Test methods on different *states* of the data structure
Implement a simple test client *before* starting to code the ADT.
Implement a simple test client *before* starting to code the ADT.

Write tests that cover all *input equivalence classes*. 
Summary

- Implement a simple test client before starting to code the ADT.
- Write tests that cover all input equivalence classes.
- Always test boundary inputs and corner cases.
Implement a simple test client *before* starting to code the ADT.

Write tests that cover all *input equivalence classes*.

Always test *boundary inputs* and corner cases.

Write tests that *cover all possible flow paths* in the code.
Implement a simple test client before starting to code the ADT.

Write tests that cover all input equivalence classes.

Always test boundary inputs and corner cases.

Write tests that cover all possible flow paths in the code.

Intermix method calls to see if one breaks another.
Implement a simple test client *before* starting to code the ADT.

Write tests that cover all *input equivalence classes*.

Always test *boundary inputs* and corner cases.

Write tests that *cover all possible flow paths* in the code.

*Intermix method calls* to see if one breaks another.

Test method calls with all possible *states* of the object.
Summary

- Implement a simple test client *before* starting to code the ADT.
- Write tests that cover all *input equivalence classes*.
- Always test *boundary inputs* and corner cases.
- Write tests that *cover all possible flow paths* in the code.
- *Intermix method calls* to see if one breaks another.
- Test method calls with all possible *states* of the object.

Lesson

Testing can show the presence of errors but not their absence!
**What property of the array does this puzzle compute?**

```
public class Program {
    public static int Puzzle(int[] a) {
        return 0;
    }
}
```

<table>
<thead>
<tr>
<th>A</th>
<th>EXPECTED RESULT</th>
<th>YOUR RESULT</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>{0, 0}</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>{1, 0}</td>
<td>1</td>
<td>0</td>
<td>Mismatch</td>
</tr>
<tr>
<td>{1, 32}</td>
<td>31</td>
<td>0</td>
<td>Mismatch</td>
</tr>
<tr>
<td>{-46, 0}</td>
<td>46</td>
<td>0</td>
<td>Mismatch</td>
</tr>
<tr>
<td>{0, 0, 0, 0, 0}</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
Failed test cases?

Time to Debug!
Failed test cases?

Time to Debug!
Tip # 1
Understand what the error means.

Confused?

▸ Copy and paste error to Google.

```
Students.java:90: error: illegal start of expression
}
```

▸ Use Java compiler messages cheatsheets.

Examples:
https://dzone.com/articles/50-common-java-errors-and-how-to-avoid-them-part-1
http://mindprod.com/jgloss/compileerrormessages.html#TYPESAFETYERAASED
Easiest Bugs: Compile Time Errors

Tip # 2

Focus on the first error first.

- An error can produce a cascade of other errors. Fixing the first error, automatically fixes all subsequent errors caused by it.

```java
ErdosRenyi.java:29: error: '.class' expected
    For (int i = 0; i < n; i++)
    ^

ErdosRenyi.java:29: error: > expected
    For (int i = 0; i < n; i++)
    ^

ErdosRenyi.java:29: error: not a statement
    For (int i = 0; i < n; i++)
    ^

ErdosRenyi.java:29: error: ';' expected
    For (int i = 0; i < n; i++)
    ^

4 errors
```
Know the anatomy of a runtime exception.

```java
Exception in thread "main"
java.lang.IndexOutOfBoundsException: -1
    at test.convertIndex(test.java:6)
    at test.findMax(test.java:15)
    at test.max(test.java:31)
    at test.main(test.java:41)
```

Confused? Copy and paste exception to Google (without message and trace).
Know the anatomy of a runtime exception.

```java
Exception in thread "main"
java.lang.IndexOutOfBoundsException: -1
   at test.convertIndex(test.java:6)
   at test.findMax(test.java:15)
   at test.max(test.java:31)
   at test.main(test.java:41)
```

Confused? Copy and paste exception to Google (without message and trace).
Runtime Exceptions

Know the anatomy of a runtime exception.

Exception in thread "main"
java.lang.IndexOutOfBoundsException: -1
at test.convertIndex(test.java:6)
at test.findMax(test.java:15)
at test.max(test.java:31)
at test.main(test.java:41)

Confused? Copy and paste exception to Google (without message and trace).
Know the anatomy of a runtime exception.

Exception name: java.lang.IndexOutOfBoundsException
Message: -1
StackTrace:
- at test.convertIndex(test.java:6)
- at test.findMax(test.java:15)
- at test.max(test.java:31)
- at test.main(test.java:41)

Confused? Copy and paste exception to Google (without message and trace).
Debugging Non-trivial Errors

Tip # 3
Know exactly *when* the error happens.
Debugging Non-trivial Errors

Tip # 3  Know exactly *when* the error happens.

Can you *reproduce* the error?
Debugging Non-trivial Errors

**Tip # 3**  Know exactly *when* the error happens.

Can you *reproduce* the error?

**Example 1:** Assume `max(-1, 1, 1)` fails:
- Does `max` fail for all inputs?
- Does it fail only with negative numbers?
- Does it fail only when there are duplicates?
- Does it fail only with `(min, max, max)`?
Debugging Non-trivial Errors

Tip #3: Know exactly *when* the error happens.

Can you *reproduce* the error?

**Example 1:** Assume `max(-1, 1, 1)` fails:
- Does `max` fail for all inputs?
- Does it fail only with negative numbers?
- Does it fail only when there are duplicates?
- Does it fail only with `(min, max, max)`?

**Example 2:** Autograder says: Intermixing calls to `push`, `pop` and `top` throws an exception.
- Can you come up a sequence of `push`, `pop` and `top` calls that would produce the same exception?
Tip # 4

Use `print` statements to know *where* and *why* the error happens.
Debugging Non-trivial Errors

Tip # 4

Use `print` statements to know `where` and `why` the error happens.

Example:

```cpp
int myFunction(int x) {
    if (isInState1())
        x = doSomething();
    if (isInState2())
        x = doSomethingElse();
    return x;
}
```
Tip # 4

Use `print` statements to know where and why the error happens.

Example:

```java
int myFunction(int x) {
    StdOut.println(x);
    if (isInState1())
        x = doSomething();
    StdOut.println(x);
    if (isInState2())
        x = doSomethingElse();
    StdOut.println(x);
    return x;
}
```

Helps understand how the value of `x` changes.
Debugging Non-trivial Errors

Tip # 4

Use `print` statements to know `where` and `why` the error happens.

Example:

```c
int myFunction(int x) {
    if (isInState1())
        x = doSomething();
    if (isInState2())
        x = doSomethingElse();
    return x;
}
```
Tip # 4  Use `print` statements to know *where* and *why* the error happens.

Example:

```java
int myFunction(int x) {
    if (isInState1()) {
        x = doSomething();
        StdOut.println("State1");
    }

    if (isInState2()) {
        x = doSomethingElse();
        StdOut.println("State2");
    }

    return x;
}
```

Helps understand program flow.
Using Print Statements

Trick # 1  Use `java.util.Arrays`.

**Print** 1D array: `StdOut.print(Arrays.toString(a))`

**Print** 2D array: `StdOut.print(Arrays.deepToString(a))`

**Fill** array: `StdOut.print(Arrays.fill(a, value))`

**Compare** arrays: `Arrays.equals(a1, a2)`
`Arrays.deepEquals(a1, a2)`
Using Print Statements

**Trick # 1**

Use `java.util.Arrays`.

- **Print** 1D array: `StdOut.print(Arrays.toString(a))`
- **Print** 2D array: `StdOut.print(Arrays.deepToString(a))`
- **Fill** array: `StdOut.print(Arrays.fill(a, value))`
- **Compare** arrays: `Arrays.equals(a1, a2)`
  `Arrays.deepEquals(a1, a2)`

**Warning # 1**

Do not forget to remove debugging print statements before submitting!
Tip # 5 Check common sources of errors.
Tip # 5 Check common sources of errors.

- **Copied-and-pasted code:** It is very common to forget to make the needed changes after copying code.
Debugging Non-trivial Errors

Tip # 5
Check common sources of errors.

- **Copied-and-pasted code:** It is very common to forget to make the needed changes after copying code.

- **Variable scope:** Are there different variables with the same name?
Debugging Non-trivial Errors

**Tip # 5** Check common sources of errors.

- **Copied-and-pasted code:** It is very common to forget to make the needed changes after copying code.

- **Variable scope:** Are there different variables with the same name?

- **Others?**
Debugging Non-trivial Errors

Tip # 5
Check common sources of errors.

- **Copied-and-pasted code**: It is very common to forget to make the needed changes after copying code.

- **Variable scope**: Are there different variables with the same name?

- **Others?**
  - if (var = true)
Debugging Non-trivial Errors

Tip # 5  Check common sources of errors.

- **Copied-and-pasted code:** It is very common to forget to make the needed changes after copying code.

- **Variable scope:** Are there different variables with the same name?

- **Others?**
  
  - if (var = true)
  
  - if (str1 == str2)
Tip # 5 Check common sources of errors.

- **Copied-and-pasted code:** It is very common to forget to make the needed changes after copying code.

- **Variable scope:** Are there different variables with the same name?

- **Others?**
  
  ```java
  -if (var = true)
  -if (str1 == str2)
  -Boolean [] isTrue = new Boolean[10];
  ```
Rubber duck debugging

From Wikipedia, the free encyclopedia

In software engineering, rubber duck debugging or rubber ducking is a method of debugging code. The name is a reference to a story in the book *The Pragmatic Programmer* in which a programmer would carry around a rubber duck and debug their code by forcing themselves to explain it, line-by-line, to the duck. Many other terms exist for this technique, often involving different inanimate objects.

Many programmers have had the experience of explaining a problem to someone else, possibly even to
Tip # 6  Document your assumptions
Debugging Non-trivial Errors

Tip # 6  Document your assumptions

When you walk through the code (with the rubber duck):
Debugging Non-trivial Errors

Tip # 6  Document your assumptions

When you walk through the code (with the rubber duck):

- **Insert comments** explaining your logic.
When you walk through the code (with the rubber duck):

- **Insert comments** explaining your logic.
- After certain blocks of code, insert comments explaining *why you are sure the code must be correct* at to that point.
Debugging Non-trivial Errors

Tip # 6  Document your assumptions

When you walk through the code (with the rubber duck):

- Insert comments explaining your logic.

- After certain blocks of code, insert comments explaining why you are sure the code must be correct at to that point.

- Use assertions!
Debugging Non-trivial Errors

Tip # 6  Document your assumptions

When you walk through the code (with the rubber duck):

- **Insert comments** explaining your logic.

- After certain blocks of code, insert comments explaining **why you are sure the code must be correct** at that point.

- Use **assertions**!

  
  Your assumption.
  An error is thrown if not true.  
  
  Message displayed when error is thrown
Debugging Non-trivial Errors

**Tip # 6** Document your assumptions

When you walk through the code (with the rubber duck):

- *Insert comments* explaining your logic.

- After certain blocks of code, insert comments explaining *why you are sure the code must be correct* at to that point.

- Use *assertions*!

  - Your assumption.
  - An error is thrown if not true.
  - Message displayed when error is thrown

  ```python
  assert booleanExpression : Value
  ```
Debugging Non-trivial Errors

Tip # 6  Document your assumptions

When you walk through the code (with the rubber duck):

- **Insert comments** explaining your logic.

- After certain blocks of code, insert comments explaining why you are sure the code must be correct at to that point.

- Use **assertions**!

  Your assumption.  
  An error is thrown if not true.  
  Message displayed when error is thrown

  ```
  assert booleanExpression : Value
  ```
Assertions Examples
void foo() {
    for (...) {
        if (...) {
            return;
        }
        assert false;
    }
}

Assumption: Flow should never reach here!
Assertions Examples

```c
void foo() {
    for (...) {
        if (...) {
            return;
        }
        assert false;
    }
}
```

**Assumption**: Flow should never reach here!

```c
if (i % 3 == 0) {
    ...
} else if (i % 3 == 1) {
    ...
} else {
    assert i % 3 == 2 : i;
    ...
}
```

**Assumption**: i % 3 == 2.
Fails if i is negative.
void foo() {
    for (...) {
        if (...)
            return;
    }
    assert false;
}

Assumption: Flow should never reach here!

if (i % 3 == 0) {
    ...
} else if (i % 3 == 1) {
    ...
} else {
    assert i % 3 == 2 : i;
    ...
}

Assumption: i % 3 = 2. Fails if i is negative.

void insert(int val) {
    assert isBST() : "BST properties violated"
    // tree insertion code

    assert isBST() : "BST properties violated"
    assert isBalanced() : "Insertion misbalances BST";
}
Debugging Non-trivial Errors

Trick # 2  Use the Debugger.
Debugging Non-trivial Errors

Trick # 2 Use the Debugger.

Demo!