Testing

How can I choose good test cases?
How can I make testing more efficient?
What’s the goal?

What we want:

Convincing evidence that our program works correctly, for all inputs.

Can we get it?

Why or why not?
The Bad News

Program correctness is *undecidable* in general

- One of the earliest theoretical results in modern theory of computing, from Alan Turing
- A “diagonalization” argument, very similar to Goedel’s incompleteness theorem in mathematics

Consequence: We can’t prove correctness by testing

- It would take an infinite number of test cases!
Mission Impossible

Your mission, should you choose to accept it:
Write a program to detect infinite loops in other programs.
Input: A program, input for that program
Output: “True” if the program halts; “False” if it runs forever
Can this program be built?

Your program must always return the correct answer, for all (program, input) pairs.
Suppose we could ...

Suppose you wrote a program P that solved the challenge (for all programs and inputs!).

I use your program P as a method, in my program $P'(\text{program } Q, \text{input } I)$:

- if $P(Q,I)$ then { loop forever }
- else { print “OK, I’m done” }

Can your program P work on $(P’, P’)$?
Infinite Loop Detector

Yes or No

Infinite Loop Detector

Infinite Loop Detector
So what?

Maybe P works on almost all programs.
Maybe P works on all practical programs.
Maybe P almost always gives the right answer.
Maybe it’s easier to prove that a program gives the right output, if it does halt.

All reasonable. None true.
Um, so then what?

Testing can never prove program correctness

Shall we give up and not test our programs?
Modest Goals

Testing can’t prove correctness, but it can be an effective way to find bugs

We’ll never be sure we’ve tested “enough”
    But we can often tell when we haven’t!

And we can definitely do better than haphazard, monkey-at-keyboard testing.
More = Better?

First cut: The more test cases I run, the better

What’s wrong with this?
more $\neq$ better

Suppose my first test of max is $\max(12, 15)$

Then I could run 5 more tests:

$\max(10, 14), \max(5, 88), \max(13, 25), \max(10, 17), \max(100, 200)$

Or I could run 1 more test:

$\max(13, 7)$
If more isn’t better, what is?

What makes a test case valuable?
(And what do we mean by “valuable”?)
Buggy Root Finder

public roots(double a, double b, double c) {
    // Roots = -b ± sqrt(b^2 - 4ac) / 2a
    q = b*b - 4*a*c;
    if (q > 0 && a != 0) {
        ...
    } else if (q==0) {
        // The equation has exactly one root
        num_roots = 1;
        root_one = (0-b)/(2*a);
        root_two = root_one;
    } else {
        // The equation has no roots if b^2 < 4ac
        num_roots = 0;
        root_one = -1;
        root_two = -1;
    }
}
How about random testing?

Strategy: Generate random inputs, uniformly distributed

Example: Consider that buggy root finder
What if we chose inputs randomly from the interval \((0, 10000000000)
Suppose we ran 1000000 test cases ... enough?
Too many possible inputs ...
But probably many are similar ...
Similar with respect to correctness ...
<table>
<thead>
<tr>
<th>But what inputs are treated the same?</th>
</tr>
</thead>
<tbody>
<tr>
<td>![Image of beetles on a grid]</td>
</tr>
</tbody>
</table>
Intuition: Look for potential differences in behavior

We know the same test case doesn’t add info
We know many test cases may act the same
We don’t know what differences matter, but we can make some reasonable guesses ...

• Based on the problem specification
• Based on the types of data
• Based on how the program works
• Based on bugs we’ve seen before
Clues: Edge cases

while (high - low > ERROR_BOUND) {
    guess = (high + low) / 2.0;
    if (guess * guess > x) {
        // Too high
        high = guess;
    } else {
        // Too low
        low = guess;
    }
}
“Edge” condition

\[
\frac{x}{2} < \sqrt{x} \quad \text{and} \quad \frac{x}{2} > \sqrt{x}
\]
Some edgy input values ...

For an integer ...
   -1, 0, 1

For a string (text) ...
   "" (empty), "x" (one character)

Plus test some extreme values (e.g., a large integer) to cross edges
From the problem spec ...

The problem involves dates and leap years, so ...
Feb 28, Feb 29, Mar 1 are edgy dates
January 1 and December 31 are edgy dates
1999, 2000, 2001 are edgy years
... etc ...

I want to test every case treated specially by the program, every case treated specially in the spec, and identifiable “edges” between cases
For a palindrome checker

Input: A string
What cases can you derive from that?

Output: Is a palindrome or not
What cases can you derive from that?
Anything else? How?
For greatest common divisor

What are some good test cases? Why?
Automating your testing

You should test your program over and over, as you develop it.
If you have to do it by hand, you won’t.

But you’re a programmer. You can make it easier.
Approach 1: Input data set

Example: Palindrome checker reads a text file, checks whether each word is a palindrome.

Write a text file of test cases.
Save output to a file.
Compare actual to expected (important!)

You can write scripts for both parts, so text execution and judgment are easy.
Approach 2: Tests in the code

static void runPalindromeTest(String s, boolean expected) {
    boolean actual = isPalindrome(s);
    if (actual == expected) {
        System.out.println("(OK) tested string "
            + s + "" => " + actual");
    } else {
        System.out.println("FAILED TEST on " + s + "]");
        System.out.println("Expected " + expected + ", got " + actual);
    }
}

•
The test suite ...

```java
static void runPalindromeTests() {
    runPalindromeTest("", true);
    runPalindromeTest("a", true);
    runPalindromeTest("xx", true);
    runPalindromeTest("Xx", false);
    runPalindromeTest("abba", true);
    runPalindromeTest("aba", true);
    runPalindromeTest("abca", false);
    runPalindromeTest("abc", false);
}
```
And the results ...

$ java Palindrome
(OK) tested string "" => true
(OK) tested string 'a' => true
(OK) tested string 'xx' => true
(OK) tested string 'Xx' => false
(OK) tested string 'abba' => true
(OK) tested string 'aba' => true
(OK) tested string 'abca' => false
(OK) tested string 'abc' => false

OK, thanks for using Palindromes R Us
Tool support

**junit** is a popular tool for inserting test cases in Java code.
Similar in concept to what we just did.

Other tools create test cases from spreadsheets, mix-and-match data to create test cases, etc.
Summary

Testing can’t be perfect, but it can help
Random or haphazard testing is ineffective
  • Because the space of possible inputs is enormous, and the bugs are not spread evenly
Systematic testing uses inputs that “might be different”
  • Different treatment in spec, or program
  • Especially “edge” values and cases
Test early, test often
  • Automate your testing to make it practical