Review from 120: A Necessary Skill: Testing Code

How do you know that your program works?

There’s only one correct answer: test it!*

Brooks’ Rule of Thumb
◦ 1/3 planning and design
◦ 1/6 writing the program
◦ 1/2 testing

Just because your program compiles
does not mean your program works!

*Becoming a good tester will take years.
Don’t worry if it seems tough.

Testing Goal: Identify and Eliminate Unwanted Behavior

Lots of room for error in difficult problems:
◦ software specification,
◦ writing code based on a specification, and
◦ coming up with strategies for testing.

Be careful about getting too wrapped up
in the theory of testing.

Instead, focus on the goal: to identify and eliminate unwanted behavior.
Common Strategies for Software Testing

1. Test-driven development:
   ◦ write tests first, then
   ◦ write code to pass tests.
   ◦ Don’t write code unless a test demands it.

2. Clear/white-box testing:
   ◦ write code based on specification, and
   ◦ write tests based on code.
   Both are useful, and both can be abused.

Extreme Case of a Real Pitfall: Test-Driven Development

```c
switch (test_number) {
    case 0: printf ("Answer 0");
             break;
    case 1: printf ("Answer 1");
             break;
    // and so forth...
}
```
Designed to pass tests, not to do something.

Real Pitfall: Clear Box Testing Can Miss the Point

Tests based only on code, so
   ◦ if developer forgets some functionality,
   ◦ there are no tests for it
   ◦ (it’s not in the code!).
To be fair, clear box testing is
   ◦ usually used for unit testing (low-level component testing);
   ◦ integration testing (of the whole program)
     is supposed to handle missed cases.

Use Both Approaches in Practice

Write tests
   ◦ based on intended behavior
   ◦ representing corner cases when possible.
Write code and use tests to debug.
Write more tests
   ◦ based on code
   ◦ to ensure that all code works as intended
   ◦ and is necessary.
**Real Pitfall: Unlikely Scenarios**

**Code can also “hide”:**
- possibly unnecessary and/or broken, but
- **difficult to execute**, so no tests cover it.

Methodologies exist for this type of problem:
- **make scenarios likely** in a debugger
- **inject failures** externally (a tool causes the unlikely scenarios), and
- **concolic testing tools** try to find inputs that cover all paths.

These techniques are beyond our class’ scope.*

*The automated feedback tool uses concolic testing.

---

**Performing Tests is Easier if One Thinks Ahead**

**How does one perform tests?**

*Best answer:*
- **design the code to be easily testable!**

One can also
- use scripts to transform code and expose details, but
- it’s better to make the tester’s life easy.
- Good testing is challenging enough.

**If you write some tests first, you’ll be forced to write more testable code.**

---

**Best Practice: Write Code that Uses the Code Under Test**

**Option 1:**
- external code calls code to be tested and
- inspects state/results.

**Best practice,** but sometimes hard to test thoroughly.

---

**Also Common: Add Debug Code to Code Under Test**

**Option 2:**
- **extend** code to be tested with **debug code,**
- then, during development and testing,
- use debug version (with extra code).

**Common approach,** but code usually shipped with debug code left in place—removing it is questionable.
Extremely Bad Practice: Test Something Different

Option 3?
- Write and test a separate version,
- then throw it away and ship the original.

Just as Bad, But Harder to Spot

What if we do it this way?
if (debug_mode) {
    separate debug version
} else {
    code to be tested
}

Less clear, but still dumb.

A Little Help, Please?

I have a question...
How are bytes stored in memory?

For example,
- when a processor writes
- a 32-bit register
- to 8-bit-addressable memory,
- what happens?

Two Choices for Endianness

Here's a register.
Here's memory.
Which way are the bytes written?
The upper form is called big endian.
The lower form is called little endian.
Endianness Depends on the ISA

So, again:

**How are bytes stored in memory?**

Loads and stores use the same approach, so who cares?

Choice depends on the ISA:
- some are little endian,
- some are big endian, and
- some support both (not at the same time).

Communication Requires a Common Approach

What happens if
- a big endian machine
- sends a stream of bytes
- to a little endian machine?

Oops.

One of the two machines must swap the order of the bytes.

Little Endian Hosts Must Swap Bytes to Use the Internet

When the Internet was developed,
- big endian ISAs dominated computing, so
- the Internet uses big endian.
- Protocol data must be in big endian order
- or a machine’s packets will be dropped.

Little endian ISAs (like x86)
- must swap the order of bytes
- to use the Internet protocols!

Reverse Order of Bytes to Swap Endianness

It’s not so bad...*

Just swap the order one byte at a time.

*When Intel, Microsoft, and Compaq cooperated to produce a standard for desktop/server room networking, the protocols were little endian. Go figure.
Let's Write `ntohl` as an Example

Standard library function `ntohl`
- "net to host long"
- *swaps bytes in a 32-bit value.*

Let's write it and test it.

Here's a sample function signature:

```c
int32_t htonl (int32_t arg);
```

Shift and Copy One Byte at a Time

```c
int32_t htonl (int32_t arg)
{
    int32_t res;
    res = (arg << 24);
    res |= ((arg & 0xFF00) << 8);
    res |= ((arg >> 8) & 0xFF00);
    res |= (arg >> 24);
    return res;
}
```

Let's Use Undergraduates to Write Tests

How about …
- each of you writes 125 tests
- along with the correct answer for each?
- Together we’ll have about **5,000 tests**.

Seem expensive?
That’s why I want to use undergraduates!
An engineer costs 5-10× more!
Maybe we should think harder?

Test All Possible Inputs?

Another idea: how about …
- we find a way to come up with answers
- and we **test all possible inputs**
- *(there are only 2^{32})*.

**Generally not the best strategy.**

Has the 64-bit adder in your laptop
- been tested that way?
- Even at 10 GHz, testing 2^{129} input combinations
- takes over 2 × 10^{21} years. So … no.
Random Testing Using Problem-Specific Features

Did you notice that, for any \( x \),
\[
\text{ntohl (ntohl (x))} == x
\]

We can use
- a random number generator (\( \text{rand} \))
- to make up 1,000,000 values of \( x \)
- (more than 5,000!)
- and test each one
- with no need for a human.

Is that good enough?

How About One More Human-Designed Test?

\( \text{rand} \) generates non-negative values (32-bit numbers in the range \([0, 2^{31} - 1]\)).

As a graduate student, my desktop's IP* address was 128.32.36.37.

Let's try that: 0x80202425.

Otherwise, we may not have a negative case...

*Internet Protocol; specifically, IPv4.

Shift and Copy One Byte at a Time

```c
int32_t ntohl (int32_t arg)
{
    int32_t res;
    res = (arg << 24);
    res |= ((arg & 0xFF00) << 8);
    res |= ((arg >> 8) & 0xFF00);
    res |= (arg >> 24);
    return res;
}
```

So far, so good!

```
arg: 0x80202425
res: 0x25000000
```

Shift and Copy One Byte at a Time

```c
int32_t ntohl (int32_t arg)
{
    int32_t res;
    res = (arg << 24);
    res |= ((arg & 0xFF00) << 8);
    res |= ((arg >> 8) & 0xFF00);
    res |= (arg >> 24);
    return res;
}
```

So far, so good!

```
arg: 0x80202425
res: 0x25240000
```
int32_t ntohl (int32_t arg)
{
    int32_t res;
    res = (arg << 24);
    res |= ((arg & 0xFF00) << 8);
    res |= ((arg >> 8) & 0xFF00);
    res |= (arg >> 24);
    return res;
}

A Correct Variant of ntohl

uint32_t ntohl (uint32_t arg)
{
    uint32_t res;
    res = (arg << 24);
    res |= ((arg & 0xFF00) << 8);
    res |= ((arg >> 8) & 0xFF00);
    res |= (arg >> 24);
    return res;
}

We told the compiler that we wanted an arithmetic right shift.
An Example of White Box Testing

Let's do another example.

Or, rather, let's recall another example.

The following slides are from ECE120.

Our Next Program Calculates the Roots of a Quadratic

Remember the equation?

\[ F(x) = Ax^2 + Bx + C \]

has roots \((F(x) = 0)\) at

\[
X = \frac{-B \pm \sqrt{B^2 - 4AC}}{2A}
\]

where \(\sqrt{N}\) is the square root of \(N\).

Short Version of the Quadratic Code

/* solution of the quadratic
   equation ax^2+bx+c=0 adapted from
   V. Kindratenko's notes on
   30 August 2016. */
int main()
{
    float a, b, c; /* coefficients */
    float D; /* discriminant */
    float x1, x2; /* solution(s) */

Read Coefficients and Explain the Program

// Get coefficients.
printf("Enter a, b, and c: ");
if (3 != scanf("%f %f %f", &a, &b, &c)) {
    // error message
    return 3; // Program failed.
}
printf("Solving equation ");
printf("%fx^2+%fx+%f=0.\n", a, b, c);
Positive Discriminant Implies Two Real Roots

// Compute discriminant.
D = b * b - 4 * a * c;
// Compute solution.
if (0 < D) { // Two real roots exist.
    x1 = (-b + sqrtf (D)) / (2 * a);
    x2 = (-b - sqrtf (D)) / (2 * a);
    printf ("x1=%f, x2=%f\n", x1, x2);
}

Handle Cases of One and Zero Real Roots

else if (0 == D) { // only one root
    x1 = -b / (2 * a);
    printf ("x=%f\n", x1);
} else {
    printf ("No real roots\n");
}
// End program
return 0;

Every Statement Must be Executed

How can we test our program?
Let’s start with something simple.
Let’s say that we have a statement that is never executed by tests.
Does the statement work correctly?
How can we know? We have no tests!
So, no, it does not work correctly.
At a minimum, we must execute every statement (called full code coverage).

What Happens When We Run the Program?

Imagine that we compile and run the program.
Take a look at the code.
The first statement is a printf.
The printf always executes, so
◦ we can check whether the printf works
◦ by simply looking at the output.
Choose a Line of Text as Our First Test

The program then waits for input with scanf.
What input should we give?
Let's just choose something concrete.
Say “0 0 0” (and then <Enter> to start).
What are the values of variables a, b, and c?

0, 0, and 0
What does scanf return? 3
What happens next? Skip the “then” block!

Continue Analyzing Until the End

With input “0 0 0” our program next
• prints the equation to be solved, and
• calculates the discriminant D.
What is the value of D? 0
(Remember that a, b, and c are all 0.)
So which of the three if-else blocks is executed (first, second, or third)? second
And what is x1? 0 / 0 → NaN

Was that a Bug?

I think so.
The equation is not quadratic when a is 0.
The person who wrote the code perhaps didn’t think of that case.
And neither did I when I edited the code to present to you.
Bugs can be subtle, and testing can be hard!
We won’t fix the bug.

Remember: We Want Full Code Coverage

Let’s try again with input “1 0 0”.
The same parts of the code execute.
And x1 is? 0
So the single root is at 0, and the program ends successfully.
Our equation was F(x) = x^2 (+0x + 0), so plugging in x = 0 does produce F(x) = 0.
But our test does not execute all code!
Adjust the Inputs to Change the \texttt{if-else} Results

What statements did not execute?

◦ “then” block of \texttt{scanf} check
◦ first case of \texttt{if-else} solution computation
◦ third case of \texttt{if-else} solution computation

Let’s adjust our inputs
to execute the other solution cases.
“1 0 0” gave the second case because
\texttt{D was not positive} and \texttt{D was 0}.

Use “1 0 1” to Test the Third \texttt{if-else} Case

To get \texttt{D} negative, change \texttt{c} to 1
(then \texttt{D} is \texttt{-4 == 0 * 0 - 4 * 1 * 1}).

For the next test,
◦ we type “1 0 1”,
◦ and the program tells us
◦ there are no real roots.

Our equation was \( F(x) = x^2 + 0x + 1 \), so
in fact no value of \( x \) can produce \( F(x) = 0 \).

Use “1 1 0” to Test the First \texttt{if-else} Case

For the first if-else case, we need \texttt{D} positive.

To get \texttt{D} positive, change \texttt{b} to 1 and \texttt{c} to 0
(then \texttt{D} is 1 == \texttt{1 * 1 - 4 * 1 * 0}).

For the next test,
◦ we type “1 1 0”,
◦ and the program gives roots at 0 and -1.

Our equation was \( F(x) = x^2 + 0x + 1 \), so
\( F(x) = 0 \) at both \( x = 0 \) and at \( x = -1 \).

We Need to Execute the “then” Block of \texttt{scanf}

So far, we have four tests:
“0 0 0” (known bug), “1 0 0”, “1 0 1”, “1 1 0”

But we still need a test to execute
the “then” block of the \texttt{scanf} check!

Anything that stops \texttt{scanf} from finding three
numbers will do. Let’s type “\texttt{hello}”.

So five tests (and verifying the output
by hand!) gives full code coverage for this program.
Good Testing Must Consider Both Purpose and Structure

Full code coverage is just a starting point.
In fact, you should notice that
◦ one of our tests ("0 0 0")
◦ exposes a bug
◦ in a statement that was already covered
◦ by another test ("1 0 0").

In general, good testing requires that one think carefully about the purpose of the code as well as the structure of the code.

So Easy that a Computer Can Do It

Full code coverage is easy to explain.
Finding tests to cover more statements means solving some equations.
Computers are good at that (well ... pretty good).
The automatic programming feedback tool uses this approach to try to find bugs in your code:
◦ generate tests to cover everything (if possible),
◦ then compare your program's results with a "gold" program (written by a professor or TA).