Announcements

- HW0, HW1, HW2, and HW3 are graded
  - Grades and feedback in Submitty
  - Email us at csci2600@cs.lists.rpi.edu
  - Use the LMS Discussion board!
- Exam 1, Quiz 1, 2, 3
  - Grades in Submitty Rainbow Grades

Exam Review: bitwise

- Loop invariant: \(a_0 + b_0 = (a + b)g + c + mg\)
- Decrementing function: \(a - b\)

Exam Review: Specification Strength

```java
int find(int[] a, int key)
```

Spec A: **requires**: \(\text{key} \text{ in } a\)
  - **returns**: \(\text{index} \text{ such that } a[\text{index}] = \text{key}\)

Spec B: **requires**: \(\text{key} \text{ in } a\)
  - **returns**: \(\text{largest index} \text{ such that } a[\text{index}] = \text{key}\)

Spec C: **returns**: \(\text{index} \text{ such that } a[\text{index}] = \text{key}\)
  - if \(\text{key} \text{ in } a\), and \(-1\) if \(\text{key} \text{ not in } a\)

Spec D: **returns**: \(\text{smallest index} \text{ such that } a[\text{index}] = \text{key}\) if \(\text{key} \text{ in } a\)
  - **throws**: KeyNotFoundException if \(\text{key} \text{ not in } a\)

And More...

- Benevolent side effects
- Poly question
  - (1) representation exposure
  - (2) mutation
  - (3) 0 coefficients
- Rep invariant and abstraction function
  - Queue
Announcements

- HW4 will be out tonight, due October 20th
  - **LONG!!!**
  - **START EARLY!!!** Several tasks:
    - 1) design the ADT following the ADT methodology
      - Mutable vs. immutable
      - Operations (creators, mutators, etc.) + their specs
    - 2) write a test suite based on those specs (before coding, test-first principle)
    - 3) then code: choose rep, write the RI and AF
    - 4) then augment test suite based on actual implementation and measure coverage

Outline

- Testing
  - Introduction
  - Strategies for choosing tests suites
    - Black-box testing
    - White-box testing

What is Testing?

- **Testing**: the process of executing software with the intent of finding errors
- **Good testing**: a high probability of finding yet undiscovered errors
- **Successful testing**: discovers unknown errors

“Program testing can be used to show the presence of bugs, but never to show their absence.” Edsger Dijkstra 1970

Quality Assurance (QA)

- The process of uncovering problems and improving the quality of software. Testing is the major part of QA
- QA is testing plus other activities:
  - Static analysis (finding bugs without execution)
  - Proofs of correctness (theorems)
  - Code reviews (people reading each other’s code)
  - Software process (development methodology)
- No single activity or approach can guarantee software quality

Famous Software Bugs

- Ariane 5 rocket’s first launch in 1996
  - The rocket exploded 37 seconds after launch
  - Reason: a bug in control software
  - Cost: over $1 billion

- Therac-25 radiation therapy machine
  - Excessive radiation killed patients
  - Reason: software bug linked to a race condition, missed during testing

Famous Software Bugs

- Mars Polar Lander
  - Legs deployed after sensor falsely indicated craft had touched down 130 feet above surface
  - Reason: one bad line of software
  - Cost: $110 million
  - And many more…
    - Toyota Prius breaks and engine stalling (2005)
    - And many many more…
Cost to Society (Source: NIST Planning Report 2002)

- Inadequate testing infrastructure costs the US $22-60 billion annually
- Improvement in testing infrastructure can save one third of $22-60 billion annual cost
- Testing accounts for 50% of software development cost
- Maintenance (bug fixes and upgrades) accounts for up to 95% of total software cost

Scope (Phases) of Testing

- Unit testing
  - Does each module do what it is supposed to do?
- Integration testing
  - Do the parts, when put together, produce the right result?
- System testing
  - Does program satisfy functional requirements?
  - Does it work within overall system?
    - Behavior under increased loads, failure behavior, etc.

Unit Testing

- Our focus will be on unit testing
- Tests a single unit in isolation from all others
- In object-oriented programming, unit testing mostly means class testing
  - Tests a single class in isolation from others

Why Is Testing So Hard?

// requires: 1 <= x,y,z <= 10000
// returns: computes some f(x,y,z)
int f(int x, int y, int z)

- Exhaustive testing would require 1 trillion runs! And this is a trivially small problem
- The key problem: choosing set of inputs (i.e., test suite)
  - Small enough to finish quickly
  - Large enough to validate program

sqrt Example

// throws: IllegalArgumentException if x < 0
// returns: approximation to square root of x
public double sqrt(double x)

- What are some values of x worth trying?
  - x < 0 (exception thrown)
  - x >= 0 (returns normally)
  - around 0 (boundary conditions)
  - Perfect squares, non-perfect squares
  - x < 1 (sqrt(x) > x in this case), x = 1, x > 1

Outline

- Testing
  - Introduction
  - Strategies for choosing tests suites
    - Black box testing
    - White box testing
Testing Strategies

- Test case: specifies
  - Inputs + pre-test state of the software
  - Expected result (outputs and post-test state)

- Black box testing:
  - We ignore the code of the program. We look at the specification (roughly, given some input, was the produced output correct according to the spec?)
  - Choose inputs without looking at the code

- White box (clear box, glass box) testing:
  - We use knowledge of the code of the program (roughly, we write tests to "cover" internal paths)
  - Choose inputs with knowledge of implementation

Black Box Testing Advantages

- Robust with respect to changes in implementation (independent of implementation)
  - Test data need not be changed when code is changed
  - Allows for independent testers
  - Tests can be developed before code based on specifications. (Do this in HW4!)

Black Box Testing Heuristic

- Choose test inputs based on paths in specification
  // returns: a if a > b
  // b if b > a
  // a if a = b
  int max(int a, int b)

  3 paths, 3 test cases:
  - (4,3) => 4 (input along path a > b)
  - (3,4) => 4 (input along path b > a)
  - (3,3) => 3 (input along path a = b)

Black Box Testing Heuristic

- Choose test inputs based on paths in specification
  // returns: index of first occurrence of value in a
  // or -1 if value does not occur in a
  int find(int[] a, int value)

  What are good test cases?
  - ([4,3,5,6], 5) => 2
  - ([4,3,5,6], 7) => -1
  - ([4,5,3,5], 5) => 1

sqrt Example

// throws: IllegalArgumentException if x < 0
// returns: approximation to square root of x
public double sqrt(double x)

- What are some values of x worth trying?
  - We used this heuristic in sqrt example. It tells us a value of x => 0 (returns normally), and a value of x < 0 (exception thrown) are worth trying

Black Box Heuristics

- "Paths in specification" heuristic is a form of equivalence partitioning
- Equivalence partitioning partitions input and output domains into equivalence classes
  - Intuition: values from different classes drive program through different paths
  - Intuition: values from the same equivalence class drive program through "same path", program will likely behave "equivalently"
Black Box Heuristics

- Choose test inputs from each equiv. class

\[
\begin{align*}
&\text{// returns: } 0 \leq \text{result} \leq 5 \\
&\text{// throws: } \text{SomeException if } \arg < 0 \text{ || } \arg > 10 \\
&\text{int } f(\text{int } \arg)
\end{align*}
\]

There are three equivalence classes:
- \( \arg < 0 \)
- \( 0 \leq \arg \leq 10 \)
- \( 10 < \arg \)

We write tests with values of \( \arg \) from each class

- Stronger vs. weaker spec. What if the spec said // requires: \( 0 \leq \arg \leq 10 \)?

Black Box Testing Heuristic: Boundary Value Analysis

- Idea: choose test inputs at the edges of the equivalence classes
- Why?
  - Off-by-one bugs, forgot to handle empty container, overflow errors in arithmetic
  - Cases at the edges of the "main" class have high probability of revealing these common errors
  - Complements equivalence partitioning

Equivalence Partitioning

- Examples of equivalence classes
  - Valid input \( x \) in interval \([a..b]\): this defines three classes "\( x < a \)", "\( a <= x <= b \)", "\( b < x \)"
  - Input \( x \) is boolean: classes "true" and "false"

Choosing test values

- Choose a typical value in the middle of the "main" class (the one that represents valid input)
- Also choose values at the boundaries of all classes: e.g., use \( a-1, a, a+1, b-1, b, b+1 \)

Equivalence Partitioning and Boundary Values

- Suppose our specification says that valid input is an array of 4 to 24 numbers, and each number is a 3-digit positive integer

  One dimension: partition size of array
  - Classes are "\( n<4 \)", "\( 4 <= n <= 24 \)", "\( 24 < n \)"
  - Chosen values: \( 3, 4, 5, 14, 23, 24, 25 \)

  Another dimension: partition integer values
  - Classes are "\( x<100 \)", "\( 100 <= x <= 999 \)", "\( 999 < x \)"
  - Chosen values: \( 99, 100, 101, 500, 998, 999, 1000 \)

Equivalence Partitioning and Boundary Values

// returns: \( \text{index s.t. } a[\text{index}] = \text{value} \), or \(-1\) if \( \text{value} \) does not occur in \( a \)
\[
\text{int } \text{find}(\text{int}[] a, \text{int } \text{value})
\]

What is a good partition of the input domain?

One dimension: size of the array

- People often make errors for arrays of size 1, we decide to create a separate equivalence class
- Classes are "empty array", "array with one element", "array with many elements"
Equivalence Partitioning and Boundary Values

- We can also partition the output domain: the location of the value
- Four classes: “first element”, “last element”, “middle element”, “not found”

<table>
<thead>
<tr>
<th>Array</th>
<th>Value</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Empty</td>
<td>5</td>
<td>-1</td>
</tr>
<tr>
<td>[7]</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>[7]</td>
<td>2</td>
<td>-1</td>
</tr>
<tr>
<td>[1, 6, 4, 7, 2]</td>
<td>1</td>
<td>0 (boundary, start)</td>
</tr>
<tr>
<td>[1, 6, 4, 7, 2]</td>
<td>4</td>
<td>2 (mid array)</td>
</tr>
<tr>
<td>[1, 6, 4, 7, 2]</td>
<td>2</td>
<td>4 (boundary, end)</td>
</tr>
<tr>
<td>[1, 6, 4, 7, 2]</td>
<td>3</td>
<td>-1</td>
</tr>
</tbody>
</table>

Other Boundary Cases

- Arithmetic
  - Smallest/largest values
  - Zero

- Objects
  - Null
  - Circular list
  - Same object passed to multiple arguments (aliasing)

Boundary Value Analysis: Arithmetic Overflow

```java
public int abs(int x)
```

- What are some values worth trying?
  - Equivalence classes are x < 0 and x >= 0
  - x = -1, x = 1, x = 0 (boundary condition)
- How about x = Integer.MIN_VALUE?

  // this is -2147483648 = -2^{31}

  // System.out.println(Math.abs(x) < 0) prints true!

Boundary Value Analysis: Aliasing

```java
void appendList(List<Integer> src,
                List<Integer> dest) {
    while (src.size() > 0) {
        Integer elt = src.remove(src.size()-1);
        dest.add(elt);
    }
}
```

- What happens if we run `appendList(list, list)`?
  - Aliasing.

Summary So Far

- Testing is hard. We cannot run all inputs
- Key problem: choose test suites such that
  - Small enough to finish in reasonable time
  - Large enough to validate the program (reveal bugs, or build confidence in absence of bugs)
- All we have is heuristics!
  - We saw black box testing heuristics: run paths in spec, partition input/output into equivalence classes, run with input values at boundaries of these classes
  - There are also white box testing heuristics

White Box Testing

- Ensure test suite covers (covers means executes) all of the program
- Measure quality of test suite with % coverage
  - Assumption: high coverage implies few errors in program
- Focus: features not described in specification
  - Control-flow details
  - Performance optimizations
  - Alternate algorithms (paths) for different cases
### White Box Complements Black Box

```java
boolean[] primeTable[CACHE_SIZE]
// returns: true if x is prime
// false otherwise
boolean isPrime(int x) {
    if (x > CACHE_SIZE) {
        for (int i=2; i<x/2; i++)
            if (x%i==0) return false;
        return true;
    } else return primeTable[x];
}
```

---

### White Box Testing: Control-flow-based Testing

Control-flow-based white box testing:
- Extract a control flow graph (CFG)
- Test suite must cover (execute) certain elements of this control-flow graph graph
- Idea: Define a coverage target and ensure test suite covers target
  - Targets: nodes, branch edges, paths
  - Coverage target approximates “all of the program”

---

### Aside: Control-flow Graph (CFG)

- Assignment \(x=y+z\) => node in CFG: \(x=y+z\)
- If-then-else
  if (\(b\)) S1 else S2 =>
  ```
  CFG for S1  CFG for S2
  ```

---

### Aside: Control Flow Graph (CFG)

- Draw the CFG for \(gcd(a,b)\):
  ```
  1 x = a;
  2 y = y;
  3 while (x != y) {
    4     if (x > y)
    5       x = x - y;
    6     else
    7       y = y - x;
  }
  8 return x;
  ```

---

### Statement Coverage

Traditional target: statement coverage. Write test suite that covers all statements, or in other words, all nodes in the CFG
- Motivation: code that has never been executed during testing may contain errors
  - Often this is the “low-probability” code
Example

- Suppose that we write and execute two test cases
  - Test case #1: follows path 1-2-exit (e.g., we never take the loop)
  - Test case #2: 1-2-3-4-5-7-8-2-3-4-5-7-8-2-exit (loop twice, and both times take the true branch)
- Problems?

Branch Coverage

- Target: write test cases that cover all branch edges at predicate nodes
  - True and false branch edges of each if-then-else
  - The two branch edges corresponding to the condition of a loop
  - All alternatives in a SWITCH statement

Example

- We need to cover the red branch edges
  - Test case #1: follows path 1-2-exit
  - Test case #2: 1-2-3-4-5-7-8-2-3-4-5-7-8-2-exit
  - What is % branch coverage?

Branch Coverage

- Motivation for branch coverage: experience shows that many errors occur in “decision making” (i.e., branching).
  - Plus, it implies statement coverage
  - Statement coverage does not imply branch coverage
    - I.e., a suite that achieves 100% statement coverage does not necessarily achieve 100% branch coverage
    - Can you think of an example?

Example

```java
static int min(int a, int b) {
    int r = a;
    if (a <= b)
        r = a;
    return r;
}
```

- Let’s test with min(1, 2)
  - What is the statement coverage?
  - What is the branch coverage?
Rules of Testing

- First rule of testing: Do it early and do it often
  - Best to catch bugs soon, before they hide
  - Automate the process
  - Regression testing will save time

- Second rule of testing: Be systematic
  - Writing tests is a good way to understand the spec
  - Specs can be buggy too!
  - When you find a bug, write a test first, then fix