EPFL, CIVIL-127

Programming and software development for engineers

Project

- Did you register your group?
  - https://moodle.epfl.ch/mod/choicegroup/view.php?id=1335310
- Are you still looking for a partner?

Random Number Generation (RNG)

How do you generate a pair of equal-probability outcomes with an unfair coin?

What is a RNG?

- RNGs are the digital equivalent of tossing a coin. We want to choose among two options with equal probability.
- Once you have one bit of randomness, you can repeat the process to generate N-bits.
- If you want to generate random numbers between [0, 10]:
  - Generate 4 random bits (giving you a number between 0 and 15)
  - Discard 11, 12, 13, 14, 15, and repeat the above step

How does a deterministic machine generate random numbers?

Sources of randomness

- Unpredictable external events are used as sources of randomness
  - Thermal noise
    - E.g. amplify noise in a circuit + measure voltage
  - Time differences
    - E.g. run two or more oscillators and detect tiny variations due to analog imperfections and environmental differences
  - Lava lamps
  - Time between network packet arrival
  - Time between keystrokes or mouse movement
  - etc.

More secure sources of randomness

- Some people need a source of randomness which cannot be tampered with
- Several quantum phenomena fill this need
  - E.g. Geiger counter, beam splitter, etc.

Typical RNG setup

- Various chips provide random numbers
  - Processor
  - TPM
  - etc.
- The operating system will use these + other sources of randomness to slowly and continuously fill an entropy pool
- The entropy pool gets used to seed a pseudo-random function. The result is a fast RNG available to applications
- On Unix command lines, you can see this in action simply with `cat /dev/urandom`
- With python, you can do `import random` followed by `print(random.randint(a, b))`

Pseudo-random function

```
class LCG:
    def __init__(self, seed):
        self.seed = seed

    def next(self):
        self.seed = (1103515245 * self.seed +
          12345) % (2**31)
        return self.seed
```

- A function which is seeded
  - A seed is random bytes which fill the functions' initial internal state
- Generates random-looking output
- Many simple constructions are weak
  - Without knowledge of the seed, given N outputs: you don't want to be able to predict output N+1

Typical software execution process (compiled code)

1. Code is written in an IDE or text editor
2. A compiler translates the code to machine code
3. (optional, for static linking) A linker combines libraries with the machine code
4. A binary file/archive is generated
5. The operating system loads the binary file in memory
6. (when dynamically linking) A linker combines libraries with the machine code
7. The processor runs the machine code, instruction-by-instruction. The software uses libraries, which use services provided by the operating system.

   Note: At any time, the operating system can pause a program being executed and run another program. Enabling multiple applications to run at the same time

Typical software execution process (interpreted code)

1. Code is written in an IDE or text editor
2. The operating system loads an interpreter in memory
3. (when dynamically linked) A linker combines libraries used by the interpreter
4. The interpreter loads the code in memory
5. (optional) the code is transformed into bytecode
6. The interpreter executes the bytecode, instruction-by-instruction

Typical software execution process (JIT compiler)

JIT = Just In Time compilation

1. Code is written in an IDE or text editor
2. The operating system loads a virtual machine (VM) in memory
3. (when dynamically linked) A linker combines libraries used by the VM
4. The VM loads the code in memory
5. (optional) the code is transformed into bytecode
6. The VM converts the bytecode into machine code and loads the machine code into memory
7. The processor executes the machine code, instruction-by-instruction

Compiled code vs interpreted vs JIT

Typically:

- Compiled code loads the fastest
- Interpreters are slightly easier to implement than compilers or a JIT. The code however runs slower (overhead of the interpreter)
- Interpreters and JITs enable “write-once-run-everywhere” as well as instrumentation

RNG applications

- for simulations
- for games
- to generate tokens or secrets
- for sampling
- etc.

Reproducible experiments

For reproducible experiments, you want:

- A pseudo-random function
  - You can share the seed and other people can reproduce your experiment
- A high quality function
  - You don’t want patterns or biases in the random numbers

Can you invent an algorithm to shuffle a list?

- Can you prove that your algorithm is correct (it does not drop or duplicate items)?
- Can you prove that with your algorithm, every outcome is equi-probable?

More info about RNG

- https://www.rfc-editor.org/rfc/rfc4086.html
- https://csrc.nist.gov/projects/random-bit-generation
- https://www.bsi.bund.de/
- https://wiki.archlinux.org/title/Random_number_generation
- (and a ton of other resources, RNGs are a well documented topic)

Debugging

Most software is filled with bugs

Can you reliably reproduce the bug?

- Yes → good, you'll soon understand why the issue is happening, enabling you to fix the issue
- No → find ways to increase the likelihood of reproducing the issue

Error messages

- Read the error message
- If there are many error messages, start with the first one
- Sometimes, error messages are misleading. E.g. “permission denied” might mean “doesn’t exist”.
- Re-read the error message

Reproduce the bug in a debugger

- Set breakpoints
- Set conditional breakpoints
- Once you have identified the line of code where the bug exists, look at the variables
  - If any have an unexpected value, find where the incorrect value came from
- Some debuggers let you break when a variable changes (watchpoints)
- Some debuggers let you record/replay programs (time travelling)
- Debugger's REPL lets you evaluate expressions at the break point
  - REPL (read-eval-print-loop) == python shell. Not to be confused with the Terminal / bash shell

Trust nothing

When the bug is in your code

- Don’t assume your code is correct
- The majority of bugs are in your own code
- Try different inputs
  - Going from “this fails when X==1” to “this fails when X !=3” can be very helpful
- Add assertions as you go. You might be able to save these assertions and avoid other bugs down the road
- Take a break and get back

Consider writing tests

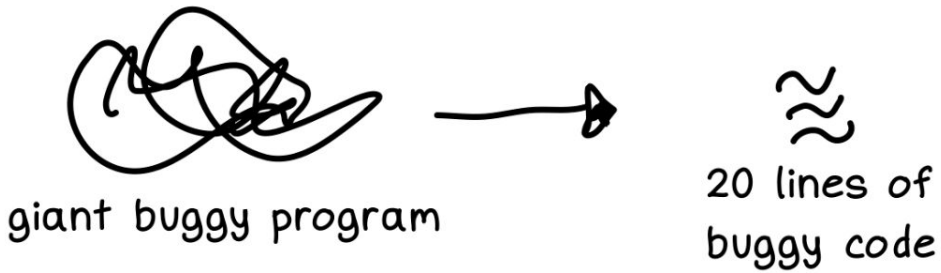
If you can write a failing test that demonstrates the bug:

- Reproducing the bug becomes easier (simply run the test)
- You can share your findings (hand the test to someone else)
- You can fix the bug and observe that the test goes from failing to passing
- You won't accidentally re-introduce the bug (called a regression)

The process of writing tests first, then fixing code is sometimes called test driven development (TDD)

When the bug is in a library

- Don’t assume your code is incorrect. In rare cases, bugs are actually in libraries your program is using
- To ensure you are 100% correct, you can write a new, minimal, program that demonstrates the bug
- It will be easier to experiment and explain to other people what’s going on
- You can even share the minimal program with the library author when filing a bug report



Graphic source: “The Pocket Guide to Debugging” by Julia Evans

Keep an engineering diary

- Write down what you have tried, what results you were expecting and what results you observed
- Keep a copy of the error messages you have seen, screenshots, etc.
- Write down assumptions you are making
- Write down what debugging techniques and tools worked for you
- Learn something new from each bug you fix

Explain the bug to someone else

- Sometimes, explaining the context and what you are seeing is enough for you to figure out the answer. Explaining every step of your code and walking through it can help pinpoint where things are going wrong
- Having another person talk through things is helpful but doing it alone also works and is called *rubberducking*

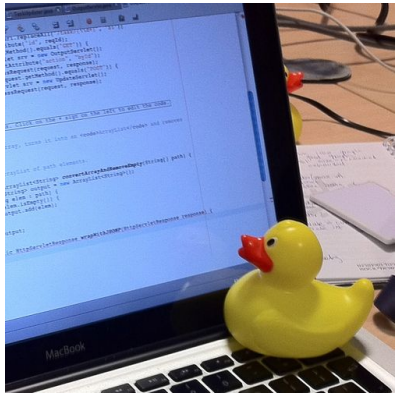


Image: https://commons.wikimedia.org/wiki/File:Rubber_duck_assisting_with_debugging.jpg by Tom Morris, CC BY-SA 3.0

git bisect

- If the bug is a regression (did not exist in a previous version of your code)
- And you are tracking your code in git

You can use `git bisect` to find which commit introduced the bug. Git bisect will checkout various revisions and ask you if the bug exists or not. It's binary search, but over a graph!

git bisect can even be automated if you have a unittest!

git bisect works best when you keep your commits small. Each commit should do one thing and only one thing.

Heisenbug

- Bug that seems to disappear when you attempt to study it
- Typically caused by (usually) subtle or (sometimes) obvious differences between a production and development environments
  - E.g. a bug which only happens under substantial load
- Can be caused by rare events (e.g. timing or race conditions)
- Can be caused by side effects when printing a variable
  - See https://github.com/numpy/numpy/issues/10382

Print-based debugging vs logging-based debugging

- Something isn't working as expected
- So you re-run the code but keep adding `print()` statements as you go
- The output helps you locate the bug
- You fix the bug
- You then have to remove the print statements

- Something isn't working as expected
- So you re-run the code but you enable additional logging. You might add new log statements as you go.
- The output helps you locate the bug
- You fix the bug
- You then turn off the logger

Add lots of logging

- Sometimes my logging is descriptive, I keep those around forever. E.g.
  `logger.debug(f"pos: {pos}, passengers: {...}”)`
- Sometimes I just log a number to make sure the code is running in the order I think it’s running and that I’m sane. I remove these once I’m convinced things are good. E.g.

```
logger.debug("here 1")
logger.debug("here 2")
logger.debug("here 3")
```

More info

- Julia Evans' has written several comics related to debugging (and other computer science-related topics)