CSE 374: Lecture 18

Testing

Writing Good Code

1. **Choose the right language.** If possible, choose languages that prevent certain types of bugs. For example, if you don't need the lower-level performance or control, pick a language like Java rather than C to avoid memory-related bugs.
2. **Think before you code.** Understand how the program will work before implementing it. Draw data structures and how you will modify them over the course of the program. Write pseudocode and consider all of the different cases you might encounter.
3. **Make defects visible.** Use "assert" statements and exceptions (if they exist in your language) to catch errors safely.
4. **Test the code.** Ensure proper behavior by writing another program to exercise the code completely.
5. **Debugging.** Unavoidable. Examples of debugging include adding print statements, gdb, valgrind or other tools, or adding more test cases.

"There are two ways of constructing a software design:

- One way is to make it so simple that there are obviously no deficiencies, and
- the other way is to make it so complicated that there are no obvious deficiencies.

The first method is far more difficult."

Sir C. A. R. Hoare
1980 Turing Award winner
Invented "quicksort"

What is testing?

Software testing evaluates the **effectiveness** of a software solution

★ Systematic

★ Objective

Effectiveness:

Does what it is supposed to do

Fails gracefully

Uses memory safely and efficiently

Computes in reasonable time

https://en.wikipedia.org/wiki/Software_testing

"Test your software or your users will."
Hunt & Thomas -- The Pragmatic Programmer

Testing Challenges

- Testing is very limited and difficult
  - Small number of inputs
  - Small number of calling contexts, environments, compilers, …
  - Small amount of observable output
  - Requires more work to implement test code
  - It is hard to imagine all the possible flaws in your own code
- Standard coverage metrics (statement, branch, path) are useful but only emphasize how limited it is.

"Program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence."
Edsger Dijkstra -- 1972 Turing Award lecture

Testing Responsibility

~ Every person writing software should be capable and responsible for testing it.

(It's a highly challenging job: Some specialize in it.)

How much testing?

- Often use ‘coverage metrics’:
  - Percentage of code that is covered by testing mechanisms.
- Nice for goal setting
- Not sufficient for guaranteeing problem free code.

Coverage

F => a OR b where only ‘0’ is false.

```
int f(int a, int b)
{
  int ans = 0;
  if(a)
    ans += a;
  if(b)
    ans += b;
  return ans;
}
```

Statement coverage: `f(1,1)` sufficient

Branch coverage: `f(1,1)` and `f(0,0)` sufficient

Path coverage: `f(0,0)`, `f(1,0)`, `f(0,1)`, `f(1,1)` sufficient

But even the example path-coverage test suite suggests f is a correct “or” function for C; it is not. `f(-1,1)`

Types of Testing

- **Unit testing** looks for errors in subsystems (functions, file, class) in isolation.
  - Small number of things to test.
  - Easier to find faults when errors occur
  - Can test many components in parallel
- **Integration testing** tests the interactions between subsystems of code. Can catch errors that unit tests will not surface, since while each subsystem may be correct separately, they may not work together properly.
- **Continuous integration testing** is automatically running integration tests and unit tests on every commit.
- **Performance testing** measures performance: how much memory it takes, how fast it is, and how much CPU it uses. Performance tests can reveal errors in how the program's resources are managed - bottlenecks in the system.
- **Reliability testing** looks at performance when a system is put under heavy and consistent use.
- **Security testing** is looking for vulnerabilities that could be abused.
- **Usability testing** is high level testing to look at whether software responds to typical user interactions correctly.

Black-Box v. White-Box testing

Black box testing allows the tester to specify input, and see output, with no vision into the actual code.

Depends only on specification of code function

Won't get stuck implementing only the logic in the code

White box testing is written with a full view of the code, tester specifies test cases to exercise known cases in the code.

Write tests for all the weird corner cases in code.

Check the edge cases - loop boundaries, empty or full data structures, max-values
Check for response to different types of input, and unexpected input

Unit Testing

Test small components of code individually

Basic approach - ‘assert’ desired performance.

(Note: Use conditional compilation
`Ifdef NODEBUG`
Plus macro
`#define assert(ignore)((void) 0)`
To compile without test code.)

```
#include <assert.h>
#include <stdlib.h>
#include "f.h"

// Assert statements will fail with a message
// if not true.
int main(int argc, char** argv) {

    assert(!f(0, 0));  // Test 1: f(0,0) => 0
    assert(f(0, 1));   // Test 2: f(0,1) => T
    assert(f(1, 0));   // Test 3: f(1,0) => T
    assert(f(1,1));    //  Test 4: f(1,1) => T

    // Test case 5: f(-1,1) => not-0
    assert(f(-1,1));
    return EXIT_SUCCESS;
}
```

```
OUTPUT >> program: f.c:9: main: Assertion
`!f(0,0)' failed.   Abort (core dumped)
```

Stubbing

Unit testing looks at one component at a time

Provide ‘stubs’ to give just enough code for executing the desired unit.

After unit testing succeeds, proceed with integration testing (combining units) and system testing (the entire product).

Testing frameworks exist to make this easier: *explore and use them!*

- Instead of computing a function, use a small table of pre-encoded answers
- Return default answers that won’t mess up what you’re testing
- Don’t do things (e.g., print) that won’t be missed
- Use an easier/ slower algorithm
- Use an implementation of fixed size (an array instead of a list?)
- Test with hard coded input.

Eat your vegetables

- Make tests
  - Early
  - easy to run (e.g., a make target with an automatic diff against sample output)
  - that test interesting and well-understood properties
  - that are as well-written and documented as other code
- Write the tests first! (seems odd until you do it)
- Write much more code than the “assignment requires you turn-in”
- Manually or automatically compute test-inputs and right answers?
- Write regression tests and run on each version to ensure bugs do not creep in for stuff that “used to work”.

Homework 5

Idea:

Write source code for a tree

Write tests to make sure the tree does what you want

ONLY THEN

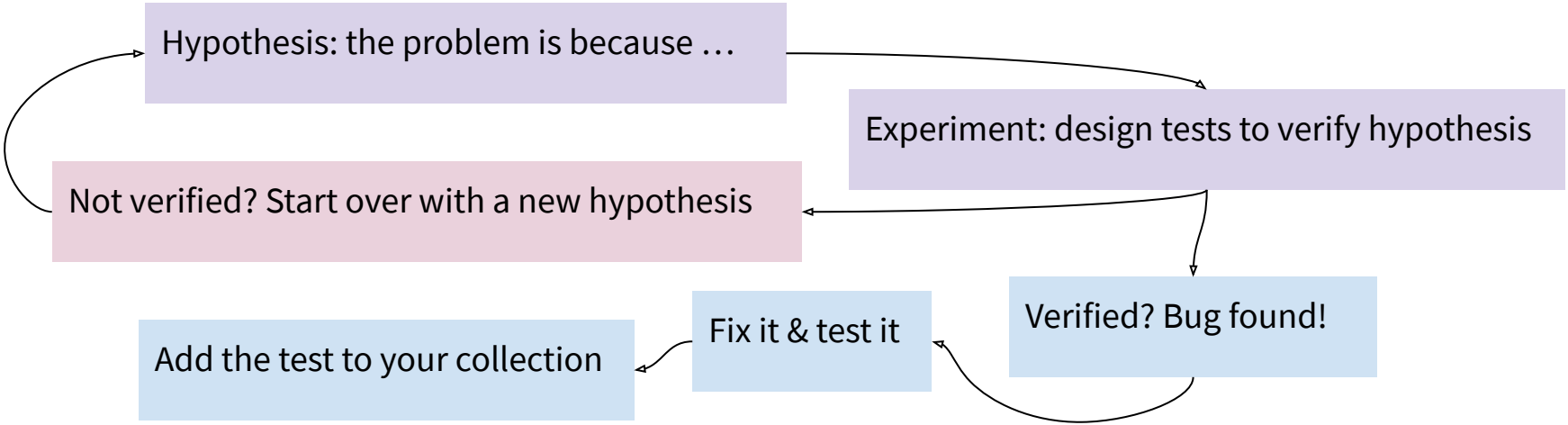
Write source code to use the tree as a trie.

What tests can you write?

- **Do letters become the correct number?**
- **Is memory allocated & deallocated correctly?**
  - **Hint: valgrind**

Testing to Debug

- Have bug? Find the cause and fix it
- A bit of an art, but, Treat debugging as a scientific experiment:



"Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it."
Brian Kernighan -- Wrote THE BOOK on C (our book!)

HW6: Bench

Exercises your code: can use it as a test as you build up the other functions