Code profilers and profiling

- When writing code, we generally have some “feel” for how effiicient it is (run time and/or memory use)
- That might be supported by big-O complexity analysis of the algorithms we use (e.g. O(logN), O(N), etc)
- That doesn’t mean we’re correct, unfortunately, so we often want experimental validation of how fast it runs (or how much memory it uses) in practice
- Tools to analyze program run time/memory use are called profilers

How code profilers work (timing)

- Code profilers are often used to analyze not just how long a program takes to run (we can get that from shell-level tools like /usr/bin/time) but also how long each function or method takes to run (cpu time)
- Two main techniques used by profilers: code injection, sampling

Code injection profilers

- These profilers require re-compiling the program with special flags
- At each point where a function call/return is made, these code is inserted to check the current system time (down to microseconds) and log that information
- The actual profiling program reads and summarizes the log file, generating a report on how much time was spent in each function/method, how many times it was called, etc

Sampling profiler

- These require operating-system level priviledges
- The program is run in a special mode, with OS interrupts generated at fixed intervals
- At each of these interrupts, it checks and logs which function/method the program is currently executing
- This builds up statistical data on how much time was spent in each function
- The profiler analyzes and reports on the statistical data

Drawbacks to each style

- Injection style of profiler adds extra code into the program before running it, which subtly alters the run time behaviour
- Sampling style of profiler requires operating system level priviledges (often not feasible), and gives a statistical analysis, not an exact measurement, of run time

Gathering useful data

- Generally we want to be able to extrapolate from our profiling data, to make predictions on how program will behave on larger and larger data sets
- To do so, we need realistic data that can reveal trends
- Means profiling the program on lots of different data sets of different sizes (e.g. for a sorting program we might run it on a set of 1000 random values, 10000, 100000, etc, to spot trends)
- We need to think about representative data sets, i.e. with properties that reflect the real-life data that would be used

Memory profiling

- In addition to run time, we often want to analyze how much memory a program uses
- Similar implementation ideas, but involves inserting code to check how much memory is in use by the program at checkpoints
- Might also include code to track each dynamically-allocated memory element, to monitor its size, see if it is freed or not, check for memory leaks etc

Gathering profiling data

- Will use profiling data as basis for extrapolating/predicting program behaviour, need varied/representative data points
- Sorting example, gather/plot following data:
  - suppose we have a sorting program that we think will normally be used on files 10,000-50,000 lines long
  - Might choose 20 test sizes: 5000, 10000, 15000, ... , 100000 lines long, for each size use 3 styles: sorted, reverse-sorted, randomly-ordered
  - For each of the 60 combinations run 5 tests, so 300 tests overall