Missing Semester

**Debugging**

1. Use journalctl on Linux or log show on macOS to get the super user accesses and commands in the last day. If there aren’t any you can execute some harmless commands such as sudo ls and check again.
2. Do [this](https://github.com/spiside/pdb-tutorial) hands on pdb tutorial to familiarize yourself with the commands. For a more in depth tutorial read [this](https://realpython.com/python-debugging-pdb).
3. Install [shellcheck](https://www.shellcheck.net/) and try checking the following script. What is wrong with the code? Fix it. Install a linter plugin in your editor so you can get your warnings automatically.

*#!/bin/sh*

*## Example: a typical script with several problems*

**for** f **in** $(ls **\***.m3u)

**do**

grep -qi hq.**\***mp3 $f \

**&&** echo -e 'Playlist $f contains a HQ file in mp3 format'

**done**

fixit.sh

1. (Advanced) Read about [reversible debugging](https://undo.io/resources/reverse-debugging-whitepaper/) and get a simple example working using [rr](https://rr-project.org/) or [RevPDB](https://morepypy.blogspot.com/2016/07/reverse-debugging-for-python.html).

**Profiling**

1. [Here](https://missing.csail.mit.edu/static/files/sorts.py) are some sorting algorithm implementations. Use [cProfile](https://docs.python.org/3/library/profile.html) and [line\_profiler](https://github.com/rkern/line_profiler) to compare the runtime of insertion sort and quicksort. What is the bottleneck of each algorithm? Use then memory\_profiler to check the memory consumption, why is insertion sort better? Check now the inplace version of quicksort. Challenge: Use perf to look at the cycle counts and cache hits and misses of each algorithm.
2. Here’s some (arguably convoluted) Python code for computing Fibonacci numbers using a function for each number.

*#!/usr/bin/env python*

**def** **fib0**(): **return** 0

**def** **fib1**(): **return** 1

s **=** """def fib{}(): return fib{}() + fib{}()"""

**if** \_\_name\_\_ **==** '\_\_main\_\_':

**for** n **in** range(2, 10):

**exec**(s.format(n, n**-**1, n**-**2))

*# from functools import lru\_cache*

*# for n in range(10):*

*# exec("fib{} = lru\_cache(1)(fib{})".format(n, n))*

**print**(eval("fib9()"))

Put the code into a file and make it executable. Install [pycallgraph](http://pycallgraph.slowchop.com/en/master/). Run the code as is with pycallgraph graphviz -- ./fib.py and check the pycallgraph.png file. How many times is fib0 called?. We can do better than that by memoizing the functions. Uncomment the commented lines and regenerate the images. How many times are we calling each fibN function now?

1. A common issue is that a port you want to listen on is already taken by another process. Let’s learn how to discover that process pid. First execute python -m http.server 4444 to start a minimal web server listening on port 4444. On a separate terminal run lsof | grep LISTEN to print all listening processes and ports. Find that process pid and terminate it by running kill <PID>.
2. Limiting processes resources can be another handy tool in your toolbox. Try running stress -c 3 and visualize the CPU consumption with htop. Now, execute taskset --cpu-list 0,2 stress -c 3 and visualize it. Is stress taking three CPUs? Why not? Read [man taskset](https://www.man7.org/linux/man-pages/man1/taskset.1.html). Challenge: achieve the same using [cgroups](https://www.man7.org/linux/man-pages/man7/cgroups.7.html). Try limiting the memory consumption of stress -m.
3. (Advanced) The command curl ipinfo.io performs a HTTP request and fetches information about your public IP. Open [Wireshark](https://www.wireshark.org/) and try to sniff the request and reply packets that curl sent and received. (Hint: Use the http filter to just watch HTTP packets).