**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.
- Effective search for bugs to fix

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!

**2012: Alan Turing 100th birthday**

*University Theatre spring season 2013:*

**BREAKING THE CODE** by Hugh Whitemore
Based on the book *Alan Turing, The Enigma* by Andrew Hodges
Joseph Gilg directs: Hope Theatre ages 13+
May 30, 31, June 1, 6, 7, 8, 9 2013

An exceptional biographical drama about a man who broke too many codes: the eccentric genius Alan Turing played a major role in winning World War II by breaking the complex German code called Enigma. He was also the first person to conceive and describe computers. After the war he was put on trial for breaking another code - the taboo against homosexuality. This play is about who he was, what happened to him and why.
**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 $(P', P')$?
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 ≠ better

Suppose my first test of max is \( \text{max}(12,15) \)

Then I could run 5 more tests:
- \( \text{max}(10, 14) \)
- \( \text{max}(5, 88) \)
- \( \text{max}(13, 25) \)
- \( \text{max}(10, 17) \)
- \( \text{max}(100, 200) \)

Or I could run 1 more test:
- \( \text{max}(13, 7) \)

If more isn’t better, what is?

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

How about random testing?

Strategy: Generate random inputs, uniformly distributed

Example: Consider a buggy square root program

```java
Assumes \( x < y \implies \text{root}(x) < \text{root}(y) \)
```

What if we chose inputs randomly from the interval \( (0, 1000000000000) \)

Suppose we ran 1000000 test cases ... enough?

Buggy Square Root Finder

```java
while (high - low > ERROR_BOUND) {
    guess = (high + low) / 2.0;
    if (guess * guess > x) {
        // Too high
        high = guess;
    } else {
        // Too low
        low = guess;
    }
}
```
Too many possible inputs ...

But probably many are similar ...

Similar with respect to correctness ...

But what inputs are treated the same?
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

```c
while (high - low > ERROR_BOUND) {
    guess = (high + low) / 2.0;
    if (guess * guess > x) {
        // Too high
        high = guess;
    } else {
        // Too low
        low = guess;
    }
}
```

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., large values, small values) 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**

```python
def test_root(x):
x_root = root(x)
xmin = x_root - ERROR_BOUND
xmax = x_root + ERROR_BOUND
min_sq = min(xmax * xmax, xmin * xmin)
max_sq = max(xmax * xmax, xmin * xmin)
if min_sq <= x and max_sq >= x:
    print("Close enough: root(" + x + ") is approx. ", x_root,
        "+/-", max_sq - x)
else:
    print(*** ERROR, got range ", xmin, " to ", xmax,
    " for root(" + x + ") = ", x_root)
```

**The test suite ...**

```python
test_root(2.0)
test_root(4.0)
test_root(3.0)
test_root(100.0)
test_root(0.25)
test_root(0.0)
test_root(1024.0 * 1024.0)
```

**And the results ...**

```
$ python3 root.py
Close enough: root(2.0) is approx. 1.4142135679721832 +/- 2.9867933681870795e-07
Close enough: root(4.0) is approx. 2.0000000298023224 +/- 5.192093057715397e-07
Close enough: root(3.0) is approx. 1.732050821185112 +/- 3.9357819536789407e-07
Close enough: root(100.0) is approx. 10.000000009313226 +/- 2.186264509873581e-06
*** ERROR, got range 0.2499999701976776 to 0.2500000701976776 for root(0.25) = 0.2499999701976776
*** ERROR, got range -1e-07 to 1e-07 for root(0.0) = 0.0
Close enough: root(1048576.0) is approx. 1024.0000000298023 +/- 0.00026583531871438026
```
**Tool support**

**JUnit** is a popular tool for inserting test cases in Java code.

**unittest** is a similar module for Python

Similar in concept to what we just did.

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

Handy, but not really necessary

---

**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