Assignment: Generic Infrastructure For Performing "Big-O" Analysis

Avraham Leff

September 6, 2022

1 Assignment-Specific Packaging

The general packaging is unchanged from the basic "Homework Requirements" (see slides from first lecture and "Homework Policies for COM 2545" on Piazza).

This assignment's "DIR" **must be named** *BigOIt*, and your application's Java class **must be named** *BigOIt.java*.

2 Motivation

In textbook (Chapter 1.4), Sedgewick makes the case that we can estimate "order-of-growth" by conducting a series of **"doubling ratios" experiments**. We discussed the importance of this "empirical" or "scientific" approach in lecture: it allows us to treat a program as a "black-box" such that we can calculate its order of growth *without looking at the code!*

Writing the code for such experiments is a good way to reinforce these concepts. Also, to grow as algorithmists, it's important for you to incorporate such doubling ratio experiments into your programming toolkit. The code that you will write in the assignment should make it much easier to do so.

> The "algorithm" concepts needed for this assignment are fully spelled out in the textbook. To make the assignment more interesting, I've therefore added a *"software engineering"* component.

The general idea:

1. I've defined a *BigOMeasurable* abstract base class that defines an API that is easily wrapped around most concrete algorithms. When you want to perform a doubling ratio experiment on a given algorithm, you extend *BigOMeasurable* in a class that fully implements the API such that the algorithm can be executed through the *BigOMeasurable* interface.

2. I've define a *BigOItBase* abstract base class that specifies the API for a "general purpose" doubling ratios experimental apparatus. You will extend this class to provide a concrete implementation named *BigOIt* that your clients (e.g., my test code) can invoke to get a "doubling ratio" estimate for a given algorithm (i.e., an implementation of the *BigOMeasurable* API).

I hope that this assignment will reinforce the lecture and textbook's discussion of "empirical" order-of-growth estimation. Another result may be to increase your suspicions of "overly neat" performance numbers and graphs ☺. Finally, you will develop a healthy respect for the "clock time" implications of "expensive" algorithms.

The challenge in this assignment is for you to devise a sufficiently robust experiment harness such that you can rely on the output numbers to make claims about the algorithm order of growth.

3 Requirements

> Please review the general requirements for a programming assignment! I've tried to reduce the chances of "mistakes" occurring through the use of a "skeleton interface", but ultimately, **you are responsible** for ensuring that I can compile and test your code without incident.

1. Begin by downloading the skeleton *BigOMeasurable.java* and *BigOItBase.java* class from this git repository.

2. Check these classes into your repository in the correct directory

3. Provide a *BigOIt* class that `extends BigOItBase`.

3.1 Suggestions

I don't require that you write test code, but don't see how you can succeed in this assignment without such code. Remember: *BigOIt* code won't be able to inspect the algorithm (*BigOMeasurable*) that's being evaluated: you can only inspect it by invoking the *BigOMeasurable* API and assessing its performance.

You will need to create instances that implement *BigOMeasurable* **given a class name**. The only reasonable way to do so is to use Java's reflection APIs. You should research their use, and are allowed to use relevant Internet code snippets in your own implementation. As usual, you should credit anything beyond "one-liners" with a comment in your code.

Your "doubling ratio" experiments need robust measurements to provide accurate answers. Some suggestions:

- Before collecting data, "warm up" the JVM to benefit from JIT compilation.
- Collect large amounts of experimental data
- Run multiple iterations, average results both within and across iterations
- Scale "n" to reasonably large values, discard values collected for "small n" as they typically produce useless data.

Note that these suggestions are loosely specified: I wish that it were possible to give you more concrete suggestions but (to an extent), this involves "art and experience" in addition to "science".

Bottom-line: use whatever set of techniques improves the accuracy of your results (ideally, based on your own experimental results).

3.2 Timing: A Complication

One implementation technique (given the above) is for your code to take its time ("collect tons of data") before producing an answer. If only because I have to grade the work of many students, this approach isn't permitted. Instead, I will specify a "time out" in which your code must produce a result (see the Javadoc).

This implies that you must "build-in" an "alarm clock" such that your code interrupts its work before the time out expires to return a result

to the client. Java provides various APIs to accomplish this task: if you haven't used these classes yet, search for (e.g.) `java set timer`).

If you feel that the specified time out doesn't suffice to produce good results, you're allowed to return NaN. I may deliberately specify a "too short" time out to see how you handle this scenario. However, if I feel that I specified a reasonable time out, and you return `NaN`, I will consider the test to have failed.

> Note: in such cases, I pad my estimate of "reasonable time" to give a safety margin.

Your code may want to handle "time out" requirements by returning the *best answer calculated so far*.

3.3 Grading

I will test your work by writing implementations of *BigOMeasurable* that have well-defined runtime behavior, and then verifying that your *BigOIt* confirms this behavior with its doubling ratio experiment. Your code must be accurate to within several tenths of a decimal point worth of precision.

3.4 Sample Test Invocation

```
public static class Mystery extends BigOMeasurable {
  public Mystery() {
    // [stuff]
  }

  @Override
  public void setup(final int n) {
    // stuff
  }

  @Override
  public void execute() {
    // stuff
  }
}

  final BigOItBase it = new BigOIt();
```

```
final String bigOMeasurable = Mysteryclass.getName();
final double actual = it.doublingRatio(bigOMeasurable,
   40_000);
final double expected = 1.0;
logger.info("Doubling ratio for class {} is {}",
            bigOMeasurable, actual);
assertEquals(actual, expected,  DELTA,
             "Mismatch on expected doubling ratio");
```