15-122: Principles of Imperative Computation

Lab Week 3 — Tom Cortina, Rob Simmons

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!

Grading: For partial credit, you should attempt task (1.a) and successfully differentiate between the four fast algorithms (the ones that are at least in $O(n)$) and two slower algorithms. For full credit, the test cases for one of your images functions should also be accepted by Autolab. (Show your TA the autograder output that says you get points.)

Reminder: it's okay if you don't get full credit on every lab! The way we grade labs, you will get all the possible points as long as you attend every lab and get full credit on a handful of labs.

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
Testing with n = 1000... Done. 0
4.602u 0.015s 0:04.63 99.5% 0+0k 0+0io 0pf+0w
```

The first number (the one with a `u` after it) is the best one to track for this activity. The "u" is for *user*, which is closer to what we want that the system time (the amount of time the program handed control over to the operating system) or the wall clock time.

(1.a) *Do not spend more than 15 mintues on this task. Work with a neighbor or two to collect and analyze all of the data.*

On AFS, 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.

You can distinguish the programs by running them for varying sizes of n and plotting your results or look for obvious patterns.

There is one $O(1)$ algorithm, one $O(\log n)$ algorithm, and two $O(n)$ algorithms, one of which has a very large constant. It is tricky to distinguish these four! There are no $O(n \log n)$ algorithms; it's *really* hard to distinguish these from $O(n)$ algorithms by timing.

Testing for the images assignment

Setup: In this activity, you will write some unit tests for a few of the image processing functions in the current homework assignment. The code you'll need for this assignment is the `images-handout` directory for the images programming assignment. (Download this code if you haven't already.) You should write your code in a new file, `images-test.c0`, in this directory.

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. (Unless we make an explicit exception like this, the academic integrity policy for 122 doesn't allow test cases to be shared, since they are code.)

(2.a) Write unit tests in `images-test.c0` for either or both of the following functions described in the images assignment:

```
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 that the outputs are what you expect them to be. Here are some testing tips:

- For `rotate`: construct a 1x1 image and a 2x2 image with 4 distinct pixels. Calculate where you expect these pixels to end up in the resulting array. Check that the resulting pixels are what you expect using `assert()` statements.
- For `apply_mask`: construct a 1x1 mask and/or a 3x3 mask alongside 1x1, 2x3, 3x4, and 4x5 test images. Calculate what you expect the mask calculation to return, and check that it does so using `assert()` statements.

If you've started writing `rotate.c0` or `mask.c0`, you can compile your test cases locally using

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

(You don't have to include `mask.c0` if you're only writing tests for `rotate`, and you don't have to include `rotate.c0` if you're only writing tests for `apply_mask`.)

You should also submit `images-test.c0` (just that file, not a `.tgz` file) to the ungraded Autolab autograder created for this lab. This is the one called "Lab 3 Activity: Testing" and not the one called "Lab 3." In the autograder, we will run your tests against three kinds of implementations:

(a) A correct implementation, which must pass all your tests.

(b) Some implementations that fail a contract when given certain (reasonable) inputs. We call such bugs *contract failures*.

(c) Some implementations that will satisfy all the reasonable postconditions we might expect you to write, but which nevertheless do obviously wrong things. We call such bugs *contract exploits*.

The autograder for the images assignment will also accept your `images-test.c0` file and record the number of buggy implementations you catch on the scoreboard!