Week Four Reading Notes

A quick note on the code auto-graders: If you pass some, but not all, of the test cases, then you will get a scaled number of points. So if the problem is worth 20 points and you get 6/9 test cases correct then you will get (6/9)*20 points on that problem.

Please don't fret if you can't get 100% of points on a given problem! Partial credit is possible! If you're in doubt, please visit your Progress page, where the number of points you've earned on each problem will be displayed.

<u>Lecture Sequence 7</u>

Remember all those programming mistakes you've made as you've begun learning how to program? How many times has the auto-grader pronounced your code to be incorrect? Sure, it may have been essentially correct, but the grader is just being too picky! You may have written "true" instead of "True," so couldn't it just accept it?

It is frustrating at first, but soon the syntax will become second nature. Unfortunately, the frustrations will not end. In fact, they will increase. As the programs get bigger, so do the mistakes -- Little programs, little bugs. Big programs, big bugs!

It is likely that you do not yet appreciate a big, fat, juicy bug. Probably all of your mistakes are due to not understanding the precise meaning of the Python expression or command, overlooking some typing mistake, or not understanding the assignment. Bugs can be worse than that, and take a lot longer to find.

Deciding at what point in the semester to cover testing and debugging in not easy. Perhaps we should wait until you have written complex programs that have complex bugs. However after having the experience of suffering for a week only to discover that the variable N was initialized to 0 rather than 1, may be a good time to teach good programming techniques.

The debugging and defensive techniques covered in lecture may seem obvious and good but not really practical. Similar to the fact that everyone should read the terms and conditions before checking the box that says you agree to them. That is obvious and good but no one does it. On the other hand, we do teach a child to wear a bicycle helmet early on, even if their parents are running alongside holding the seat and the bike has training wheels. Testing and debugging is like a helmet and training wheels, seat belts, or wearing a belt and suspenders. It is best to start early and get into the habit. I can guarantee that your programs will have far fewer bugs and will be better by following the testing strategies presented in this lecture sequence.

<u>How To Properly Use Our Code Auto-graders To Maximize Your 6.00x Experience</u>

Guess and check is an interesting strategy but it must be used wisely and in the right places. **As soon as you write a function, test it**. When you finish coding up a problem set, test it with your own data. *Then* use the test cases we provide. All this testing should be done on your own computer.

Once you believe your is correct, then paste it in the box and have our servers check it. You **will not learn to code by simply guessing what might be correct and seeing if it is right**. That is not what we mean by guess and check. Thinking and trying things out on your own computer is the way to learn.

We have noticed that many students are abusing the large number of checks we give you on problem set problems, which makes us worry that many of you are trying the guess-and-check approach to coding. Note that we provide 30 checks per problem because we understand that there are numerous things that can go wrong: poor or intermittent internet connections, temporary overloaded servers, copy-paste indentation errors, or some other random issue. We assume that no more than 3 checks should be needed for any given coding problem, because we expect our students to be running their code locally, testing and debugging as just described. To be safe we multiply that by 10. When I learned to program, we were able to run our programs on the main computer at most three times per day and the only personal computer we had was our brains!

When my code does not work, I usually play detective and talk out what the code should have done and what it did instead. Some people find it difficult to talk to themselves, and talk to someone or something like a rubber duck! (I'm not joking - see http://en.wikipedia.org/wiki/Rubber_duck_debugging!) If you still have problems, the discussion forum is the best place to go. Like rubber ducks, simply describing your difficulty may help you solve it, but unlike rubber ducks, you will get an answer back.

(I am hoping to see more discussion on the forum; that is why it is called a *discussion forum*)

Lecture Sequence 8

Complexity is a computer science topic, not a programming one. Plenty of computer programmers do perfectly fine without ever learning complexity theory. We include it as part of the course because it is both fundamental and important. No matter your computer's speed or memory size, it seems that it is never fast enough and never has enough storage. A good program tends to have a life of its own and will be used in all sorts of ways not envisioned by the programmer.

For example, mobile phones initially came with a 50-entry address book. Today, there are people with 5,000 entries. An inefficient algorithm is fine for 50 entries, but may be painfully slow for 5,000. Many programs take a long time, but only because they have been poorly programmed or poorly designed. It may not be fair to blame the programmer, as he or she may not have learned how to analyze algorithms. Complexity theory is useful in understanding the computational needs of programs.

To appreciate the material covered in Lecture 8, it may be helpful to consider the computational needs of a Massive, On-Line, Open Course (like 6.00x!). Let's focus on the challenges presented by the auto-graders.

What is the most important metric?

- The number of finger exercises?
- Whether or not there are multiple choice questions?
- The speed of the written code?
- The number of students in the course?

Well, if there are 50 students, probably any computer will be sufficient provided the programs do not require too many resources. But if there are ten thousand, a hundred thousand or a million students, then the number of students is going to be the dominant factor. We have to have enough compute power to handle that load. If the enrollment doubles, we will need twice as much compute power. If S is the number of students, we can say that the load increases linearly with S.

Do the number of exercises matter? Sure. But how does this number compare to the number of students? The number of exercises does not depend on the number of students; rather, it is related to the number of lectures. This course, 6.00x, has the same number of problem sets as given to the classroom version with just 100 students.

There is a similar argument concerning the multiple choice questions. The amount of overhead to check them can be considered to be a fixed number and independent of the number of students.

But, what about the number of times a problem set can be checked? If there was no limit on it, then there is no way of predicting the load. One week, it could be the case that every student submits a check of his or her problem set 1,000 times. This would have serious effects on the response time of the auto-graders. (Note that the reason we limit the number of checks is so that you will first test and debug your programs on your local computer – see the Lecture 7 reading notes).

What about the compute power to autocheck the programs? Now, things get interesting. Some programs need more compute power. Which ones? The ones with many iterations or lots of recursion. A small program with a loop of 50,000 iterations, submitted by each of, say, 100,000 students would mean a total of 50,000 * 100,000 loop iterations to be evaluated. We can start feeling sorry for the compute servers for having so much work to do.

Remember how many steps it took to do the Tower of Hanoi puzzle? The number of moves depends on the number of disks. Just adding one disk doubles the number of moves. With D disks, there are 2^D moves. So, even one student can make the computer work hard. That number of 2 to the power of D can be very big. It can easily dominate the number of clicks that can be continuously performed by every student in the class every minute of the day.

Complexity theory is more precise than just described. Nevertheless, the reasoning is similar. What is the right thing to measure and what can be ignored? We ignored the communication time or internet traffic from each of the students, which could very well make a difference, but it makes for a very messy model. In a similar way, computational complexity ignores the time it takes to access memory. Anyone who has studied computer architecture knows that this is silly. Every memory bit takes up some physical space. With more memory, it is going to take longer to move the electronic signals across the memory chip. We fix the clock beats making it long enough to accommodate the furthest areas of memory. So then, it is ok to assume all memory accesses take the same time. This is what is meant by the “RAM model”.

I would be happy to engage in a discussion on complexity theory in the forums. While the RAM model is one potential model, there are ones that more accurately capture the actual physics on the computer and others that are simpler to understand. When I started programming, people used the number of lines of code as a model. People would boast that they could program the 8 queens problem in under 100 lines of code. Nowadays, however, we strive for readable code – having a 100-line program that is impossible to understand won’t make you many friends!

I once wrote a program to simulate a soccer game (or should I say football for those billions of fans outside the US) on a computer that had 32 processors. There were 22 processors simulating the actions of the players and another processor simulating the ball. As soon as my team got the lead, those players would just send messages to each other overloading the communication bus in the computer and preventing any messages to get to the processor simulating the ball. To understand why my strategy worked, complexity theory needed to account for the communication mechanisms.

My strategy worked because there is a limit on communication bandwidth. Most of the time, programs do not reach this limit and it can be safely ignored. The model of computation assumed in the lectures ignores the communication limits focusing instead on the time to execute instructions. That model is fine for most computations – but not for all! That is both its strength and its weakness.

-Larry Rudolph