What’s the goal?

What we want:
Convincing evidence that our program works correctly, for all inputs.

or

Effective search for bugs to fix

Can we get it?
Why or why not?

Testing

How can I choose good test cases?
How can I make testing more efficient?

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 function, in my program P’( program Q, 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.

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.

Um, so then what?

Testing can never prove program correctness

Shall we give up and not test our programs?

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  \( \max(12,15) \)

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”?)

How about random testing?

Strategy:  Generate random inputs, uniformly distributed

Example:  Consider a buggy square root program

\[ \text{Assumes } x < y \implies \sqrt{x} < \sqrt{y} \]

What if we chose inputs randomly from the interval (0, 1000 000 000 000)

Suppose we ran 1 000 000 test cases ... enough?

Buggy Square Root Finder

\[
\text{while (high - low > ERROR_BOUND) }
\begin{align*}
\text{guess} & = (\text{high} + \text{low}) / 2.0; \\
\text{if (guess * guess > x) }
\begin{align*}
& \text{// Too high} \\
& \text{high} = \text{guess};
\end{align*}
\end{align*}
\begin{align*}
\text{else }
\begin{align*}
& \text{// Too low} \\
& \text{low} = \text{guess};
\end{align*}
\end{align*}
\]
\]
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

```java
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 of days between dates ...
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

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
See the test code in game_dict.py and test_harness.py:
“Test driven development”:
- Write test cases, then write code
- The tests are an asset for future changes.
(Mea culpa: I should have re-run the test cases on my revised version of game_dict.py)
If I had done it right ...

```python
solved > python3 game_dict.py
Passed -- First word in dictionary (alpha) result: 1
Passed -- Last word in dictionary (omega) result: 1
Passed -- Within dictionary (beta) result: 1
Passed -- Within dictionary (delta) result: 1
Passed -- Prefix of first word (al) result: 2
Passed -- Prefix of last word (om) result: 2
Passed -- Prefix of interior word (bet) result: 2
Passed -- Prefix of interior word (gam) result: 2
Passed -- Prefix of interior word (dell) result: 2
Passed -- Before any word (aardvark) result: 0
Passed -- After all words (zephyr) result: 0
Passed -- Interior non-word (axe) result: 0
Passed -- Interior non-word (carrot) result: 0
Passed -- Interior non-word (hagiography) result: 0
```  

***FAILED***  
First word in dictionary (alpha) Expected: | 1 | but got | 0 |
***FAILED***  
Last word in dictionary (omega) Expected: | 1 | but got | 0 |
Passed -- Short word omitted (beta) result: 0

---

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