15-122: Principles of Imperative Computation, Fall 2014
Lab 3: Timing and Testing

Tom Cortina (tcortina@cs) and Rob Simmons (rjsimmon@cs)

Monday, September 15, 2014

COLLABORATION: In lab, we encourage collaboration and discussion as you work through the problems. These activities, like recitation, are meant to get you to review what we've learned, look at problems from a different perspective and allow you to ask questions about topics you don't understand. Feel free to talk with your neighbors about the problems if you get confused or stumped!

1 Timing Code

On the Linux machines, there is a way to determine the actual running time of a program. You use the `time` command followed by the program name (and its arguments) that you want to time. For example, to time an executable `a.out` in your current directory, you would enter:

```
time ./a.out
```

and you would get an output that looks something like this, that shows that the program used 4.602 seconds of user time:

```
Timing 1000 times with 2^18 elements
0
4.602u 0.015s 0:04.63 99.5% 0+0k 0+0io 0pf+0w
```

NOTE: When timing code, do not use `-d` during compilation. The extra debugging done by contracts will increase the overall runtime and can affect the overall asymptotic complexity of the algorithm you are timing.

Exercise 1. In this activity, there are six programs, `timingtest1`, `timingtest2`, `...`, `timingtest6` that allow you to vary the input size for the algorithm they're running by giving an optional `-n` argument. The default size, 1000, is a good starting place for all 6 programs. For example, to time the first program using the input size 1000, you would enter:

```
time timingtest1 -n 1000
```

Your job is to determine the asymptotic complexity (runtime) of all six programs expressed using big O notation as a function of n, in its simplest, tightest form. HINT: Exactly one of them is $O(logn)$. Do this by running the programs for varying sizes of n and plotting your results (or looking for obvious patterns). Remember you can work with a neighbor or two as you collect and analyze all of the data.

2 Testing

In this activity, you will write some comprehensive unit tests for a few of the image processing functions of Program 3. Please do not look at your `rotate.c0` or `mask.c0` code in this lab! However, just for the duration of this lab, you can collaborate on writing test cases.

Exercise 2. Write unit tests in images-test.c0 for either or both of the following:

```
pixel[] rotate(pixel[] A, int width, int height)

int[] apply_mask(pixel[] pixels, int width, int height, int[] mask, int maskwidth)
```

We don't have good tools for testing what happens when you give precondition-violating inputs to these functions, so your test cases should be valid inputs, and you should confirm the outputs. Here are some testing tips:

- Constructing an input of a 1x1 array and a 2x2 array with 4 distinct pixels would be good in the tests for `rotate`.
- Constructing a 1x1 mask or a 3x3 mask alongside 1x1, 2x3, 3x4, and 4x5 images would be good in the tests for `apply_mask`.
- Confirming, using assert() statements, that the results are the ones you expected (that is, checking that all the resulting pixels are in the right place).

You can compile your test cases locally using

```
% cc0 -d pixel.c0 imageutil.c0 rotate.c0 mask.c0 images-test.c0
% ./a.out
```

or you can submit `images-test.c0` (just that file, not a `.tgz` file) to Autolab. We will run your tests against a correct implementation, and also against very very wrong implementations with bugs that we would classify as *contract exploits*: they'll satisfy all the reasonable postconditions we would expect you to write, but will usually produce an answer that is flatly wrong.