Testing

We know the code is bad but where?

Joseph Hallett

March 10, 2025

University of BRISTOL

We're almost done!

This is the last lecture!

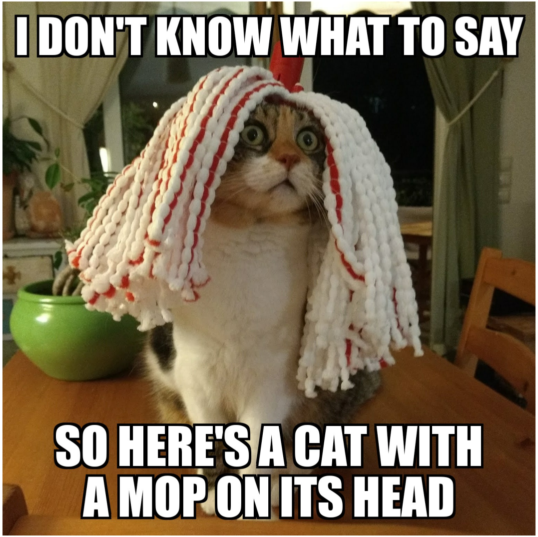
- Well done!
- We're nearly all free!

Repeated questions...

Over the course of this unit people have asked the same question again and again

How do I know if I got it right?

IDK?



I wish I could give you a better answer...

Truth is working out whether something is right or wrong is really hard...

If you've asked me or the TAs before you've probably been told

- Well does it work?

Example

Write an algorithm to find the mean of an array.

```
int mean(int *data, size_t len) {
  int sum = 0;
  for (size_t i = 0; i < len; i++) sum += data[i];
  return sum / len;
}

int x[5] = { 1, 2, 3, 4, 5 };
printf("%d\n", mean(x, 5));
```

3

Example 2

What about now?

```
int mean(int *data, size_t len) {
  int sum = 0;
  for (size_t i = 0; i < len; i++) sum += data[i];
  return sum / len;
}

int x[4] = { 1, 2, 3, 4 };
printf("%d\n", mean(x, 4));
```

2

Ruh roh...

Example 3

```
/** Find the median element */
float mean(int *data, size_t len) {
  float sum = 0.0f;
  for (size_t i = 0; i < len; i++) sum += (float) data[i];
  return (float) sum / (float) len;
}

int x[4] = { 1, 2, 3, 4 };
printf("%d\n", mean(x, 4));
```

4

Ruh roh...

Example 4

```
/** Find the median element */
float mean(int *data, size_t len) {
  float sum = 0.0f;
  for (size_t i = 0; i < len; i++) sum += (float) data[i];
  return (float) sum / (float) len;
}

int x[4] = { 1, 2, 3, 4 };
printf("%f\n", mean(x, 4));
```

```
2.5
```

I mean still ruh-roh...

Example 5

```c
/** Find teh mean element */
float mean(int *data, size_t len) {
  float sum = 0.0f;
  for (size_t i = 0; i < len; i++) sum += (float) data[i];
  return (float) sum / (float) len;
}

int x[] = { 1, 2, 3, INT_MAX, INT_MAX };
printf("%d\n", INT_MAX);
printf("%f\n", mean(x, 5));
```

2147483647
858993472.0

Choices, choices...

```
/** Find the mean element */
float mean1(float *data, size_t len) {
  float sum = 0.0f;
  for (size_t i = 0; i < len; i++) sum += data[i] / len;
  return sum;
}

float mean2(float *data, size_t len) {
  float sum = 0.0f;
  for (size_t i = 0; i < len; i++) sum += data[i];
  return sum/len;
}
```

Which is better?

- Why?
  - IEEE754 floating point isn't exact...
  - Which way is going to accrue less error over time?
  - Which way is going to be able to handle more data?

So what?

The point I am trying to make is that even when your code is right…

- It is still always wrong
- We cannot win
- We are doomed
- We should look for alternative careers
  - I recommend pottery
  - Mud can't hurt you like a computer can

No, we are engineers...

The artists and the philosophers had the right idea

Instead of focusing on absolute truth...

- Lets focus on knowing the limitations of our work
  - When is our code going to be wrong?
  - When is it going to fail?
  - When is it going to be acceptably wrong...
  - When have we made it worse?

Testing

Our strategy for dealing with error is called testing

- Were going to go over some strategies for doing it in the lecture

For some people this is a job…

Whole branch of software engineering called Quality Assurance that deals with making sure that the tests pass and requirements are met.

- Can make you somewhat unpopular…
- I love it though…
  - …can be a good entry point to security work

QA people should probably test their tests...

Brenan Keller
@brenankeller

A QA engineer walks into a bar. Orders a beer. Orders 0 beers. Orders 99999999999 beers. Orders a lizard. Orders -1 beers. Orders a ueicbksjdhd.

First real customer walks in and asks where the bathroom is. The bar bursts into flames, killing everyone.

1:21 PM · 30 Nov 18

Note for me so I know what to write about

Today we're going to cover:

- Asserts
- Unit tests
- Behavioural testing
- Property testing
- Fuzz testing
- Formal proof

Assert

Probably the simplest mechanism you will ever see.

- Every language has some version of it...
- In C it's in `<assert.h>`

You write your check in the `assert()` statement

- If it's not true your program crashes

Example

Exponentiation by squaring...

```
int power(int base, int exponent) {
  if (exponent == 0) return 1;
  if (exponent == 1) return base;
  if (exponent & 1) return base * power(base*base, (exponent-1)/2);
  return power(base*base, exponent/2);
}

printf("%d\n", power(4,7));
```

16384

Great seems to work...

For the sake of argument…

```c
int power(int base, int exponent) {
  if (exponent == 0) return 1;
  if (exponent == 1) return base;
  if (exponent & 1) return base * power(base*base, (exponent-1)/2);
  return power(base*base, exponent/2);
}

printf("%d\n", power(4,7));
printf("%d\n", power(2,-3));
```

`Segmentation fault.`

Looks like we missed something…

Why not just make it unsigned?

```
int power(int base, int exponent) {
 assert(exponent >= 0);
 if (exponent == 0) return 1;
 if (exponent == 1) return base;
 if (exponent & 1) return base * power(base*base, (exponent-1)/2);
 return power(base*base, exponent/2);
}

printf("%d\n", power(4,7));
printf("%d\n", power(2,-3));
```

```
assert-is-cool.c:11: power: Assertion `exponent >= 0' failed.
```

Is this better than a segfault?

Problems...

Assert is (usually) just a macro used for debugging...

- Wouldn't want it in production code... might cause a crash
- So usually it will be removed from release code

If you compile with -DNDEBUG=1 the `assert()` will be removed

```
int power(int base, int exponent) {
 assert(exponent >= 0);
 if (exponent == 0) return 1;
 if (exponent == 1) return base;
 if (exponent & 1) return base * power(base*base, (exponent-1)/2);
 return power(base*base, exponent/2);
}

printf("%d\n", power(4,7));
printf("%d\n", power(2,-3));
```

```
Segmentation fault.
```

(Could be worse... it used to be if you didn't compile with -DDEBUG=1 the `assert()` would get removed...)

- Still is for some languages!

So what's the point?

Assert is good for testing programmer assumptions when writing your code...

- But it isn't good for checking that your code always works
- Ideally we want a series of checks against known good values and results
  - (and maybe against known causes of failure)

Unit testing!

Unit testing tests your code against a series of known inputs and checks the results.

- Great to see if your code is getting better or worse
  - You pass or fail more tests
- Fancy frameworks for every language
  - But you don't need them… can be as simple as an option that runs your program in a test mode with a bunch of `if` statements.

If you are writing code for production this is the minimum you will be expected to do.

- Write tests
- Check it passes them
- Check no regressions
  - Passing tests now fail

Python

If you write Python there's a lovely test framework called `pytest`...
Run the tests with `python -m pytest`

```
import pytest

def power(base, exponent):
   assert(exponent >= 0)
   if exponent == 0:
       return 1
   if exponent == 1:
       return base
   if exponent & 1 == 1:
       return base * power(base*base, (exponent-1)//2)

class TestPower:
   def test_knowngood(self):
      assert power(4,7) == 16384

   def test_knownerror(self):
      with pytest.raises(AssertionError):
         power(2,-3)
```

And it says...

```
============================= test session starts ==============================
platform linux -- Python 3.13.2, pytest-8.3.4, pluggy-1.5.0
rootdir: /home/goblin/Repos/Talks/Software-Tools/2024/20-testing
plugins: asyncio-0.24.0, mock-3.14.0
asyncio: mode=Mode.STRICT, default_loop_scope=None
collected 2 items

power.py ..                                                              [100%]

============================== 2 passed in 0.01s ===============================
```

Everything passed! Our code must be good!

Bad Python

Let's see what happens when it fails...

- Rerun the tests with `python -m pytest`

```
import pytest

def power(base, exponent):
   assert(exponent >= 0)
   if exponent == 0:
       return 1
   if exponent == 1:
       return base
   if exponent & 1 == 1:
       return base * power(base*base, (exponent-1)//2)

class TestPower:
   def test_knowngood(self):
      assert power(4,7) == 16384

   def test_knownerror(self):
      with pytest.raises(AssertionError):
         power(2,-3)

   def test_failing(self):
      assert power(1,100) == 5
```

Whoopsie! Made an Oopsie!

```
============================= test session starts ==============================
platform linux -- Python 3.13.2, pytest-8.3.4, pluggy-1.5.0
rootdir: /home/goblin/Repos/Talks/Software-Tools/2024/20-testing
plugins: asyncio-0.24.0, mock-3.14.0
asyncio: mode=Mode.STRICT, default_loop_scope=None
collected 3 items

power2.py ..F                                                            [100%]

=================================== FAILURES ===================================
____________________________ TestPower.test_failing ____________________________

self = <power2.TestPower object at 0x72b704b16650>

    def test_failing(self):
>       assert power(1,100) == 5
E       assert None == 5
E        + where None = power(1, 100)

power2.py:21: AssertionError
=========================== short test summary info ============================
FAILED power2.py::TestPower::test_failing - assert None == 5
========================= 1 failed, 2 passed in 0.01s ==========================
```

Oh no! Better go explore that...

Unit tests are great!

But there's a problem...

- They're written in code

Not everyone can write code...

Not everything is well described by code

Seriously?

Yes we all can write code

- I would hope

But your manager may not

- I mean a good manager should...
- Just maybe not in a long time...

The person paying you may not

- I mean, they're paying you to write their code...

Specification, specification, specification

When we specify what an application should do we don't (often) write it in code.
For example:

> You should write a function raise a base to an integer exponent. When given negative integers an error should be triggered. When raising 4 to the power 7 then the result should be 16384.

Managers and clients like this sort of thing

- It makes sense to them even without writing code

Also makes sense for more nebuluous tests, e.g.

> When a minesweeper tile is clicked it is revealed. If a bomb is revealed then the game is over.

Acceptance testing

Wouldn't it be nice if we could take the specification that we're being asked to implement and test against that!

- Then we wouldn't have to write our own unit tests
- If we forget something it's the bosses fault!
- We just did what they told us!

Cucumber

There are many acceptance testing frameworks...

- But I quite like Cucumber
- Bindings for a tonne of languages
- `https://cucumber.io`

To use it:

- You write a spec in natural language (a feature)
- You write syntax rules to translate the spec into code
- You check your tests pass

Some conventions for where things go

- Read a tutorial or look at the code in the repo

Exponentiation as a feature

Lets try and use Cucumber with JavaScript!

- We define our tests in `features/exponentiation.feature`

```
Feature: Exponentiation works.
  Our power function should be able to calculate (integer) exponents.

  Scenario: 4 to the power of 7 is 16384.
    Given a base of 4
    When taking it to the power of 7
    Then the answer should be 16384.
```

But what does it mean?

We define syntax rules in `features/step_definitions/stepdefs.js`:

```
import { power } from "../../power.js";

import assert from 'assert';
import { Given, When, Then } from '@cucumber/cucumber';

Given('a base of {int}', function (int) {
  this.base = int
});

When('taking it to the power of {int}', function (int) {
  this.exponent = int
});

Then('the answer should be {int}.', function (int) {
  this.answer = int
  assert.strictEqual(power(this.base,this.exponent),this.answer);
});
```

And given our JS power function...

Hey lets go iterative this time...

```
export function power(base, exponent) {
  let answer = 1;
  while (exponent > 0) {
    if (exponent & 1) {
      answer = answer * base;
      exponent--;
    }
    base = base * base;
    exponent = exponent / 2;
  }
  return answer;
}
```

Wahey!

```
$ npm test

> power@1.0.0 test
> cucumber-js


...

1 scenario (1 passed)
3 steps (3 passed)
0m00.008s (executing steps: 0m00.000s)
```

Everything passes!

- Our bosses had to write the tests
- But we had to translate them...

While we're collecting implementations

We seem to be implementing exponentiation in a bunch of languages...

In Prolog (for Casey)

```
power(B,E,R) :- power(B,E,1,R), !.
power(_,0,A,A).
power(B,E,A,R) :-
   1 is E mod 2,
   !,
   B1 is B * B,
   E1 is div(E - 1, 2),
   A1 is B * A,
   power(B1,E1,A1,R).
power(B,E,A,R) :-
   0 is E mod 2,
   !,
   B1 is B * B,
   E1 is div(E, 2),
   power(B1,E1,A,R).
```

And in Haskell

```haskell
power :: Int -> Int -> Int
power = recur 1
 where
   recur acc _ 0 = acc
   recur acc b 1 = b*acc
   recur acc b e
     | odd e  = recur (b*acc) (b*b) ((e-1)`div`2)
     | otherwise = recur acc (b*b) (e`div`2)
```

Tail recursive this time!

Haskell is a lazy programming language

Meaning it doesn't evaluate things unless it has to
I am also lazy...

- I want to test this function but...
- I don't want have to write tests

Property testing

Instead of writing individual tests... lets try and capture global properties.

- For example... if we take a number to the power of any number greater than 1... it gets bigger.

We want to check that this property holds for any input

- We don't want to have to generate function inputs ourselves

QuickCheck

Is a property testing framework that does just this.

- Originally for Haskell, now ported to everything

We write property tests, it checks random values to see if it holds

```
import Test.QuickCheck

power :: Int -> Int -> Int
power = recur 1
  where
    recur acc _ 0 = acc
    recur acc b 1 = b*acc
    recur acc b e
      | odd e  = recur (b*acc) (b*b) ((e-1)`div`2)
      | otherwise = recur acc (b*b) (e`div`2)

{- For any exponent greater than 0, b`power`e > b. -}
propertyIncreases b e
  | e > 0 = b`power`e > b
  | otherwise = True
```

```
ghci> quickCheck propertyIncreases
*** Failed! Falsified (after 18 tests and 5 shrinks):
0
1
```

Edge cases suck

Aha… if our base is 0 then this doesn't hold

- And actually if our base is negative and our exponent is odd…

```
import Control.Applicative

{- For any base and exponent greater than 0, b`power`e > b. -}
data PositiveArgs = PositiveArgs Int Int
instance Show PositiveArgs where
  show (PositiveArgs b e) = show b ++ "^" ++ show e ++ "␣=␣" ++ show (b`power`e)

instance Arbitrary PositiveArgs where
  arbitrary = PositiveArgs <$> arbitrary `suchThat` (>0)
                           <*> arbitrary `suchThat` (>0)
propertyIncreases' (PositiveArgs b e) = b`power`e >= b
```

```
ghci> quickCheck propertyIncreases'
*** Failed! Falsified (after 19 tests):
8^21 = -9223372036854775808
```

Fixed width integers suck (if you care about correctness)

```haskell
power :: Integer -> Integer -> Integer
```

```
ghci> quickCheck propertyIncreases''
+++ OK, passed 100 tests.

ghci> quickCheckWith stdArgs { maxSuccess = 1000000 } propertyIncreases''
+++ OK, passed 1000000 tests.

ghci> let prop = \(PositiveArgs' b e) -> (b`power'`e) == (b^e)
ghci> quickCheckWith stdArgs { maxSuccess = 1000000 } prop
+++ OK, passed 1000000 tests.
```

I'm kinda convinced now that my exponentiation code works

- For positive integers...

But you still have to come up with the properties to test...

So what properties might be good to test?

- Equivalence to old code?
- Invertability?
  - If I calculate the square of the square root of a number i ought to get back the same number
  - If I print a data structure, I ought to be able to read in the same data structure
    - This catches lots of parsing bugs
  - No input to my program should crash it...

No crashing allowed

If a program crashes you can sometimes do horrible things

- See the debugging lecture

We would like to find inputs that trigger crashes…

Fuzzers Fuzz test programs

Try random inputs... see what triggers a crash

- Try corrupt inputs... see what logic paths get hit by instrumenting the code
- Try and find inputs that trigger every path through every conditional in a program!

American Fuzzy Lop(++)

- Is really good at this...
- Other fuzzers exist...
- `https://apfplus.plus`

Right!

So far we've talked about testing...

We've talked about

- Assertions
- Unit tests
- Behavioural tests
- Property tests
- Fuzz tests

If you are a normal person this is where you stop...

The nuclear option

There is one other way to make sure our code works...

- We could formally prove that it is correct
- That it will never crash (under any input)
- That it will always find the answer (in a reasonable time)

To do this we have to use maths

Formal methods

In general reasoning about programs is impossible

- Halting problem means you can't tell if a program will complete or not

But, given enough effort, you can prove somethings about a program

- If you have a mathematical model of a computer you can prove an algorithm's correctness...
- If you have a mathematical model you can prove equivalence between algorithms...
- If you have a mathematical model you can establish accurate bounds for how many operations it will take to complete...
- If you can translate that model into code in a sound way then you have a path to generating correct programs

No need to test if the code is right!

"GIVEN ENOUGH EFFORT"

The effort required is monumental.

- Routinely a PhD's worth of effort for a medium sized problem

That's not to say that it occasionally isn't worth it...

- Astronauts and aircraft are crazy expensive... worth proving control systems always work
- If you're Amazon, its worth proving that all AWS networking will have guaranteed uptime and performance metrics
  - They do this out in Bath—go see Rod Chapman talk if you get a chance!
- seL4 is an OS that you cannot hack
  - Monumental effort, but worth it for roots of trust that we know won't break
- Cryptography and the modern web works
- So does memory allocation

So what do you need to do this?

Lean 4 is a proof assistant

- It knows about mathematical reasoning…
- It can check your proofs and help you write them

We could prove that our exponentiation by squaring algorithm is equivalent to the repeated multiplication one…

- But that is way beyond my ability with these tools
- Not beyond everyone in the departments ability though…

We should play with this in the lab ;-)