

*Greetings, reader!*

Thank you so much for taking the time to slog through this behemoth. I have done my best to distill the contents to representative nuggets but feel compelled to reiterate a line oft attributed to Blaise Pascal:

*Je n'ai fait celle-ci plus longue que parce que je n'ai pas eu le loisir de la faire plus courte.*

*I would have written a shorter letter, but I did not have the time.*

Still, perhaps you will find something to amuse, delight, or inspire within. If not, perhaps this document can serve as a sleep aid as students have sometimes commented that my writing on HW assignments helps them to surmount bouts of insomnia.

To facilitate dissection, I have structured the document according to the outline provided by the folks at the CTFE as you will see in the table of contents on the next page. Main sections appear in the upper left corner with subsections appearing in the middle at the top of the page.

If you had not heard, computer science folks are a pessimistic lot. An analysis from a few years ago indicated that NSF proposals in CS received scores about 0.5 points lower on a five-point scale than in other areas. I believe this mentality comes from our training: code never compiles the first time so one is constantly looking for what is wrong and when it does finally run the first time one is on high alert for the first sign that answers it produces are faulty. If answers look correct, then there is usually a desired feature missing that is yet to be written. This thinking so colors a coder's mindset that we often forget to appreciate the magic of a working system.

That is why constructing this portfolio was so refreshing for me. Of course I see the bugs in what I have done, know all the little flaws in my classes, the fact that a programming assignment doesn't quite reflect best practices, that I could have produced a better in-class activity to practice a new concept, and so forth. But rarely have I been in a position to synthesize the disparate pieces of my assorted classes into a holistic picture. That I have built these wonderful things surprises me somewhat but also brings great satisfaction.

Describing all of it feels strange, unnatural even, as I am not much of a marketer. However, I am extremely proud of my students and what we have accomplished together: much learning and much fun. While this document primarily comprises artifacts I have created, all of them exist to serve the fine folks in my courses. To them, I dedicate this work.

Happy reading,

A handwritten signature in black ink that reads "Chris Kauffman". The signature is fluid and cursive, with the first name "Chris" and last name "Kauffman" clearly legible.

Christopher Kauffman

## Contents

<b>1</b>	<b>Cover Letter</b>	<b>1</b>
<b>2</b>	<b>Contents</b>	<b>2</b>
<b>3</b>	<b>Teaching Curriculum Vitae</b>	<b>4</b>
<b>4</b>	<b>Statement of Teaching Philosophy</b>	<b>7</b>
<b>5</b>	<b>Reflection</b>	<b>12</b>
5.1	Lecture Discussion via Hot Seats . . . . .	12
5.2	In-class Activities . . . . .	15
5.2.1	CS 105: First Day Intro to Ethical Theories . . . . .	16
5.2.2	CS 211: Day One Exercise . . . . .	18
5.2.3	Exam Review via Java Jeopardy . . . . .	20
5.2.4	Other Class Activities . . . . .	22
5.2.5	Feedback on In-class Activities . . . . .	23
5.3	Assignment Specifications . . . . .	24
5.3.1	First Passes . . . . .	25
5.3.2	Testing Code with Code . . . . .	25
5.3.3	Beyond Correctness . . . . .	26
5.3.4	Not just What but How . . . . .	29
5.3.5	Organization . . . . .	30
5.3.6	Milestones and Deadlines . . . . .	30
5.3.7	Educating the Team . . . . .	31
5.3.8	Feedback on Assignments . . . . .	31
5.4	Office Hours . . . . .	32
5.5	New course development . . . . .	33
5.5.1	A Computing Course for Everyone . . . . .	33
5.5.2	General Approach to CS 100 . . . . .	34
5.5.3	First Steps Programming . . . . .	35
5.5.4	Beyond the Code on the Page . . . . .	35
5.5.5	Feedback on CS 100 . . . . .	36
5.6	Potpourri . . . . .	37
5.6.1	Assessments . . . . .	37
5.6.2	Gathering Feedback . . . . .	38
5.6.3	Community . . . . .	39
5.7	Sample Materials . . . . .	41
5.7.1	CS 310 Week 2 Lecture Slides . . . . .	41
5.7.2	CS 310 TripleStore Project . . . . .	46
5.7.3	CS 211 CensoredWriter Lab . . . . .	62
5.7.4	CS 100 Mini-Exam 4 . . . . .	67
5.7.5	CS 499 Mini-Exam 1 . . . . .	69
5.7.6	CS 100 Final Feedback Form and Results . . . . .	71
5.7.7	CS 211 Midterm Feedback Results . . . . .	78

<b>6</b>	<b>Evidence</b>	<b>91</b>
6.1	Summary of Teaching Evaluations . . . . .	91
6.2	Testimonials . . . . .	95
6.2.1	Sanjeev Setia, Department Chair, CS . . . . .	96
6.2.2	Pearl Wang, Associate Chair, CS . . . . .	98
6.2.3	Mark Snyder, Coordinating Instructor, CS . . . . .	100
6.2.4	Richard Carver, Coordinating Instructor, CS . . . . .	102
6.2.5	Jill Nelson, NSF SIMPLE Project Leader, ECE . . . . .	103
6.2.6	Student Successes . . . . .	104
6.2.7	Student Thank Yous . . . . .	105
6.3	Closing Statement . . . . .	107

## Chris Kauffman

### Teaching Curriculum Vitae

Term Assistant Professor, Department of Computer Science    Phone: 703-993-5194  
 George Mason University MSN 4A5    Email: kauffman@cs.gmu.edu  
 4400 University Drive, Fairfax, VA 22030 USA    URL: <http://cs.gmu.edu/~kauffman>

### Education

- 2013 Ph.D. Computer Science, University of Minnesota, Twin Cities  
 Thesis: *Computational Methods for Protein Structure Prediction and Energy Minimization*  
 Thesis Supervisor: Prof. George Karypis
- 2010 M.S. Computer Science, University of Minnesota, Twin Cities
- 2004 B.S. Computer Science, University of Minnesota, Twin Cities, Minor in Mathematics.  
 Graduated with High Distinction.

### University Teaching Experience

- 2012–on Term Assistant Professor teaching a wide array of computer science courses  
 Department of Computer Science, George Mason University, Fairfax, VA
- Fall 2011 Adjunct Instructor for CSC 301: Programming and Problem Solving  
 Department of Mathematics, Concordia College, Concordia College, St. Paul, MN
- Summer 2011 Adjunct Instructor for CSCI 2011: Discrete Structures of Computer Science  
 Department of Computer Science, University of Minnesota, Minneapolis, MN

### Awards Related to Teaching and Mentorship

- Spring 2015 Computer Science Department Teaching Excellence Award

### Summary of Teaching Ratings from GMU Student Evaluations

Course#	Teaching <sup>1</sup>	Course <sup>2</sup>	#Rates <sup>3</sup>	#Stdnts <sup>4</sup>	#Sects <sup>5</sup>	Course Title / Comment
CS100	4.61	4.07	85	114	3	Principles of Computing
CS105	4.77	4.24	568	654	19	Computer Ethics and Society
CS211	4.78	4.53	140	299	6	Object-Oriented Programming
CS222	4.75	4.46	105	145	4	Computer Programming for Engineers
CS310	4.70	4.44	244	364	8	Data Structures
CS499	4.84	4.84	25	33	1	Parallel Computing
All	4.74	4.34	1167	1609	41	Overall average of all ratings
Dept.	4.24	4.06				Overall Department Average ratings

- Ratings are averages over all students who provided ratings in a course.
- Teaching<sup>1</sup> measures answers to the prompt: “My overall rating of the teaching” rated from 1 (worst) to 5 (best)
- Course<sup>2</sup> measures answers to the prompt: “My overall rating of this course” rated from 1 (worst) to 5 (best)
- #Rates<sup>3</sup> counts the total number of students who provided paper evaluations over all courses
- #Stdnts<sup>4</sup> counts the total number of students who were enrolled each courses
- #Sects<sup>5</sup> counts the total number of sections of a course which was rated



## Curriculum Development

### Course Revision Contributions

- CS 105 Computer Ethics and Society. Course coordinator four times, helped to solidify material and guide adjuncts on material to be covered.
- CS 211 Object-Oriented Programming. Contributed to improving use of lab time; now split between some collaborative exercises, paper quizzes, and coding assessments. Helped to implement improved grading schemes for programming assignments. Served as course coordinator twice.
- CS 310 Data Structures. Served as course coordinator three times. Improved grading schemes for programming assignments.

### New Course Contributions

- CS 100 Principles of Computing. Chosen to develop and teach a course for non-CS majors to cover fundamental aspects of CS, programming, and social impact of computing. Instructed the pilot section in Fall 2014 and two section in Fall 2015.
- CS 499 Parallel Computing. Chosen to develop a new course exploring use of parallel programming models to undergraduates to handle large scientific problems. Taught during Spring 2016, introduced junior and senior students to a variety of technologies that enable multiple CPUs to be harnessed. Administered a computing cluster for use with the course.
- CS 110 Led departmental curriculum reform to merge two intro level courses, CS 101 and CS 105, to streamline the CS program and cover material currently missing from the curriculum. Set to teach its initial offering in Fall 2017.

### Program Development Contributions

- CS Honors Primary author for the Departmental Honors in Computer Science. Unanimously passed by full CS faculty in Fall 2015. Honors program allows advanced CS students to enroll in more challenging courses and culminate their degree with a research project to obtain additional recognition at graduation.
- GMU IT Active contributor to CS sub-committee providing guidance to reform of the Information Technology Outcomes for all GMU students to meet the University General Education requirements.
- BS CS Active contributor to assessing the current state of the computer science BS degree. Interviewed various faculty to identify strengths and deficits of courses to bring the program into closer alignment with ACM and ABET guidelines.

**Undergraduate Research Advisees**

- 2016-on Ethan Rarity. OSCAR Federal Work Study. Support Vector Machine / R Interface Library.
- 2016-on Hernan Ariascu. OSCAR Federal Work Study. DrJava IDE improvements to support CS course instruction in several GMU courses.
- 2013-15 Cycielya Schultz, OSCAR Federal Work Study. AIDS Education Game Development.
- 2015-16 Vankhanh “Lilas” Dinh. OSCAR URSP Award. Lojban as an Intermediate for Natural Language Translation. Presented work at the National Conference on Undergraduate Research and the GMU/VSE Undergraduate Research Celebration.
- 2014 Saif Rizvi. OSCAR Federal Work Study. Support Vector Machine / R Interface Library.
- 2013 Edward Martin. OSCAR URSP Award. AIDS Education Game Development.

**Teaching Related Service**

- 2016-on Academic adviser for 65 undergraduate students including 26 honors college students enrolled in computer science
- 2015-on Member of SPARC research group investigating alternative methods to improve scalability of teaching resources in intro CS courses.
- 2015-on Leader of SIMPLE Teaching Development Group for CS Department
- 2014-2015 SIMPLE Teaching Group member, led by Craig Lorie.
- 2014, 2016 Attendee to ACM SIGCSE conference on computer science education
- 2013-on Faculty Adviser for PatriotHackers Student Group
- 2013-on CS Department Web committee
- 2012-on CS Department Undergraduate Studies Committee

**Presentations Related to Teaching**

- Fall 2016 SIMPLE Summit. Led group discussion on balancing inclusive and evaluative practices in active learning classrooms.
- Fall 2016 Poster presentation at the Innovations in Teaching and Learning conference. Discussed the use of “Hot Seats” to increase class engagement through direct questioning and bonus credit.
- Fall 2016 GMU Student Run Computing Technology (SRCT) workshop on open source software development, keynote address.
- Fall 2013, Fall 2014 Guest lecture in CS 390: Research and Project Design Principles in Computing. Discussed applications of machine learning and optimization to protein folding,
- Spring 2016 Guest lecture in CS 101: Preview of Computer Science. Discussed applications of machine learning and optimization to protein folding,

## Statement of Teaching Philosophy

Christopher Kauffman

Fall 2016

**Falling off tables** The code was out of reach. I could use the mouse to hover over those lines on the dangling white projector screen, but I suspected a tiny wriggling arrow would not inspire focus from the CS 222 students in the room. I walked over to the screen and stretched up. No good: the first lines in the loop were at least three feet above my outstretched fingers. I turned to the class and could see minds starting to wander toward YouTube, Facebook, and the myriad of other distractions lurking on those open laptops. I tried jumping and pointing which drew some giggles, but I was still about a foot short. Okay, two feet short. Realizing this was an elementary engineering problem, I started with the most sensible engineering solution - I hauled a folding table beneath the screen and proceeded to climb on top. Much better: all the code was well within reach. I pointed at each line as I explained the execution path, moving down through the prints and variable increments then back to the top as the `while(i > 0)` loop repeated itself. Delight murmured from the audience. YouTube seemed far away. After a few iterations, the loop condition stopped being true and the program fell off the closing curly brace to completion. Similarly, I fell off the table. Part of the blame lay with the shoddy build quality of GMU's classroom furniture as the table was definitely in the lower stability quartile of movable objects I have stood upon. But then again, my paltry vertical jump is complemented by a relative lack of gymnastic balance which is to say, I did not stick the landing. Dismounts were not part of my Preparing Future Faculty course in grad school. As the class exploded and a few students rushed over from the first row to help me up, it was I who giggled from the lesson. No more table standing. I ordered an extendable pointer that evening, the little kind that doesn't take batteries and is three feet long. Low tech for certain, but it may save my life one of these days. Unless I lose the pointer. Better order a backup.

Most of my class sessions are not quite as bruising as this incident, for me or for students. Still, though, I endeavor in each meeting to bring a sense of engagement and urgency to the room. We have so little time with students: 2.5 hours each week for roughly 15 weeks amounting to a paltry 37.5 hours. Every minute in that room together is precious and when students' minds are elsewhere, it is a golden opportunity lost. I have devoted a lot of time and effort over my short teaching career to craft an experience in class meetings that draws students in and makes them willing to take risks in class. I will follow with some examples of my methods to coax students, gently but insistently, just outside their comfort zone where learning happens.

**What do you think?** When I was a junior studying computer science, I took a course on data structures and algorithms. Graduate student Robert Bodor led my discussion section and in our first session did something amazing. As was standard practice, Robert set up a problem on the whiteboard as we half-sleepy undergraduates shuffled our papers around and feigned interest. Well, I wasn't feigning which I suppose was an early sign of impending graduate school. After getting the necessary code written on the whiteboard, Robert turned to the class and asked if anyone could analyze the computational complexity of the algorithm. As was standard in such cases, all students stared intently at notes, textbooks, belly buttons, or anything else that was not Robert. The protocol here was that, after an awkward pause, instructors would reveal the answer, students would copy it down, and we could all get on our merry way. Robert broke protocol. He looked directly at my dear friend Andy Blair and spoke: "*What do you think?*" Everyone stared in shock. Andy's eyes went wide. "Um... I'm not sure. I think it's related to the size of the

input array.” Robert paused then replied, “That’s right, but how is it related? How about you?” Robert’s gaze had jumped several rows back and his finger traced a laser sight at another member of the class. Everyone in the class was suddenly engaged. It was apparent that any of us might be responsible for answering any of these questions. Discussion section suddenly involved actual discussion.

Andy Blair later went to law school and told me most of his law school professors employed a similar direct questioning approach and that it had the similar effects: students prepared for class so that they were not caught off guard by questions. I became friends with Robert a few years later when I started graduate school. He said that our section was his very first experience teaching a class and that he somewhat regretted his blunt approach to generating classroom conversation. I do not. That sense of immediacy, of being responsible for answering and justifying, is something that I want my students to feel each session. Learning to take responsibility for understanding the quality of one’s own answers is the first step towards mastering any discipline. Instilling that sense of responsibility during class sessions is paramount.

**Hot Seats** Direct questioning is how I spur conversation in almost all my classes. I have refined a participation system over several years of use which I refer to as “Hot Seats.” Students may elect to sit toward the front of the class and may be asked to give answers to direct questions. Passing to a neighbor is acceptable, but honest attempts are rewarded as are follow-up questions. To track contributions, I hand out playing cards to those who answer. At the end of each class session, students who have accrued cards report their count on a sign-out sheet. As was my impression as an undergrad, suddenly singling out a student can be a jarring experience for that individual. However, in every class I have taught, students quickly adapt. Those inclined to participate in our conversation congregate at the front, use the allotted class time to work on exercises, and practice presenting their ideas in a public forum. They earn cards for coming prepared and staying engaged. Students nervous about being randomly selected move back a bit and ask questions when they need clarification, still hopefully involved but without danger of being put on the spot.

Most of my direct questions concern in-class work, particularly practice exercises. My common strategy during class meetings is to introduce a concept in the first 20 minutes, then shift immediately to an exercise which students work through to solidify their understanding. While they work, I wander the classroom helping folks get started, prompting deeper inquiry, answering questions, and frequently losing my coffee mug on a random desk. I encourage students to justify their answers to their neighbors during this time, which leads to collaboration and discussion. After closing down work time, I will query students in the hot seats for answers. For problems involving computer code, I will write student code on the projector, compile, and debug it so that students can see how mistakes can be fixed. We often explore several different solutions so that students get a sense of the variety of plausible programs that can work and begin developing their sense of elegance.

**About Those Bonus Cards** Students who contribute to the class get credit for it. I feel this is important, that students engage much more often if there is even a small incentive involved. Handing participants a physical object, playing cards in my case, cements the fact that such contributions are appreciated and worthwhile. A classroom observer once reported hearing a student mutter, “One of these days I’m going to get one of those cards,” affirming the motivating potency of bonus cards.

Participation in most of my classes is purely for extra credit with students maintaining good records getting a maximal bonus. Bonus cards are a kind of currency which can be used for other purposes aside from Hot Seat accounting. My realization of this started as an off-hand remark at

the end of one session concerning a tough problem that we did not finish. As we were closing down, someone asked, “Will the answer be posted?” to which I replied, “Probably not, but if you come in with working code for it next time I’ll give you three cards.” This tiny incentive motivated the student to mail me a solution later that night prompting 10 minutes of the following class to be spent analyzing the student’s code. Cards are a currency to motivate, to encourage, to reward. Student contributions outside of class also garner cards. Examples include answering the discussion board questions of other students well or bringing news items of interest to the attention of the class.

A particularly rewarding opportunity for bonus cards arose in my CS 310 class a few years ago. Examining the textbook, I found that the author made a claim regarding an unfamiliar property of a data structure we were studying. The only evidence to support the claim was a reference to an academic paper. On perusing the paper, I found no evidence to support our textbook, a dissatisfying result. With research time in short supply, I explained the omission to the class, asked that they write nasty letters to the textbook author, and issued a challenge: 10 bonus cards to anyone who could produce evidence that proved the property. A few days later a student sent me a detailed analysis of simulations he ran, complete with graphs and code supporting the property. A day later, a second student sent me links she discovered to older academic papers in which the property was formally proved. A third student volunteered to update Wikipedia so that the data structure article reflected this kernel of truth for all to benefit. The cards flowed and it was a proud teaching moment.

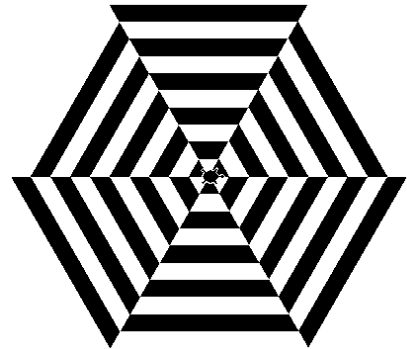
**The Game’s Afoot** The card system bakes naturally into review games we play in class. I favor “Java Jeopardy” in some of the programming classes in which we adapt Jeopardy to the large classroom setting without buzzers and with exam style problems. Review days are a little crazy with me dashing from one student to the next to evaluate correct answers and doling out cards based on question difficulty. Students get a chance to practice exam problems under time pressure, see answers and ask follow-ups, and get access to all the questions in the game after class for their reviewing pleasure. Course feedback usually includes a few one line comments to the effect of “Love Java Jeopardy!”

Students in my CS 105 Computer Ethics and Society course get a different taste of active learning to underscore the difficulty of ensuring software is safe in critical systems. One of the more famous examples of a deadly failure of computer code was the THERAC-25 radiation machine. Due to faulty code and poor product testing, the machine over-exposed a number of cancer patients leading to their deaths. We demonstrate the flaws of the system in class by students acting out the flawed program and radiation machine while a cancer patient waits apprehensively to see if the settings lead to a cure or to death.

Students find it easy to look at the designers of the THERAC-25 as unethical without considering the pressures that exist that can give rise to such mistakes. Competition combined with limited time and personnel to test a product often lead to rushed development and over-confidence. To simulate this setting, we play a team-based game in which students must answer a variety of questions for real class credit. As in true engineering situations, no single participant can win alone requiring students to rely on their teammates. The exercise also encourages students to assess their confidence in teammates in order to adjust the risks the team is willing to take for greater rewards. A central lesson in accountability is to consider how much risk an engineering team should ethically take on. Many students initially express a “go big or go home” attitude which may work in some settings, but does not keep planes in the air. This is illustrated by the 50% failure rate of teams in the game who tend to favor high risk choices and find they are not up to the challenge. Feeling such a failure in an academic setting may seem a bit harsh, but I

would much rather students have this experience in the safety of the “ivory tower” and adopt a conservative stance when it comes to engineering real-world systems where lives are at stake.

**Know Your Audience, Challenge Them** I teach a mixture of computer science courses for CS majors (CS 211: Object Oriented Programming, CS 310: Data Structures, CS 499: Parallel Computing), a programming course for engineers (CS 222: Computer Programming for Engineers), and several computing courses for non-majors (CS 105: Computer Ethics and Society, CS 100: Principles of Computing). The latter two fulfill part (CS 105) or all (CS 100) of the University IT and IT/Ethics requirements and are usually populated by a diverse group of students, many of which will not practice computing to make their way in the world. Despite this, it is crucial to illustrate to all students how their lives and careers will be impacted by computing technology. College-educated humans should understand how to leverage computing and avoid being replaced by an automated system. While it is challenging to teach non-majors about computing, my students have excitedly posted their very first hand-crafted web page hosted on Mason’s servers, developed a sense of empowerment by upgrading their personal computer security, and felt the thrill of victory when they finally perfect the Python code to draw the intricate Hypnoturtle (see image).



*The “Hypnoturtle” featured in CS 100 HW3; non-CS majors gain quite a bit of experience with loops in order to draw this creature.*

Meanwhile, I have developed a reputation in the department for the programming projects I concoct for CS majors in CS 211 and CS 310. My brother-in-law Kevin, a professional programmer, examined an assignment I was developing with fellow CS instructor Mark Snyder. Kevin commented, “What are you trying to do, train actual, solid developers or something?” Kevin frequently complained that his undergraduate CS education did not hone his coding ability and I aim to ensure that does not happen at GMU on my watch. While talking to a class about her internship, an undergraduate teaching assistant recently said this of the coding assignments: “My internship projects at Google were not as intricate as CS 310 projects. You’ll be ready to code at Google when you finish this class.” Learning to program is not easy, but can be a lot more enjoyable if the programs themselves are engaging. I meticulously craft CS 211 and 310 assignments to ensure they are challenging and amusing. I try to hit topics of interest for students, incorporating cryptography, popular online games like 2048, the underpinnings of spreadsheets, and simple database implementation. Many students start out daunted and end up experts. I issue midterm surveys every semester, and the resulting feedback commonly shows appreciation for the detail and guidance provided in the assignments.

**DegreeProgram++** Continuous improvement of individual courses and the CS program at large is important to me. For example, the teaching staff in CS 211 and 310 initially faced trouble when evaluating whether student programs run correctly. Grading can be a very tedious affair where graders are lost on what to look for and students end up frustrated that the evaluation scheme was not reflected in the assignment description. Working with other CS instructors, I have implemented a concrete solution to this problem. Computer programs can evaluate other programs so half of every assignment is graded automatically via testing programs which students can access while they work on their assignments. They have immediate feedback and a sense of progress as their programs progressively score better on the tests. The other half of the assignments are graded on a manual inspection rubric which graders follow to look for things that only a

human can check, qualities such as code elegance, efficiency, and documentation. Students have the rubric as they work, giving them a solid idea of our expectations. Several other instructors have adopted variants of this split evaluation for programming projects. I have also helped to plan out how the CS 211 labs are utilized and found that dividing time between collaborative exercises, in-class quizzes, and programming assessments works very well. The variety of activities gives students with different strengths multiple chances to prove their mettle.

Within the computer science department I am an active participant in the Undergraduate Studies Committee and have contributed to our curriculum reform and program assessment. I am the primary author for a CS Departmental Honors Program which was initiated during the Fall 2016 semester. This program gives our most talented students a vehicle to earn additional recognition for deepening their computing experience. I was touched when the CS faculty passed the proposal unanimously and our department chair, Sanjeev Setia, commented that my work was “among the most detailed and well-thought out proposals” he had seen come through the department.

Outside of the classroom and meeting room, I keep busy with a variety of teaching-related service activities. I presently advise 65 undergraduate students. Seeing the relief on students’ faces as I help plot their last semester before graduation is a joy. I have also advised six students on OSCAR research projects, two of which received the GMU’s URSP award and four others through the OSCAR Federal Work Study program. Projects span a variety of areas include an AIDS education game, an efficient machine learning library interface, and a natural language translation project. I am the proud faculty mentor for the PatriotHackers, GMU’s computer security club, and have helped them to secure computing resources on campus.

**Seeking Sustainability** While I am committed to teaching, it is difficult work. I am a recovering perfectionist and coming to grips with the time constraints imposed by our frenzied academic schedule has been a struggle. There are never enough hours to perfect the humor and nuance I want in my lectures. I am consistently behind in advising the undergraduate researchers who seek guidance from me on our projects. The discussion board continuously beckons for answers and students have bugs that they need help squashing. I have also had my share of discouragements such as managing to schedule a major assignment deadline the night before a midterm exam. Lesson learned, never again.

Reminders that my teaching matters help buoy me against such setbacks. Student ratings and feedback help: a few months after each semester ends, I receive the university reviews. They are usually positive with a good distribution of tips to improve and praise in the free-form feedback. The reviews offer a bit of a refresher in the midst of subsequent semesters that, on average, students appreciate what I do. Recognition from colleagues is also encouraging. I certainly felt honored when my department recognized me with its Outstanding Teaching Award during the 2014-15 year.

But the best educational energy drink I have quaffed thus far was an unexpected visitor in my office during Spring of 2015. Sarabjeet Kapur fought through my CS 310 course, conquered the tricky projects, passed his later CS courses, and won the privilege to speak at the Spring 2015 graduation ceremony. He was ecstatic as the intense data structures work he had done under my tutelage had paid great dividends. Sarabjeet aced interview questions and was offered several jobs at high-end companies in Northern Virginia. He said it was a high point in his life. Sarabjeet visited me to offer his thanks and to personally invite me to come listen to his graduation speech.

I will not elevate every student to enlightenment. That is still code that is out of reach. But a student like Sarabjeet every now and then will keep me dealing out cards, plotting projects, and falling off tables.

## 5 Reflection

The specification for elements that should appear in this document mentions four criteria by which it will be evaluated by reviewers. These are as follows.

Criterion 1: Evidence of Growth and Development as an Educator

Criterion 2: Evidence of Student and Learner Engagement

Criterion 3: Evidence of Assessment of Student Learning & Achievement

Criterion 4: Evidence of Teaching Effectiveness & Impact

These criteria are interesting to me as they reflect, with a slightly different order, the basic cycle that most scientific endeavors employ. My training in graduate school honed instincts for scientific inquiry so it should come as no surprise that my teaching has taken the same flavor which is roughly the four criteria re-ordered:

**Repeat Until  $\infty$**

1. Engage
2. Gather Evidence
3. Assess
4. Grow

In this largest section in the portfolio, I will attempt to isolate and describe several iterations of this cycle as it manifested in my  $4\frac{1}{2}$  years of teaching. None of these developments is finished as we are presently in the middle of another semester of teaching so the cycles continue. However, this is an excellent chance to reflect upon how those steps have played out in various aspects of my teaching thus far.

### 5.1 Lecture Discussion via Hot Seats

As I describe in my teaching philosophy, I drew early inspiration on how to encourage classroom participation from several sources, notably graduate teaching assistant Robert Bodor who would randomly select folks from the audience to ask questions, a practice my friend and classmate Andy Blair told me is oft-employed in law school where he was subjected to it again later in life. In my first full-time teaching role, I was interested in utilizing the technique to encourage in-class discussion but wanted to make sure that students actually felt it was worth their while to engage.

This first iteration of “Hot Seats” appear in the syllabus for the first class I instructed, CS 2011 “Discrete Structure for Computer Science” during summer of 2011 (irony) at the University of Minnesota (Syllabus excerpt in Figure 1). I aimed to asked students frequent questions and

Figure 1: *Excerpt from CS 2011 Syllabus*

**Bonus credit** will be awarded based on participation in class discussions. The instructor will randomly select names from the class roster during class to assist with problem and concept demonstration. Students present that exhibit honest effort and attention are awarded a point. Occasionally extra credit problems will be assigned to be done outside of class. These will count towards the participation score. The highest point winner at the end of the semester will receive a 3% bonus to their overall score in the course. All other students will receive a bonus proportional to the highest point winner. For example, someone tied with the highest point winner will also receive a 3% bonus while someone with half the participation points will receive a 1.5% bonus.



then keep a tabulation of their effort on a course roster each time they participated. My theory was that this would allow me to learn names faster, spur conversation, and reward students who were attending and engaging regularly. I picked the particulars for the following reasons:

- Quantifiable: There are actual numbers that students can see, count, and gauge their participation on.
- Self-balancing: If someone stops showing up frequently, their contributions are not lost entirely but become less over time compared to those that continue their effort.
- Inclusive: Every effort is worth some bonus. If everyone participates equally, everyone gets the same bonus (though this has not yet happened). Passing a few times doesn't hurt compared to long-term effort.
- Small but Significant: a 3% max bonus is enough to bump folks over a grade break (B- to B, B+ to A-) but not enough to turn a C to an A.
- Cheap: No clicker technology or subscriptions required for students.

I continued this same strategy in CSC 301 at Concordia University in Fall 2011.

For these relatively small courses ( 25 students), the system worked well: students didn't take long to get over their initial aversion to being called on and appreciated the opportunity to garner extra credit. I noticed using the course roster to select names randomly ended up being a bit of a wash as my calling on a student who was not present would result in crickets until I learned names with faces. It was also a little distracting as students answered questions for me to break eye contact to place a mark on a sheet of paper next to their name. Beside the benefit of generating bona fide conversation, these were minor inconveniences but I was still interested in refining further.

When I started at GMU in Fall 2012, my first large programming class was CS 222 "Computer Programming for Engineers" which was twice the size of my previous classes. I was worried that there would be an uneven distribution of participation and elected to try rotating the alphabet for hot seats, A-L for the first 1h 15m, M-Z for the next half of lecture (see Figure 2). These fears proved to be unfounded:

students were mildly annoyed by the need to shuffle around in the room halfway through lecture and those interested in participating would simply ask questions from farther back when booted from the hot seats. After a month of class I canceled the rotating alphabet policy and allowed anyone willing to come to the front of the room. Rather than call out names from a roster, I would randomly select folks actually in the room which cut down on awkward pauses. I asked that students write down how many times they participated on slips of paper which they handed in at the end. This proved a bit logistically challenging as most students would tear off a small piece which inevitably got stuck in the bottom of a folder. I noticed several students all writing their count on a single sheet of paper before handing it in; this was much easier for me to enter into a spreadsheet anyway so became the new policy in the next session: everyone writes their count on a sign-out sheet.

It was not until the next semester while instructing CS 211 that it occurred to me to hand out some sort of token to indicate the participation. I am not worried about dishonesty in

Figure 2: *Excerpt from CS 222 Syllabus*

**Bonus credit** will be awarded based on participation in class discussions in lecture and online. Each course meeting will be divided into two halves. Before each half, I will announce a range of the alphabet. Students with last names in that range may elect to sit in the first 2 rows of the class and answer



Figure 3: *Notable Women in Computing Cards used to count class participation.*

reporting counts. Rather, I was interested in providing a visual reinforcement of how I valued their engagement. I wanted to actually hand something to students that answered questions are contributed to the class discussion. This would, in my mind cement the “what you have to say is worth something” sentiment I want to pervade my classes. At the end of each class session, students would return their participation tokens and write down their daily total on a sign-out sheet. I enter the data into a master spreadsheet (again helping to learn names) bonuses are proportional to the total accumulated tokens by the end of the semester.

While teaching CS 105, I had need to randomly assign students to groups of 4 a few times and had used a deck of cards for that (find others with the same face value that you have). I reasoned that playing cards are easy to replace if students lose them and thus was born the “Kauffman Cards” which are a staple of many of my classes. Participants who contribute to class discussion get a card or two. I have used various items for participation, including computer punch cards from the computing my Dad did in college, but lately favor the “Notable Women in Computing” cards I obtained at a CS education conference (Figure 3).

Student feedback on Hot Seats has been positive frequently coming up in course feedback forms as a high point. I asked about the participation frequently on midterm feedback surveys and students generally respond positive to neutral on the issue as evidenced in the sample result from midterm feedback in CS 310 in Figure 4.

When I solicited feedback in preparation for this portfolio, variety of students mentioned the system as memorable. Here are a few comments from them which summarize well.

Professor Kauffman encourages class participation by offering a reward system for those who participate. As a rather introverted student, this really helped me to begin speaking up in my classes. —*Briana Abrams*

His sense of humor, engaging presentation style and famous “Kauffman-cards” (playing cards which could be earned for class participation and traded in for extra credit points) made every one of his classes very interesting and enjoyable. —*Ananya Dharwan*

Chris is master of the surprisingly fun and effective “hot seat” participation rule that I believe more professors should adapt. While initially I thought hot seats would be comprised of the same students answering all the questions, it quickly became apparent that these seats were sought after by many students. A large variety of

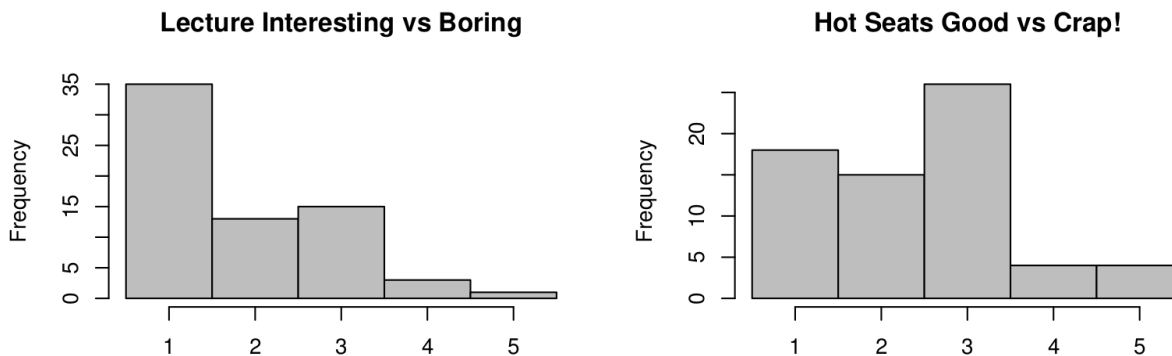


Figure 4: Feedback on hot seats from a CS 310 midterm survey. Answers of 1 indicate strong approval and 5 strong disapproval.

students choose to participate, and it was actually difficult to get a spot in the hot seat section sometimes, prompting students to come earlier to class. —Bridget Lane

I have attempted few times to set up a session at GMU’s Innovations in Teaching and Learning to discuss the system a few times as I think others could benefit from it. I met with some success this past fall when I was asked to do a poster on hot seats as part of the SIMPLE teaching development group I run with. While a static image doesn’t necessarily convey the whole story, it does fit in a portfolio nicely so examine Figure 5 for a nice summary.


## 5.2 In-class Activities

My basic cycle in most class meetings is summarized well by a former student.

A class with professor Kauffman begins with announcements, followed by student questions about the homework, a quick review of old material, an introduction to new material, and some examples to explain the new material in depth. Then comes my favorite part – as it is perhaps the most important aspect of learning, yet overlooked by so many: professor Kauffman make sure to provide students with 5 to 15 minutes of time where he encourages us to collaborate and discuss among ourselves the different aspects of some interesting and well defined problems. He walks around the class while we discuss and takes some time to talk with each group individually. He listens to our ideas and guides us towards the right answer. He then discusses the answers with the class as a whole and opens the floor to any students who want to share their particular approaches. He answers any questions that arise and concludes the class by summarizing the new concepts we learned that day. —Ananya Dhawan

I adopted this approach early on in teaching due to guidance from the instructors in a preparing future faculty course I took in graduate school. My instincts have always been that students will benefit far more from actually doing something in class rather than just listening to me monologue. If they are going to program, have them write programs in class. If they must problem solve, give them problems to work on. If they need to analyze ethical dilemmas, put them on a bridge with an out-of control train, some people on the tracks, and a lever that dictates who lives and who dies.


I have learned a lot about how to structure activities well.



## Hot Seats: Large Classroom Engagement through Measurable Participation

Chris Kauffman, Dept. of Computer Science, GMU

This project is supported by the National Science Foundation, Division of Undergraduate Education, under Grant No. 1347675.



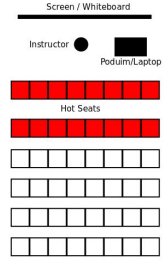
### Typical Large Classroom Problems

- Initiating Proper interactive discussion is difficult: many students prefer anonymity of silence
- Invisible wall between instructor/class
- Gauging student understanding difficult without feedback through discussion
- In the absence of technology (clickers) measuring participation in large classes is difficult, tedious, or both
- Single student may attempt to dominate discussion


### Typical Lecture Setup

Screen / Whiteboard

Instructor ● Podium/Laptop



Hot Seats



Playing cards are a readily available and inexpensive token to hand out as students answer questions. I use souvenir cards from trips/museums.

### What are the logistics?

- Hot Seats are seats close to the front of the room students may elect to sit there
- During discussion, questions are directed at students in hot seats
- Effort and answers are rewarded with a "token" physically given to participant
- Unprompted questions from any part of the room also nets a token
- At the end of class students return tokens and record how many they received on a sheet of paper at the front of the room
- Works well with in-class activities: rewards for contributing answers after 10-minute work period
- Instructor tallies total participation over the course of the semester: quantifiable participation based on token count
- Frequent Q&A shows temperature of the room; can speed up or slow down presentation based on answer quality
- Emphasis on discussion encourages all students to ask questions

### Hot Seats Help Solve these Problems

- No major technology required
- Incentivizes optional participation or provides easy accounting of required participation
- Creates an atmosphere of discussion rather than lecture

### Sample Adaptations

Situation	Variation
Very large class	Rotate hot seats by last name: A-K, in lecture 1, L-Z in lecture 2
Participation Required	Notify students they must reach a certain total by end of semester
Participation Optional	Top token-earner gets a 3% bonus, others get proportional bonuses
Single dominant student	Rotate who gets to answer to spread tokens around

Figure 5: ITL 2016 Hot Seats Poster

- More instructions are better than fewer so students have guidance while they work
- Sitting down next to students who are daydream to ask them about how their solutions and analyses are progressing is a good way to refocus their attention.
- Interrupting the class to announce a clarification more than two people asked about is worthwhile
- Students always need time to digest what is to be done initially so silence at the beginning is not a bad sign though silence at the end may be.
- Students rarely move as fast as I like but attempting to rush them is not effective. Allowing 10-15 minutes for a moderate programming problem is fair along with 10-15 minutes to discuss solutions.

This section surveys some of the activities that have clearly evolved over my teaching career. It emphasizes what I have learned about how to direct student effort during in-class activities to maximize their time and learning efficacy. I will sample a variety of in-class activities I have used and how they have changed over repeated offerings.

### 5.2.1 CS 105: First Day Intro to Ethical Theories

The first topic in CS 105 "Computer Ethics and Society" is to survey three standard ethical theories: Utilitarianism, Kantianism, and Social Contract Theory. This can be a rather dry affair but the year

before teaching the course I had the good fortune to attend a presentation by Harvard philosopher Michael J. Sandel. He used an opening example which I have shamelessly stolen and refined.

**Utilitarian Scenario:** Imagine you are on a bridge which crosses a series of railway tracks. From your perch, you see a group of three workers doing heavy maintenance on a track one side of the bridge and a single worker similarly operating a jack hammer on a side track. A loud noise alerts you to the fact that a train is rapidly approaching from the other side of the bridge which will in moments strike the three workers who are unaware of its approach due to their work. You notice that there is lever within your reach which would switch the track the train is on to collide instead with the single worker. *Should you pull the lever?*

Thrusting students immediately into this severe situation on day one startles some but most are intrigued.

CS 105 has a large population of computer science students who are naturally inclined to probe the situation for exploits: Can the workers be alerted by shouting or throwing things? Can the lever be pulled halfway to stop the train? Are any of the workers older, younger, male, female, etc? Will anyone see me pull the lever? With practice, I have eventually learned to answer all these queries in unsatisfying ways that force the students to ultimately make a choice about who lives and who dies with very little information. To underscore the need to make a choice, I give students a moment to contemplate before casting a vote to “Pull” or “Not Pull” and tally the votes on the board.

While somewhat morbid, Sandel used this initial example to demonstrate an ethical principle: many folks will pull the lever to save three people versus one. To test this principle, I ask students who do not want to pull the lever to keep their hands in the air while I increase the number of people who would be killed and lower their hand when they feel compelled to pull. This is the Utilitarian philosophy, that a decision is moral if it results in more happiness, in this case more lives saved.

It is not the only way to view the problem though. The second variant reads as follows.

**Kantian Scenario:** Imagine again that you are on the bridge with an out of control train barreling down on three unaware workers. There is no lever this time but standing ridiculously close to the edge of the bridge is an enormous man. You realize that with just a small push you could unbalance the man and cause him to topple down onto the tracks. Due to his enormity, the train would be significantly slowed on striking him, the workers would notice the commotion, and they would be able to escape.

In most sections, there is a dramatic shift from the first scenario with many fewer voting to “Push” and many more voting to “Not Push.” This is despite the similarity to the first scenario of one life versus three lives. I ask folks who voted to “Pull” earlier but to “Not Push” this time to explain how the scenario seems different. Ultimately this results in identify an alternative to Utilitarianism, that it does not feel right for many to use a person to stop a train even if the goal is to save other people. This principle is roughly formalized in the Kantian philosophy as “Do not use a moral actor as a means to an end, no matter how noble the end may be.” As a sample, I ask students if they would be willing to push a large sack of flour off the bridge to save the three people. Few are unwilling to sacrifice the flour for human life demonstrating that the enormous man is being treated as less than human if he is nothing but a train stopper.

We also discuss Social Contract Theory which Sandel did not work into his original example. I struggled for a couple semesters to determine whether another example was needed but finally settled on the following final variant which does the trick.

**Social Contract Theory Scenario:** Imagine the same scenario with three rail workers about to be crushed by a train unless an enormous man is pushed off a bridge to save them. Each of the five people, three workers, enormous man, and decider, will eventually be assigned identifies from students in the room. Each student is given a playing card and the instructor will draw cards from a similar deck to decide which students end up in the scenario in which roles (only 5 of the class will). If a student “dies” in the scenario, they lose credit for their first day of participation, about 2% of their overall grade.

Before the roles are assigned though, students must discuss and vote on a Law to guide the action of the decider who must “Push” or “Not Push” the enormous man. On being assigned the role, the decider is free to make their choice but goes to jail if they break the Law decided upon by the group also forfeiting their 2% participation.

I was very happy with this setup as it fits Social Contract Theory quite well, introducing the idea of the *Veil of Ignorance*: students must decide what law will benefit their small society before knowing who they will be in the scenario, unaffected, on the tracks, or about to be pushed off a bridge. This aspect of social contract theory is difficult for students to understand and having this opening day example to refer back to is extremely helpful. The tension that mounts as I draw cards and write student names on the white board next to various stick figures on the tracks and the bridge leaves a lasting impression on most classes. The exercise culminates with one randomly selected student who must decide whether to push or not push with other student’s credit on the line.

Ultimately I would never deduct 2% from student grades on day one purely by chance and admit as much to them after the activity concludes. That would be unethical. However, I am not above creating some dramatic tension through deception as it markedly increases student engagement.

### 5.2.2 CS 211: Day One Exercise

I like to hit the ground running in all my classes on day one as reading out of the syllabus would put me to sleep as quickly as the students. While ethics classes can deal with life and death situations, programming courses need a bit more structure. In CS 211 “Object-Oriented Programming” my day one exercise has evolved quite a bit over time. Initially it was as sparsely defined as possible which was an inappropriate choice under the light of experience.

As a course, CS 211 is about designing somewhat larger programs aided by techniques from object-oriented discipline. My initial reasoning is that the day one exercise should reflect this intent by essentially a thought exercise to design something reasonably large. The card game “Go Fish” seemed a plausible candidate so I chose this and issued a fairly barren slide as seen in Figure 6 for 2013.

My plan had always been to lead a discussion of the rules of the game but I distinctly recall spending most of the discussion answering questions about various facets and helping students to recall the rules of “Go Fish.” This was an early and crucial lesson which guided my thinking on lots of programming projects to come. One must be very careful picking the problem domain of the programming problem as students unfamiliar with it will be at a huge disadvantage. My long-time collaborator on CS 211, Mark Snyder, conveyed a similar experience to me after I told him about the exercise. He explained, “I never would have thought it but when I assigned students to implement a card game one year, an international student approached me with the concern that his culture didn’t play *any* card games so he was completely lost.” Subsequently, I have also been very careful to pick problems which students can gain familiarity with rapidly. It is impossible to



Figure 6: Day 1 exercise in CS 211 developing over several years

program something correctly which you would not be able to do or predict by hand so getting students enough information in the form of demos and playable models is crucial.

After a couple years of “Go Fish” I decided to rework it due to its myriad of deficiencies.

- Students would have a hard time remember the rules to the game.
- It was too big of an exercise to feasibly finish even a rough sketch in the 15-20 minutes I wanted to allot.
- The large-scale design principles that would make for a beautiful solution were not yet in the student toolset on day one.
- It did not give me chance to examine what reasoning and programming skills students had acquired in their prior programming experiences.

All of these caused me to scrap the exercise entirely and pick something fresh. Importantly though, the lessons I identified from what went wrong with Go Fish could be baked into the next version for improvement.

Every semester the teaching team for CS 211 struggles to balance the grading load among graduate teaching assistants (GTAs) equally. Prof. Snyder and I often compute the *discrepancy*

between GTAs which is the difference in head count between GTA with the most students in their labs and the GTA with the fewest. Having just engaged in this exercise in a staff meeting in 2015, I quickly formulated it into a programming problem for students in CS 211 for their day one exercise. Its description is much more structured as shown in Figure 6 in the 2015 panel. My experience with it over two years was as follows.

- There are several correct ways to solve the problem so good discussion can be generated from the problem. It often illustrates whether students have gotten prior exposure to certain useful data structures which lets me know what to expect from them.
- Students with a procedural programming background (CS 112 at Mason) should be ready to solve this problem. If they struggle, I site it as a flag that they may be rusty apply effort to get in shape quickly.
- Many students struggled with how to initially read the file which contains the problem data. They ended up spending a significant amount of time on this ancillary part of the problem and missed the opportunity to demonstrate their programming skills on the more essential aspects.
- By updating the data each year, I can introduce students to the names of the lab leader GTAs on day one which is a nice side benefit.

After two years of experience with the grading discrepancy problem, I introduced this year a final alteration to correct the file processing deficiency identified above. This final variant is in Figure 6 2017 which I utilized just a few weeks ago. I am more satisfied with the day one results than I have ever been. With 15 minutes of work time and some guidance from me, we generated several solutions that gave students a good sense of whether their programming chops are in shape and my style of work time during lecture. This success I can attribute to observing students carefully during the first day and making both mental and digital notes about the activity.

### 5.2.3 Exam Review via Java Jeopardy

I feel that class time to review prior to exams is worthwhile and also appreciated greatly by students. That said, the initial structure of how to do so effectively escaped me. After all, if content for 4-8 weeks of class could be covered in a single class session, what was going on in the past 4-8 weeks?

My initial foray was to provide a series of review questions which we went over as a class according to my standard pattern: introduce the problem, 10-15 minutes of work time, discuss the answer as a class. An example of this is given in the slides shown in Figure 7. I used this style in CS 222 and CS 211 the first time I taught each.

Review questions worked alright but did not seem to give an appropriate signal of the impending exam as it was so similar to what we had done in most previous classes. In part, it also did not give students the sense of risk and reward that exams often have: answers on in-class exercises could be loose or flubbed entirely with little consequence.

About that time I had a conversation with my one of sisters in which she proclaimed satisfaction at completing a guest lecture and review for a pharmacy class. I asked her how she had conducted her review to which she replied she had used PowerPoint to create a version of Jeopardy!, the popular TV gameshow, to make the review a game. I lamented the fact that there were so many students in my class I could not use a similar technique nor easily divide the students into teams to make it work.



## Exam 1 Topics

- ▶ Basic Memory Diagrams
- ▶ Writing Loops and Conditionals
- ▶ Equality
- ▶ Simple Input/Output
- ▶ Defining Small Classes
- ▶ Debugging

Questions on any of these?

## Practice

## Define an Interval

- ▶ A 1-D range continuous range
- ▶ Has `start` and `end` fields which are real numbers public and constant
- ▶ Public constructor, 2 real number args
  - ▶ Lower of two is `start`, bigger is `end`
- ▶ Public method `duration`, takes no arguments, returns a real number `length` of interval
- ▶ Public method `overlap` which takes another `Interval` and returns true if the two have any overlapping portion

Figure 7: Midterm exam review for CS 211 in 2013 which comprised a few practice problems like the sample given.

The next day I began toying with the idea anyway. If I had a Jeopardy board to load questions into that would at least present students with a list of optional questions to choose for the review. Similarly, perhaps there was a way to adapt to the large classroom.

In the end, I derived a set of modified rules which are described in Figure 8. Someone selects a category and question from the game board. I read the question in as close an approximation to Alex Trebek as possible after which students work on the question. Students “buzz in” by raising their hands after writing down their answers. I will field the first three answers by wander the room and examining the written solution. My verbal instructions are that answers should be fairly close to what student would write on their exams to simulate preparation as much as possible. If I can’t read answers, they don’t count nor do broken fragments. This creates a realistic time/accuracy trade-off that is present during exams. Importantly, buzzing in and getting a question wrong loses points. I keep track of the scores for everyone in the class via a small text table that is updated after each questions similar to the table in Figure 9. The points are counted towards or against their participation score described in the section on Hot Seats.

I was nervous to try the game in Fall 2013 for the first time with my CS 310 class but delighted when it worked out well. Students enjoyed the small measure of control over selecting the questions during class and the fact that I provided the entire game board after class for further study. Students pre-disposed to rush their answers learn quickly to throttle themselves to avoid penalties. Lately I have become somewhat more lax in doling penalties out for incorrect answers which has encouraged students to rush too much. While a bit harsh to tell students to their faces that their solutions would not receive full credit on an exam, I must resolve to be disciplined about for fairness and to give them an adequate picture of exam expectations.

Overall student reception of the review strategy has been very positive. The only frequent request for modification is that I provide all of the answers to all of the questions. I am usually not inclined to do this straight-away however am always happy to discuss how to solve any problem with students willing to bring a partial solution to the discussion board or office hours.

Figure 9: Sample Java Jeopardy scores.

netid	score
cgrebe2	+3
nmehmane	-1
dpeacoc2	-1
zalfakir	+1
ctowsend	+2
fandica2	+1
rreegis2	-1

## Jeopardy

A quick demo for those not in the know: A famous game, watch from 8:30-11:00

- ▶ [Youtube Link](#)
- ▶ First to buzz in answers
- ▶ Question right: gain points
- ▶ Question wrong: lose points
- ▶ Get it right, pick next question
- ▶ Can pick anything available on the board

BST, Yeah You Know Me	Balanced Tree Cold War	Mind your PQs	Answer in the form of a question	Legen-... (wait for it) ...-dary
1 Card	1 Card	1 Cards	1 Card	2 Cards (H...
2 Cards	2 Cards	2 Cards	2 Cards	3 Cards (Oi)
2 Cards	3 Cards	2 Cards	3 cards	4 Cards (P...
3 Cards	3 cards	3 Cards	3 Cards	5 Cards (O...

## Java Jeopardy

## Problems

- ▶ More than 3 players
- ▶ No buzzers
- ▶ Harder Questions

## Solutions

- ▶ Answers are **NOT** in the form of a question
- ▶ Write your answer on a piece of paper
- ▶ “Buzz in” by putting your paper in the air
- ▶ I will assign ranks 1,2,3
- ▶ Correct answers get points for ranks 1,2,3
- ▶ Wrong answers lose points for ranks 1,2,3
- ▶ Highest rank correct answer picks next question
- ▶ One of rank 1,2,3 describes correct answer

Below is a binary min-heap which uses the 1-indexing convention. Draw the tree version of the binary heap after the below code finishes executing.

```
pq.deleteMin();
```

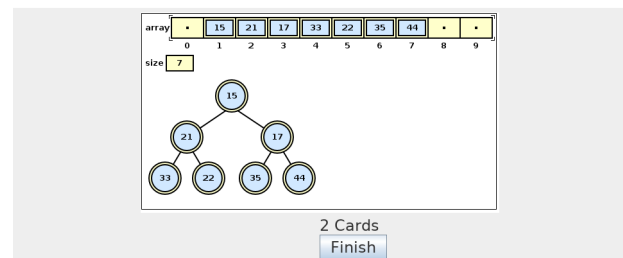


Figure 8: Slides and picture demonstrating the Java Jeopardy! midterm review game used in several programming classes.

### 5.2.4 Other Class Activities

I favor a *lot* of in-class activities exercises in my classes so it is not possible to track the development of all of them as I have done in the previous sections. However, a small sample of additional material is probably in order.

**Programming Exercises** I ask students to do quite a bit of live coding to prepare them for HW and exams. Examples include coding data structures, making use of new program elements, or introducing CS 100 students to their first bits of python. A couple samples appear in Figure 10.

**Analysis Exercises** Programming isn’t the only activity in computer science. A substantial amount of analysis is required in a variety of settings. This can range from highly technical areas in CS courses to broad social problems in general education classes. Regardless of topic, I ask my students to engage in critical thinking regularly. Some samples are in Figure 11.

**Collaborative Activities** In CS 105, I use an activity which places students in groups of “companies” which involve taking risks which can affect bonus credit they earn along with bonus credit

other teams will earn. It illustrates how group thinking can change as pressure mounts to take risks for greater gain, an effect which can take a heavy toll if engineering projects fail and the engineers are held to account. In CS 100, students group together to derive algorithms to sort numbers as quickly as possible. These must inherently be “parallel” algorithms if the groups are to gain anything through cooperation. A short competition to see which group has streamlined their approach the best shows the need for any parallel processing elements to communicate, coordinate, and that ultimately it may not be worth the price to parallelize an algorithm.

### 5.2.5 Feedback on In-class Activities

One of the most oft-mentioned facets of my teaching in feedback is the structure of my course meetings. Students appreciate the chance to flex their skills and get immediate feedback on their work. To some extent this is why I prefer the term “course session” to “lecture.” Below are a few quotes about the student impressions of lectures along with some sample midterm feedback data from students on lectures in Figure 12.

He would walk around and survey almost every student in the classroom to make sure that if anyone was having difficulty, he was there to help. Everyone loved this, because it was essentially like having a tutor. I feel as if a lot of students, when they have a question about something they don’t understand – and they are in a class of over 30+ students – are very reluctant to ask that question because they don’t want to seem dumb’, or have the spotlight put on them. So by going around and giving people personalized attention, Dr. Kauffman neutralized this sort of stage-fright’ that students get, and could really get down and help each student individually. –Taylor Kohler

Professor Kauffman’s goal is never to speed through as much material as possible, but rather to ensure that every single person in his lectures understands the material on the board. He encourages questions and participation in class, which help make his lectures my hands-down favorite in my college career. His lectures are also brightened by the pop culture references cleverly (or obviously) hidden in his slides, and even some tests. –Will Crosswait

CS 310 exercise		CS 100 exercise
<p>Define <code>bst.find()</code></p> <ul style="list-style-type: none"> <li>▶ <code>find(T x)</code> is publicly accessible</li> <li><code>tree.find("Mario");</code></li> <li>▶ <b>Define</b></li> <li><code>find(T x, Node&lt;T&gt; t)</code> which works on a given start node</li> <li>▶ Compare via Comparable: <code>if(x.compareTo(t.data) &lt; 0)</code></li> </ul> <p>Give 2 versions</p> <ul style="list-style-type: none"> <li>▶ Recursive</li> <li>▶ Iterative</li> </ul>	<pre>public class BinarySearchTree &lt;T extends Comparable&lt;T&gt;&gt; {     protected Node&lt;T&gt; root;     // Return x if in tree,     // null otherwise     public T find( T x ){         Node&lt;T&gt; result =             find(x, this.root);         if(result == null){ return null;}         else{ return result.data; }     }     // Find node containing x     // starting at node t     // Return null if not found     private static     Node&lt;T&gt; find(T x, Node&lt;T&gt; t){         // DEFINE ME     } }</pre>	<p>Exercise: Two Houses Single House</p> <pre># Draw the body of the house color("blue") begin_fill() for i in range(4):     forward(200)     right(90) end_fill()  # Draw the roof of the house color("red") begin_fill() right(300) for i in range(3):     forward(200)     right(120) end_fill()</pre> <p>Now <code>penup()</code>, change angle, move, <code>pendown()</code> and do it again</p>

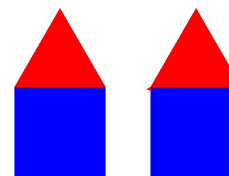


Figure 10: Sample in-class programming exercises for two courses.

## CS 499 analysis

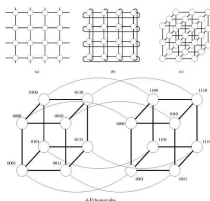
## Compare Networks: Parallel Sum

- ▶  $p$  processors
- ▶  $\log_2(p)$ -dimension Hypercube:  $(p \log_2(p)/2)$  links
- ▶ 2D-torus:  $2p$  links
- ▶ **Discuss** advantages/disadvantages of torus vs hypercube arrangement for this application
- ▶ Outline an algorithm, estimate cost-effectiveness

## Sum Array of Numbers

- ▶ Each proc holds a hunk of the data array
- ▶ Want a single processor to eventually contain sum of
- ▶ **State your algorithm:** Try to minimize communication at each step, exploit as much parallelism as possible

## Networks



## CS 105 analysis

## What If...



- ▶ Has the US ever been at war with Mexico?
- ▶ Implications for you?
- ▶ Any personal data out there that would affect others' view of you?
- ▶ Implications for certain groups of Americans?

Figure 11: Sample in-class analysis exercises for two courses.

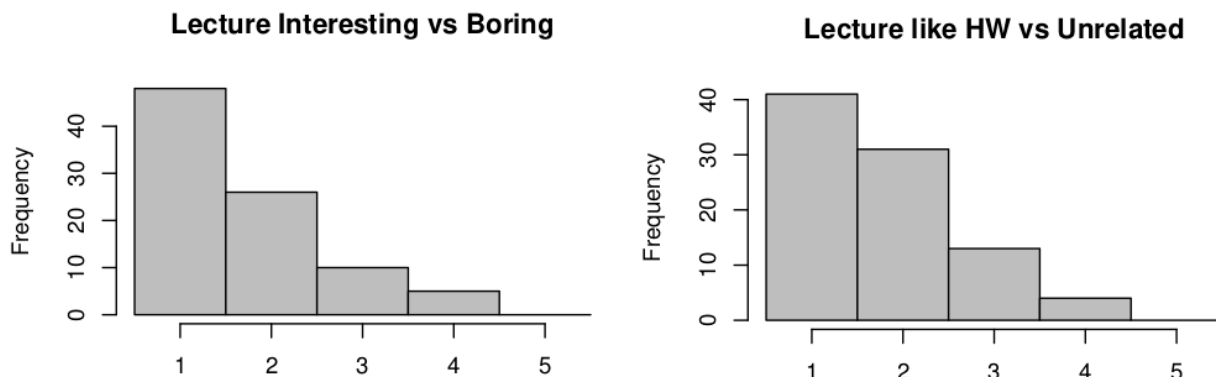


Figure 12: Sample midterm feedback on lectures. Responses of 1 indicate favorable impressions of lectures and how they prepare students for HW.

## 5.3 Assignment Specifications

While I enjoy leading class sessions immensely, I think my biggest joy in teaching is designing assignments. It is a creative activity of the highest order. Consider the variety of competing goals for any significant assignment:

- Introduce students to a new concept and allow them to practice
- Show best practices associated with that concept though it may not make sense to students immediately why certain conventions are considered best practices
- Challenge but not overwhelm students with a variety of innate talents and work ethics
- Relate to the “real world” in some appreciable way but not so much as to increase difficulty beyond the student’s level
- Concerns a topic that inspires curiosity in students while not excluding any

- Respect the time students have to complete it, usually one to two weeks with a maximum expectation of a few hours per day for that period
- Be gradable, potentially not by the assignment designer; allow well-defined partial credit for students who do not complete all parts completely
- Be interesting to teach

During part of my doctoral work, I studied what is known as *inverse problems* which start with a solution and attempt to derive the problem criteria which resulted in that solution. Course assignments are very much the same flavor and usually inspire all of my analytic and creative pistons to start firing.

In this section I will overview a few assignments which I have developed over my career and how I have grown to improve my pedagogy concerning assignment design.

### 5.3.1 First Passes

My initial assignment offerings when I started teaching were based on my own experiences as an undergraduate learning to program. They were lists of rather shortly specified problems. In some cases this was merited as there wasn't much need to explain the problem structure. However, I did not convey much information about how one might actually solve the problem nor remind students of their relation to in-class work we had done. There was also not much in the way of grading guidance to either student or the graduate teaching assistants which often did the bulk of the grading. Only a point value appears associated with the problem as seen in Figure 13. Aside from shifting from points to overall percentages for different parts of the assignments, this was largely the structure of my specifications for the first few years.

### 5.3.2 Testing Code with Code

Mark Snyder and Kinga Dobolyi introduced me in Spring 2013 to the use of *test cases* or *unit tests* in their courses. I was slated to teach CS 211 with Prof. Snyder for the first time and we were in the planning phases when he said that he wished there was a way make use test cases without resorting to a special tool. I didn't know much about the subject so I asked him to show me what he meant. He proceeded to fire up a large, clunky code editor and demonstrate. In the educational setting, test cases are essentially a series of small programs that the instructor can write to evaluate whether student code is produce correct answers or not.

I immediately saw the value in what Prof. Snyder was showing me. His concern was that students needed to use one particular code editor or a web site that Prof. Dobolyi utilizes to benefit from the test cases. This didn't smell right with me as unit tests were a very standard industry tool: they should have more than those two limited access methods. After a day of exploration, I had derived a variety of ways that students could use any test cases we provided for them. Prof. Snyder was excited by this development as it meant that we were not limited in the tools that we could suggest to students. Writing test cases became a central part of developing programming assignments henceforth. Mention of them can be seen in Figure 14

Providing students with tests cases allows them immediate feedback on whether their code is in alignment with the specification for their programs. If so, they also provide a measure of whether their programs are behaving correctly or not in a variety of situations. This immediate feedback is greatly appreciated by many of them as it does not require human instructors to be on hand at all times. In that respect, it is a fantastic effort multiplier, a great luxury that computer science programming classes enjoy over other areas.

## Problem 2 (4 points)

Files to submit

### **p2.c**

source code for problem 2. Includes a prototype for `lillie_magnitude` function but **not** the definition of it. Includes function `main`

### **lillie\_magnitude.c**

definition for the function `lillie_magnitude`. This is the same file as for Problem 1 - no modification should be necessary

### **p2-description.txt**

describes whether `switch/case` can be used

(Adapted from Hanly/Koffman 7th ed., pg 232 #5) Write a program which prints qualitative damage descriptions of earthquakes based on their magnitude on the Richter scale. The descriptions are given in the following table.

Magnitude	Description
$M < 5.0$	Little to no damage
$5.0 \leq M < 5.5$	Some damage
$5.5 \leq M < 6.5$	Serious damage: walls may crack or fall
$6.5 \leq M < 7.5$	Disaster: houses and buildings may collapse
$7.5 \leq M$	Catastrophe: most buildings destroyed

In `p2.c`, write a `main` function which takes an input amplitude and distance and uses the `lillie_magnitude` function to compute the earthquake magnitude (as was done in Problem 1). Print the magnitude. Then, in the `main` function, use a C conditional to print an appropriate description of the expected damage **on the same line**.

In the file `p2-description.txt` describe whether `switch/case` can be used easily for the table above. If not, what sort of conditional structure did you use.

Here is example output. Note as in Problem 1, we are compiling 2 C files at once, this time with `main` defined in `p2.c`.

```
lila [hw2]% gcc lillie_magnitude.c p2.c -lm
```

Figure 13: Sample assignment problem from CS 222 from early in my career.

### 5.3.3 Beyond Correctness

While test cases removed doubt from students as to whether their programs were correct and greatly reduced the tedium of graders checking for that correctness. They also introduced some problems. Both students and graders began to look at the tests as oracles: students would write sloppy code that just barely passed tests without striving towards aesthetic beauty which we discussed in class. Similarly, graders often just ran the tests and assigned scores based on how many passed and how many failed without bothering to comment on how students could improve the elegance of their code.

Prof. Snyder and I were frustrated by this in CS 211 and I was again in CS 310. A critical concept covered in CS 310 was program efficiency as specified by the “Big-O complexity” of the algorithms students derive. With the rationing of assignment credit to different functionalities of the program, it was difficult to emphasize efficiency to students via the assignment structure. It is very difficult to write test cases which can automatically evaluate efficiency leaving graders and

## Evaluation

Problems will be graded for their functionality and correctness. They should pass the test cases given and be able to pass additional test cases not presented as part of the project.

Code that does not compile will be heavily penalized (note all code will compile once you include sufficient definitions in your `Cat` and `FileMouse` classes).

We are providing a battery of *unit tests* in the file `P3Tests.java` which test the code. The actual grading process will definitely test more than these tests, so do not assume your project is done because you've passed these sample tests. Whether you manually test your code or add your own unit tests, it is ultimately your responsibility to ensure your program works.

To run the tests on the command line, make sure that the `junit-4.11.jar` is in the project directory (this is a library for running tests and is provided in the project download code). Then use the following commands on windows/macOS:

```
PCs:
demo$ javac -cp .;junit-4.11.jar *.java #compile everything
demo$ java -cp .;junit-4.11.jar P3Tests #run tests

Macs:
demo$ javac -cp .:junit-4.11.jar *.java #compile everything
demo$ java -cp .:junit-4.11.jar P3Tests #run tests
```

DrJava can run these tests by opening `P3Tests.java` and clicking on the *Test* button. (Clicking *Run* also runs the tests, but will also close abruptly; use *Test* instead). Click on test errors to jump to the test code. There, you can inspect the specific values that were used when a specific assertion (claim of truth) turned out to be false.

Figure 14: Mention of how Test Cases are used in the Evaluation of a CS 211 assignment.

students clouded on how to handle computational complexity issues.

The solution came to me over the summer of 2014 while I was in CS Associate chair Pearl Wang's office. I do not recall what we were discussing, but I was distracted by a stack of slender volumes on her desk titled "Introduction to Rubrics" by Stevens and Levi. I asked about them and Dr. Wang said she had been meaning to hand out a copy to all of the instructional faculty. I later paged through the book in my office. While I cannot say I was inspired by any of the specific rubrics in the book the over-arching idea was inspiring. All we needed was a check list of criteria which would be checked in each program. This could include anything that could not be easily verified by unit tests. In fact, it should *only* include things that need a set of human eyes to examine.

This was the impetus for the grading split that we now use in a variety of our intro programming classes: 50% for Correctness based on Automated Tests and 50% for meeting Manual Inspection Criteria. This is shown in some detail in Figure 15. This split provides students an explicit description of what their code needs to do beyond passing the provided tests. It also provides graders an explicit list of criteria to check for during grading. Below are some samples of manual inspection criteria I have used in assignments along with their total weight for the projects.

- **Partition Manual Inspection Criteria - 10% (CS 211 Project 1)**

Solution does not create extra arrays, but works in-place. Learning to work in-place with array structures is an important basic skill.



## 6 GRADING

Grading for this HW will be divided into two distinct parts

- Part of your grade will be based on passing automated test cases
- Part of your grade will be based on a manual inspection of your code and analysis documents by the teaching staff to determine quality and efficiency.

### 6.1 Automated Tests (50%)

- JUnit test cases will be provided to detect errors in your code.
- Tests will usually not be available on initial release of the HW but will be posted at a later time.
- Tests may be expanded as the HW deadline approaches.
- **It is your responsibility to get and use the freshest set of tests available.**
- Tests will be provided in source form so that you will know what tests are doing and where you are failing.
- It is up to you to run the tests to determine whether you are passing or not. If your code fails to compile against the tests, little credit will be garnered for this section
- Most of the credit will be divide evenly among the tests; e.g. 50% / 25 tests = 2% per test. However, the teaching staff reserves the right to adjust the weight of test cases after the fact if deemed necessary.

### 6.2 Manual Inspection (50%)

- Teaching staff (GTAs and professors) will manually inspect your code and analysis documents looking for a specific set of features.
- Most of the time the requirements for credit will be posted along with the assignment though these may be revised as the HW deadline approaches.
- Credit will be awarded for good coding style which includes
  - Good indentation and curly brace placement
  - Comments describing private internal fields
  - Comments describing a complex section of code and invariants which must be maintained for classes
  - Use of internal private methods to decompose the problem beyond what is required in the spec
- Some credit will be awarded for clearly adhering to the target complexity bounds specified in certain methods. If the specification lists the target complexity of a method as  $O(N)$  but your implementation is actually  $O(N \log N)$ , credit will be deducted. If the implementation complexity is too difficult to determine due to poor coding style, credit will likely be deducted.

Figure 15: Description of grading criteria for a recent programming assignment.

- **SparseBoard Design Clarity - 10% (CS 310 Project 2)**

- The design of the SparseBoard class is documented and apparent.
- It is clear what information is tracked in which fields and how they are manipulated during various methods.
- Internal classes are used to store tile coordinates.
- Private methods are employed to maintain the correctness of the tile position lists after shifts.

- **Complexity Targets - 10% (CS 310 Project 3)**

- It is clear that the `bestScore()` is returned in  $O(1)$  time
- It is clear that `stateReachable()` returns in  $O(1)$  time
- It is clear that `movesToReach(game)` and `bestMoves()` do not re-explore significant portions of the state space but instead can directly compute a path by either returning immediately or performing no more than  $O(P)$  steps where  $P$  is the length of the path to be returned.



Weight	Social Impact Grading Criteria
5%	The broad impacts of the technology on human life and behavior are discussed
5%	Specific groups that have greatly benefitted from the technology are identified and discussed
5%	Specific groups that have been adversely affected are identified and discussed
5%	The current societal impression of the technology is discussed with sources described for the societal impression
5%	The author's personal experience and perception with the technology is described

Figure 16: *Excerpt from CS 100 Term Paper Rubric*

Rubrics have found their way into several other areas where I teach. We have used rubrics to facilitate grading in CS 105 since I started teaching the course but I have made some recent upgrades based on grader feedback to assign specific credit for facets such as writing quality and understand-ability and provide additional guidance on which elements should be present in student answers. I also employed a rubric to guide CS 100 students on their final research papers. This made it relatively easy to answer questions about what should be present in the paper and to eventually grade it.

#### 5.3.4 Not just What but How

I mentioned this earlier but it bares repeating: if you don't know how to do something by hand, you will not be able to program a computer to do it. This surprise many new programmers but it eventually sinks in that computers are dumb as bricks. If one wants computers to do better things, one needs to ensure that the human programmers know what they are doing.

To that end, I am disappointed that many of my early programming project specifications did not convey much information on how various problems could be solved. This often left students a bit bewildered. I found myself repeating things a lot in office hours to different students, wishing I hadn't erased the picture I had drawn from the previous inquiry. This has proven a reliable indicator that a structural improvement to an assignment can be made.

More recently, for most appreciably sized programming tasks, I now try to include a wealth of detail on how the parts of a programming system will fit together, what the explicit steps should be taken to solve a problem, or hint at situations which might arise and how to deal with them. These "implementation notes" often cause my program specifications to sprawl: the longest in CS 310 during Fall 2017 would have been 30 pages if printed. Thankfully for the trees of the world most students navigate the large specifications via the web page they are posted as. While the additional girth is undesirable, it is important to adhere to Einstein's advisement on complexity: *Make things as simple as possible but no simpler*. Things that require explanation should get it. A sample of implementation notes from a project are in Figure 17. In some cases I will include diagrams to illustrate the layout of important aspects of a project such as a data structure. In other cases I include demonstration sessions of how a program should function from a user perspective so that students know what behavior they are expected to code for.

Detailed explanations were critical in CS 100 where students were expected to be gaining computing literacy through the course. In this setting, I frequently wrote assignments as an alternating series of tutorial sections and problems. The tutorials often re-iterated what we had done in lectures during practice sessions adhering to the old adage "Repetition is important in education." A small demonstration of this pattern is shown in Figure 18 though the full effect requires a view of the full assignment.

### 3 AStack CLASS

#### 3.1 Overview

Stacks are a simple linear data structure in which elements are added to a figurative "top" via `push(x)` calls. Only the top element can be retrieved, either via calls to `top()` to receive a copy of the top value or via `pop()` to remove and retrieve the top value. Stacks are ideal for tracking potential paths through a maze in which a new direction like `North` is pushed onto the top of the stack but if it is found to do no good, is subsequently popped from the stack to be replaced by another direction.

You will implement `AStack<T>`, a standard approach to stacks which uses standard java arrays to store elements. It is a generic class so can store any type of element.

#### 3.2 Implementation Notes

- **CONSTRAINT:** You may not use any external classes while implementing `AStack` except in methods indicated (`toString()` and `toList()`).
- There are two notions of size in the `AStack`:
  - `size()`: How many elements are actually present in the stack (pushed but not popped)
  - `getCapacity()`: The number of elements that can be stored in the `AStack` without triggering an expansion of the internal array
- The internal array used by the `AStack` presents some issues with respect to types and generics. Java does not allow for the creation of generic arrays like

```
T stuff[] = new T[10];
```

There are two ways to surmount this for classes like `AStack`.

The first is outlined in Building Java Programs 15.4 and involves casting `Object` arrays to generic arrays. This will generate warnings which can be suppressed by placing the annotation `@SuppressWarnings("unchecked")` above offending methods.

The other solution is to make the internal array an `Object` array and perform casts on objects that need to be returned from `top()` and `pop()`.

Figure 17: Sample of implementation notes for a programming project.

#### 5.3.5 Organization

An explicit goal of CS 211 and CS 310, indeed the entire purpose of the object-oriented programming paradigm, is to handle larger programming problems by decomposing them into human-comprehensible chunks. Teaching students how to cope with larger problems is a great challenge. This happens over the entire three-course programming sequence, but as the problems become larger in CS 310, I have found it beneficial to students to include certain information in project specifications.

First, since students are working on larger problems for the first time, I make every effort to describe why I decomposed the problem into the program parts they are required to implement. It is extremely important to me that students recognize that eventually they will need to be the ones decomposing. I want them to be exposed to my best efforts to realistically encapsulate parts of a system so that they may serve as good examples in the future. Usually the many engineering choices involved in a design have consequences that merit discussion and only through those discussions will students begin to understand the trade-offs involved.

Simply keeping the many moving parts associated with a large project in one's head can be difficult. I attempt to provide several road-maps for large projects to aid student work. The following items have been requested frequently enough to be made part of every programming specification.

### 2 BASIC ENCRYPTION

In class we studied two very basic symmetric encryption techniques, the [Caesar Cipher](#) and the [Vigenere Cipher](#). We covered these during Week 7-2 and 8-1. The next two problems will have you do a little encoding and code breaking.

Problems 3-5 use the following table of number to letter correspondences. Notice the underscore ( `_` ) is the last character so there are 27 letters in this alphabet numbered 0 to 26. Underscore will be used in place of spaces in messages

#	0	1	2	3	4	5	6	7	8	9	10	11	12	
Char	A	B	C	D	E	F	G	H	I	J	K	L	M	
#	13	14	15	16	17	18	19	20	21	22	23	24	25	26
Char	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	_

For example the word HOMER\_BOWLS would be converted to numbers as

7 14 12 4 17 26 1 14 22 11 18

### 3 PROBLEM 1: BASIC CAESAR CIPHER

Use the Caesar cipher to encrypt the following message using the given key.

- Message: EAT\_MY\_SHORTS
- Key: 14

Decrypt the following message using the given key.

- Message: XLBRP
- Key: 11

**What to put in your HW Write-up**

- Put your encrypted and decrypted messages in your write-up
- Make sure to show some work so you can garner some partial credit in case things go wrong and your final answer is a little off

Figure 18: Sample tutorial section followed by an associated problem in a CS 100 assignment.

- A list of the required files along with short descriptions of each. Often large files involve some files provided to students without need for changes, some that require alteration, and some that must be created by the student. A table of such information helps keep straight what needs doing.
- A “how to get started” section which advises students on the first steps to take to get off the ground. This is particularly useful for the third and fourth projects in CS 211 which tend to ramp up complexity over previous efforts where first steps are more obvious.
- A table of contents for the project specification. Seeing the structure of the specification document itself can help guide student work and their inquiries about where relevant information might be found. During Fall 2016, Prof. Carver used a Google doc with a floating table of contents on one assignment. The TOC showed regardless of what part of the document was being viewed. I liked this feature very much so I used a little HTML wizardry to incorporate it into my project specifications as shown in Figure 19.

### 5.3.6 Milestones and Deadlines

A portion of students in CS 310 always leveled the complaint that the projects were so large that they never seemed to start early enough to finish. They wanted for more smaller projects. A learning outcome for CS 310 is to complete larger projects but I understood their sentiment. Even in industry large projects have Milestones that should be reached partway through to facilitate timely project. That line of thought caused me to institute a new feature during Summer and Fall 2016 in CS 310: Milestone Deadlines. In it, a small portion of student grades would be devoted to implementing part of the project and passing tests about a 1 week before the final deadline. This simple effort broke otherwise large projects into smaller hunks that students felt was more manageable.

### 5.3.7 Educating the Team

Since adopting the split grading scheme I described earlier, I have always had grading meetings with graders to discuss how the manually inspected criteria should be assessed. At grading meetings, I and any other instructors on board walk teaching assistants through how we would go about grading several sample assignments, talking out loud about what we are looking for

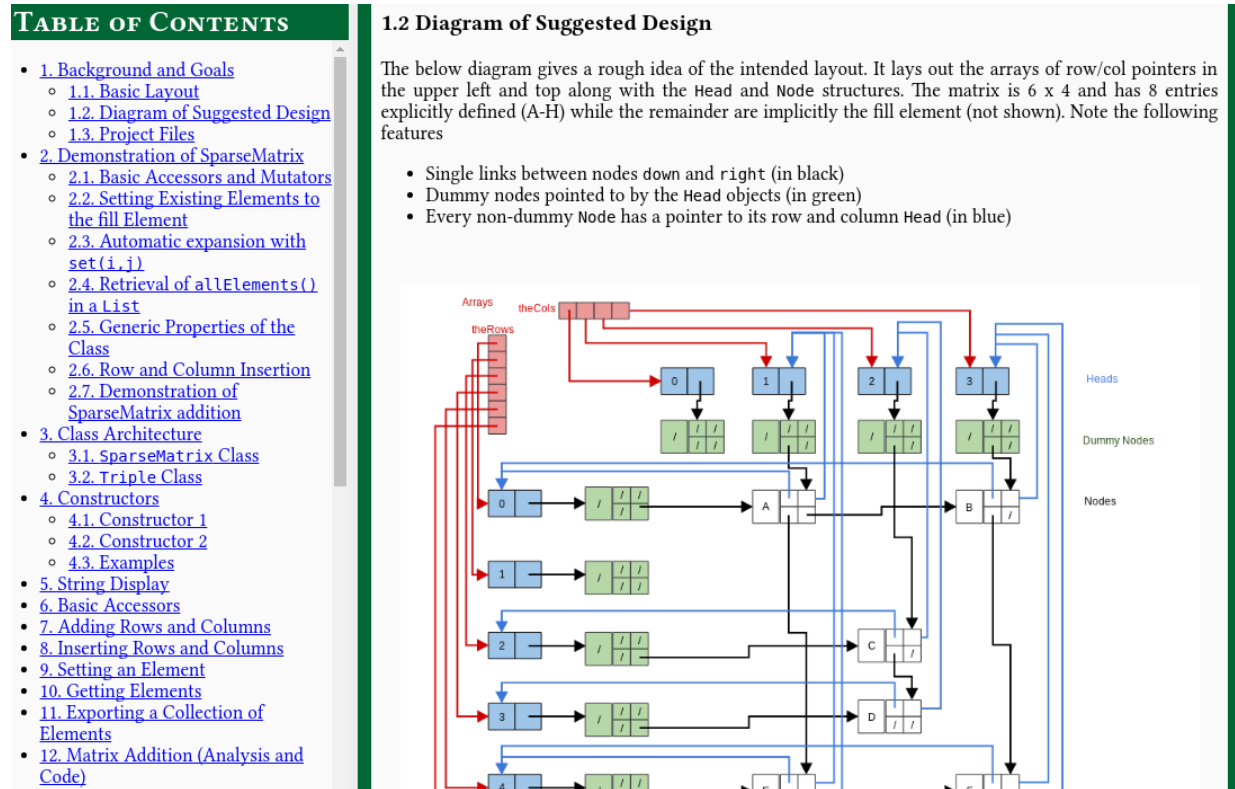


Figure 19: Sample from a CS 310 assignment showing both the floating table of contents and a diagram illustrating the data structure to be implemented.

and reasoning about how to award credit. This greatly improved the confidence of most teaching assistants that they would be grading according to the standard I desired. It is now my standard practice when working with any teaching assistants. I also tend to invite both graduate and undergraduate teaching assistants to a “project overview” meeting at which we discuss the basics of the project and I point out issues that the should make students aware of during office hours. Several TAs commented that this is greatly speeds their acclimation to the project so that they are more quickly able to render help to students.

### 5.3.8 Feedback on Assignments

I have developed a bit of a reputation in the department for the programming assignments I use. Not everyone views them as a universal win: while feedback indicates I do a good job of supporting students the process, it is clear that the difficulty level is high (see Figure 20 for some sample end-of course summaries). This is something I can live with, to be known as the prof that doesn’t pull punches. That sentiment is solidified by the feedback from several students. Particularly the comments from the last two students who have since graduated and taken on jobs encourages me to continue to challenge students.

He assigned us a project on how we might use maps and directed acyclic graphs to implement the back-end of an actual spreadsheet application. By assigning us these projects, professor Kauffman gave us a valuable glimpse into how we can apply the material learned in class to real problems outside of the classroom. *–Jimmy Bodden*

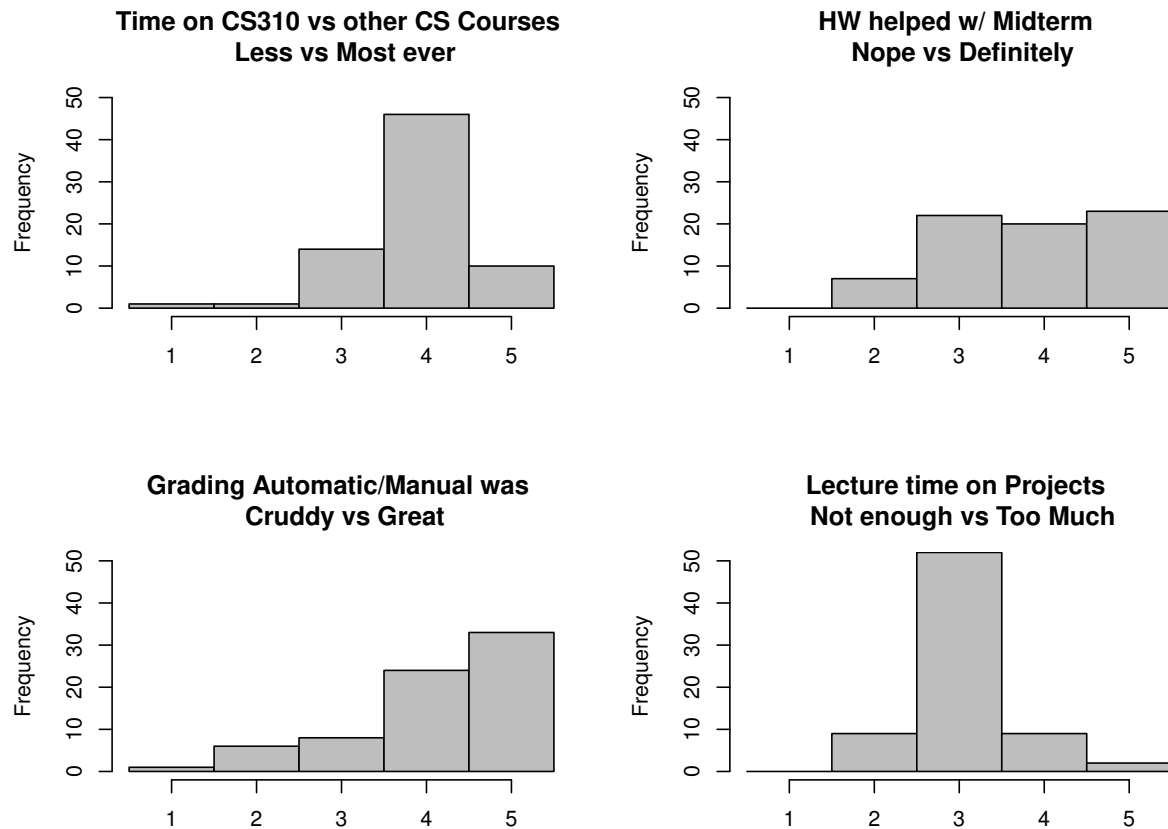


Figure 20: Sample feedback from CS 310 concerning assignments and support provided in class for them.

The assigned projects were quite difficult, but incredibly well organized. Professor Kauffman used portions of class time to assist struggling students, such as myself, with the trickier aspects of the projects. *–Will Crosswait*

I recall on multiple occasions, as a student and a UTA, admiring the clear layout of materials and assignments by Chris. His course websites, slides, and project directions are extremely straightforward, easy to find and follow. He is one of the few professors I know who have done a great job of striking the balance of providing details for students without over or under loading materials and course pages. *–Bridget Lane*

Dr. Kauffman's projects for CS 211 were also very challenging and provided me with the necessary skills and knowledge to be successful in my second internship, in which I had to write Java code for android development. He provided us with the guidelines and tips necessary to complete the projects during class. *–Emmanuel Tagoe*

Although the class (CS 211-Honors) was time consuming with weekly projects, I most definitely saw how each project tied in with that weeks material and my struggle to perfect each project showed me how my way of thinking was flawed and aided in the mastering of the class material as shown in my test scores. I put in a very high degree of effort into the class and it was ultimately rewarded with a newly marketable skillset.

–Forrest Bussler

Kauffman made notoriously challenging projects [in CS 211]. At the time these projects were a real pain, but they taught me the crucial soft’ skills of breaking down large problems, managing your time, and most importantly knowing how to ask questions. Today, I would argue, these teachings are what allow me to succeed in the industry.

–Mohammad Ahmad

## 5.4 Office Hours

As most professors do, I hold office hours regularly to make myself available to students. I view it as one of the most critical times to interact with students. Students attending office hours are taking time away from other endeavors to broach some subject with their instructors. This could be on account of curiosity or desperation. Both situations require care and encouragement from the instructor. Extra curious students seek guidance on how to go beyond what they see in the standard textbook material. Proper guidance can kindle the flame of curiosity which fuels scholarly inquiry and success. Likewise, desperate students need guidance as well, with their immediate, seemingly insurmountable problems, and also on how to avoid getting into such sticky situations again, an important facet to growth.

The first thing I noticed about holding office hours at GMU was that I needed to adjust my space for it. The standard layout of offices in the engineering building is great for working on one’s own or having a conversation with a colleague across one’s desk. Unfortunately, it is woefully deficient for examining a student’s code on their laptop or even showing them code of my own. To boot, students often had to sit the hallway as despite having a reasonable amount of space, it was arranged such that only a couple folks aside from me could be in the room at once. Figure 21 shows this original sub-optimal configuration.

After a couple years of banging my knees while navigating around my desk, I decided a re-arrangement was in order. I visited home depot and purchased a few wooden table legs. I detached the desk which bisects most offices, attached an extra leg, and with a little maneuvering of the bookshelf created a student work table.

This open layout was much preferable as I could actually have students sit next to me while we examined their code and easily show them code on my desktop screen to compare. The open layout also meant I could fit a few more chairs in the room and access the whiteboard much easier. Rather than make students wait outside, I started packing in as many as I could. This enabled several wonderful things to happen.

- Students with the same question would overhear my answer and reduce the amount of repetition necessary. These days, group conversations about programming projects or lecture material starts spontaneously with great frequency during office hours as waiting students are naturally drawn in.
- Students help each other while waiting. While rules around collaboration on assignments are designed to ensure accountability of answers, students are encouraged to help each other to understand the goals of the project and to understand errors that occur in their code. While in my office, they have fairly free reign to chat as they know I will curtail conversation that approaches too much help.
- I can operate much more efficiently to answer more questions. A typical code debugging cycle requires one to understand what is going wrong in a program, hypothesize about the





Figure 21: Office re-arrangements to facilitate office hours. Top Left: original layout dividing the room making it hard to see student screens. Top Right: Ad Hoc table leg to open create a second desk for students. Bottom: Current Open layout to facilitate group conversations.

cause, introduce some new code to verify the hypothesis, and repeat based on the results. Teaching students this cycle is one of the most critical aspects of introductory programming. I typically help a student with the first two steps in office hours but then let them code the program changes themselves and analyze the results. While one student codes, I can move to aid another student and then return later to help assess the original student's results. This asynchronous flow makes me feel a little like a chess player engaged in several games simultaneously, but students appreciate the fact that I almost get to everyone who comes to office hours through this "time slicing" approach.

The new open layout was great except for my ad hoc table: it was rather unsteady and students tended to knock it askew by bumping it too hard. Eventually I negotiated for a proper second desk in with my department head to improve that facet and the final layout is shown in panorama in Figure 21. I have held many meetings with up to seven people comfortably with the new layout. Students sometimes ask why I got a bigger office than my neighbors to which I usually shrug and reply, "Let's get to work."

Several former students commented on my office hours when I asked for feedback for this portfolio. The most frequent discussion point seems to be that they appreciated the availability of help outside of class.

He always made the time to help us complete and understand the Java projects, and also took a lot of time to guide us during his office hours. *—Emmanuel Tagoe*

He committed his time to aid me with a particular coding assignment in which I was experiencing trouble, and helped to guide me to not only correct my errors but explain the process and solution. *—Selena Chaivaranon*

Mr. Kauffman also went above and beyond with assisting students outside of the classroom during his office hours which mostly consisted of people waiting until the last moment to do the project. These students went against Mr. Kauffman's advice of not waiting until the last moment to do the project, yet he still spent countless hours aiding these students because they wanted help. *—Forrest Bussler*

He took more time out for office hours than any professor I have ever seen. The line outside of his door on the days where he provided office hours was wrapped all the way around the hall in his building. Dr. Kauffman was there every week, willing to help anyone and everyone who needed it. *—Taylor Kohler*

## 5.5 New course development

I have developed two new courses for the computer science department, CS 100: "Principles of Computing" and the special topics CS 499: "Parallel Computing". CS 100 has been offered several times so I will describe in the sections below some of the planning and growth that has gone into the course.

### 5.5.1 A Computing Course for Everyone

There is a large national push to increase the amount of computing education available to students at all levels. Particularly, bodies that govern the computing profession including the ACM, IEEE,



and NSF are interested in increasing the generally literacy of Americans concerning computing. This will be crucial to harnessing the powers of computation and automation to benefit our society.

A variety of pilot programs teaching general computing concepts to students have been demonstrated around the country at both the late high school and introductory college level. Pearl Wang, who chaired our CS Undergraduate Studies Committee, became aware of this development and felt it would be a good move for our department to offer such a course. During the Fall of 2013 she recruited me to help develop the course to be offered for the first time in Fall 2014.

I worked with CS faculty Dana Richards during the spring and summer of 2014 on the course. Prof. Richards has a deep knowledge of computing history and the texts targeted at lay people concerning computers. He and I were in strong agreement that the course should not be a standard information technology skills course. Instead, we wanted to explore the nature of computers and its ramifications for human society and culture.

While attending SIGCSE, a large computing education conference, in Spring 2014, I took the opportunity to attend several sessions on “CS o” courses which are intended for non-computer science majors with little background on computing. A wealth of resources and ideas were described in the courses which would eventually make their way into my offering of CS 100.

As Summer 2014 drew to a close, I began finalizing my plans and writing assignments. While I had experience with non-programmers from my early teaching, that course was designed to give students narrow exposure to Java programming. CS 100 promised to be its own can of worms, exciting and, as worms often are, a bit muddy.

### 5.5.2 General Approach to CS 100

My general approach to CS 100 was to hit on what I felt were the most important overarching concepts in computing in an active and pragmatic fashion. For that, I am exceedingly grateful to Prof. Richards for introducing me to the short textbook “The Pattern on the Stone” by W. Daniel Hillis. This marvelous little volume serves as the backbone of the course. It unifies diverse concepts in computing such as bit representation of images, basic programming, algorithm efficiency, and computer intelligence. While not sufficient on its own to run the course, it provides just the right structure around which to build a curriculum.

My first lecture centers around a devious question: what is the square root of 18? All students can pull out a calculator or web site to crank out the numeric answer, but if such tools are forbidden, the problem becomes interesting. The solution is the first formal algorithm to which students may become acquainted, the Babylonian Algorithm shown in Figure 22 along with a demonstration. All students know how to add, subtract, multiply, and (when sufficiently coerced) do long division. With these basic operations repeated in the specific pattern prescribed by the Babylonian algorithm, students can calculate accurate square roots as well, just as ancient humans could do.

As an introductory example it is a microcosm of information about computing.

- Algorithms specify what a computer can do and are precise recipes to transform some input problem into desirable output.
- People can execute algorithms just as machines can, slower in some cases but much faster in others.
- Every computer, be it human or metal, must have some primitive operations which it can already execute and some logic which allows those operations to be stitched together with repetition and conditional execution.

### Babylonian Algorithm

- $S$  is an input number, the algorithm calculates  $\sqrt{S}$  (square root of  $S$ )
- Initialize: Set  $x$  to a guess
- Repeat
  - Calculate  $x_{next} = \frac{1}{2} \left( x + \frac{S}{x} \right)$
  - Set  $x$  to be  $x_{next}$
  - If  $x^2$  is close enough to  $S$ , quit

```
s = 18    # Find my square root
x = 4     # A guess
# Repeat these steps
xnext = (1/2) * (x + 18 / x)
x = xnext
x
4.25000000000000000000000000000000
x^2 - 18
.06250000000000000000000000000000
# Pretty close, but can we get closer?
xnext = (1/2) * (x + 18 / x)
x = xnext
4.24264705882352941176
x = xnext
x^2 - 18
.00005406574394463663
# That's close enough for me
```

Figure 22: *The Babylonian Algorithm to compute square roots and a demonstration to find  $\sqrt{18}$ . This is the introductory algorithm in CS 100.*

- Executing algorithms is usually not difficult but establishing algorithms that work in the first place requires great creativity and ingenuity.

### 5.5.3 First Steps Programming

On day one I make it a point to advise students that CS 100 is not just about number crunching by demonstrating several other algorithms all of which are well above and beyond square roots: recognizing text and faces, distinguishing between cats and dogs in images, and standing on one foot are all extremely significant computations at which humans still far exceeds machines. Some of them concern assignments they work on in the weeks to come. We also allude to assignments students will work on which concern more visual computing.

A fantastic resource which SIGCSE exposed me is the Code.org website which features a wide array of introductory programming activities. Their Advanced Course has served as the basis for the second CS 100 HW assignment since it began and features an accessible drag-and-drop programming style with visual puzzles. A sample activity is shown in Figure 23. This style of programming serves as an excellent bridge for the uninitiated programmer as it gently introduces many major programming such as variables, conditional execution, repetition, and abstraction of code into individual procedures.

While Code.org is fine for as a first pass at coding, it is not a “real” programming tool and a major goal was to expose CS 100 students to such a tool. For this, I selected the Python programming language for its simple syntax and wide use in industry and engineering. To narrow the gap, our first programming exercises center around drawing, just as was done in Code.org. Python has a built-in library for drawing in the style of the old Logo teaching language. Students guide a “turtle” around the screen to draw various shapes and colors. This gives immediate visual feedback about the state of the program which quickly teaches students that the turtle will do exactly what they tell it but no more, both infuriating features which instruct on the precision required to “teach” a computer to do anything useful. Figure 24 shows a couple such introductory exercises. While the main textbook for CS 100 does not have much information on Python programming, I supplement it with lots of in-class activities and readings from the free online programming text “How to Think Like a Computer Scientist: Learning with Python 3” by

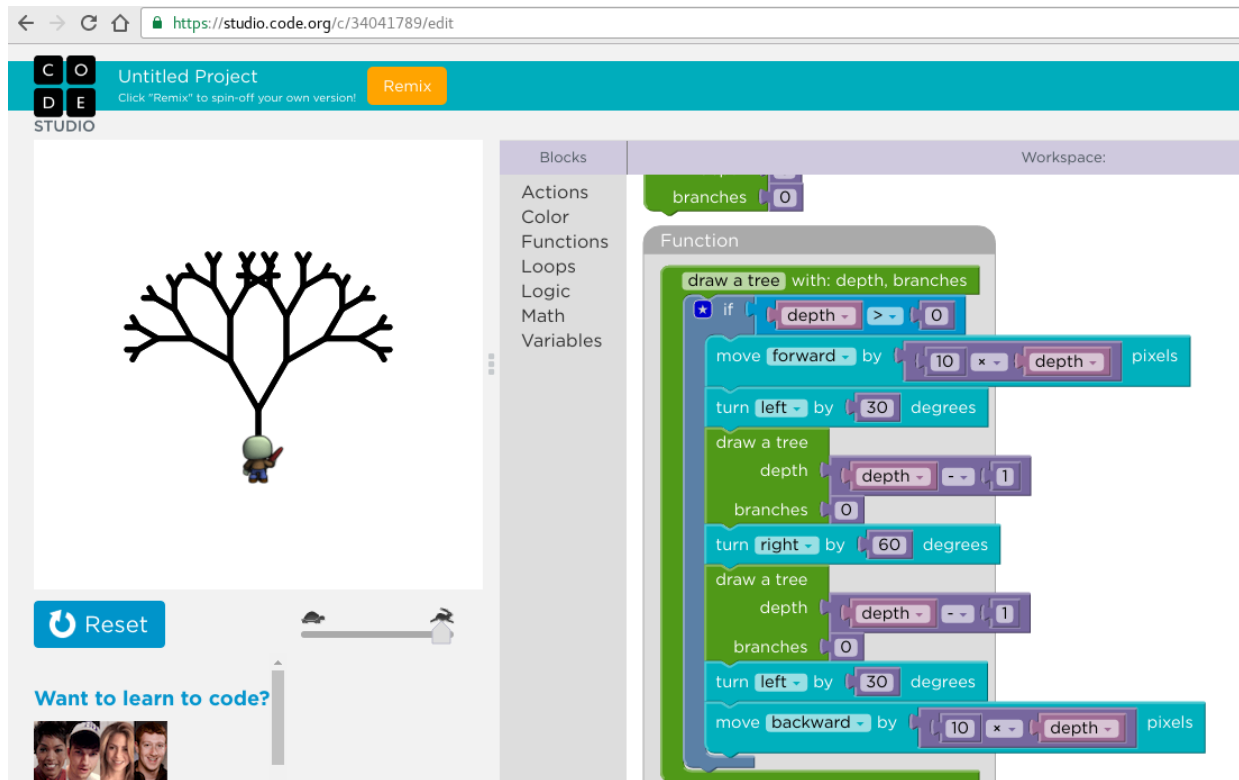


Figure 23: Sample exercise from Code.org which is used in a CS 100 HW. Students must trace how code causes the binary tree shown to be drawn.

Wentworth et Al.

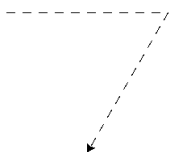
#### 5.5.4 Beyond the Code on the Page

A large segment of CS 100 is designed to give students basic familiarity with how computing and programming works. However, this is intended to set up deeper discussions of what can and should be done with computing. We focus throughout on social issues concerning computing. A variety of these subjects draw from my experience teaching CS 105: “Computer Ethics and Society” but with the additional time in CS 100, we can devote more discussion to social issues. Topics of conversation include the following.

- **Networks and Encryption:** Students will make many, many transactions over networks so a basic understanding of the Internet architecture is important along with how communications between two distant parties can be secured through encryption. This assists students in staying safe as they traverse the tubes. Activities to support learning include writing simple encryption programs, simulating packet transfer by passing notes in class, creating and posting publicly accessible HTML pages, and watching John Oliver’s insightful commentary on Net Neutrality.
- **Data Privacy:** Tied up with discussions of encryption are always data privacy issues. Who can find out what sites you have visited? What level of evidence is necessary for companies to give digital information to law enforcement? What if government can’t break future

## Pen goes up, Pen goes down

- ▶ `penup()` stops drawing lines, allows turtle to move without drawing
- ▶ `pendown()` starts drawing lines again
- ▶ Useful for dashes and for `face.py`



```
for i in range(10):
    forward(10)
    penup()
    forward(10)
    pendown()

right(120)
for i in range(10):
    forward(10)
    penup()
    forward(10)
    pendown()
```

## Loop Variables Change Each iteration

The `range(N)` statement produces a sequence of numbers from 0 to N; good for loops

```
# prints 0, 1, 2, 3
for i in range(4):
    print(i)
```

```
# Square spiral
size = 0
for i in range(15):
    size = (i+1) * 25
    forward(size)
    right(90)
```

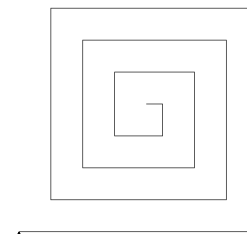


Figure 24: Sample in-class exercises from CS 100 which introduce Python programming via drawing.

encryption? Students discuss all of these issues in CS 100 so that they can make more informed personal and policy choices going forward. Classroom activities include viewing a video on the myriad of digital information that is regularly collected about students and viewing part of an episode of Frontline on the topic of government data mining to prevent terrorism.

- **Machine Intelligence and Automation:** Computers have eliminated a tremendous number of jobs from human number crunches to phone operators to accountants. As our understanding of how to train machines for complex tasks increases, it is likely automation will supplant yet more jobs. Students in CS 100 learn the basics of artificial intelligence and discuss both its beneficial and unsettling impacts. In class activities include discussion of how SPAM filters work and research on likely jobs to be automated in the near future.

It is extremely important for me that students devote critical thought to these issues. Computers are a mirror which reflects back the values of users. Students with a basic understanding of the capability of machines that acknowledge they will do what we tell them to do can collectively guide us towards the most beneficial use of computing and automation.

### 5.5.5 Feedback on CS 100

Generally feedback on CS 100 has been very positive for my sections. Students appreciate the pace, the challenge, and the variety of topics we cover.

That is not to say everything has gone perfectly. For example, during Fall 2015 I introduced another textbook, an paid online text from Zyante called “Computing for All.” I had experimented with Zyante offerings a few times in other classes with mixed results and wanted to supplement the main text with some more activities. I required students to engage do the online activities at regular intervals for credit in the class. My initial impression was that the exercises were fairly boring but though they might introduce students to unfamiliar concepts. Unfortunately, most students were already well acquainted with the text topics and only bought the book to do exercises for credit (see top left panel of Figure 25). This has caused me to drop the Zyante book in my most recent offering of the course as there is no need for students to spend money on something that is providing no benefit. If it appears there is still a need to supplement our main text, I will explore further options.

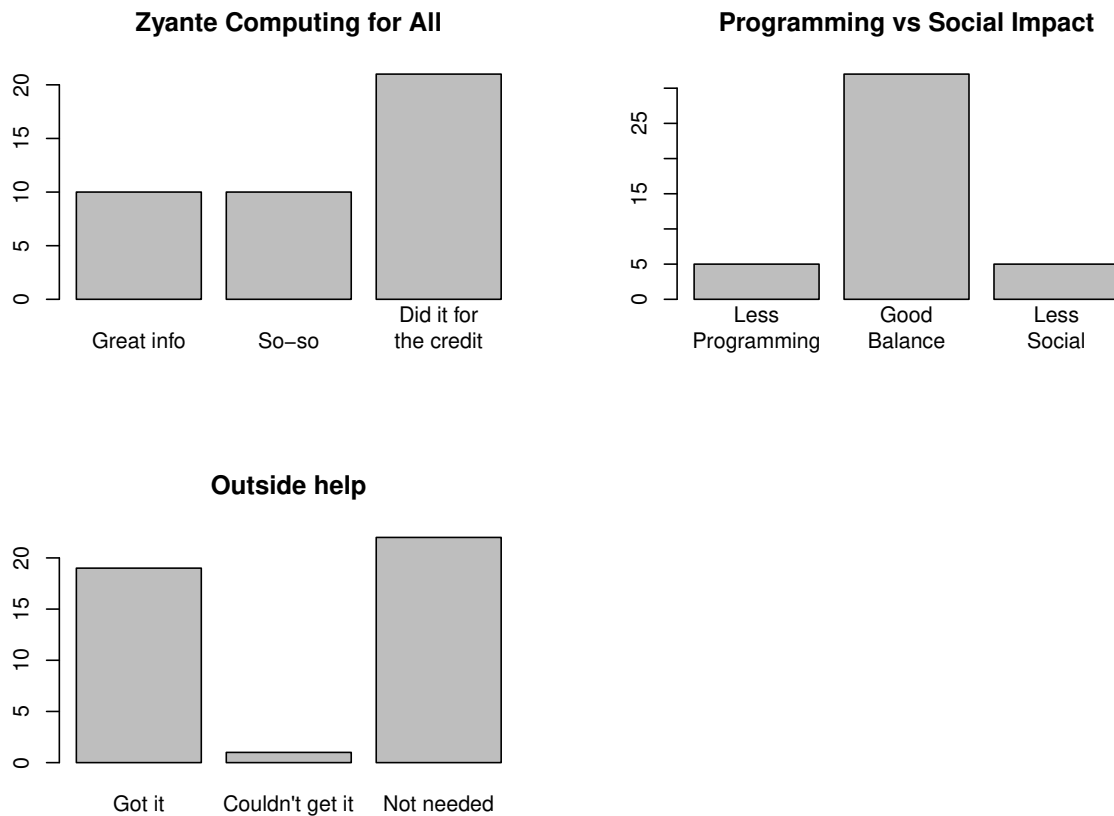


Figure 25: Some feedback from CS 100 from Fall 2015. This led me to drop the Zyante text as it did not provide much benefit.

Two students who took CS 100 with me provided some evidence that the course is going well to support the overall favorable course feedback ratings. Here are excerpts from their descriptions.

By nature, CS 100 is fast-paced and oftentimes challenging, and I feel that it is thus imperative to emphasize Professor Kauffmans willingness and ability to reach out to his students in an effort to ensure they are given the best learning experience available to them in an introductory level course. He readily considers and addresses multiple approaches to students questions, takes student feedback very seriously, and diligently collects and makes available past data. He is tirelessly and honestly committed to engaging students in a fun, interactive, and stimulating course which challenges students to not only try their own hand at basic coding, but consider how changes in technology in the digital age affect their education, and even their day-to-day lives.  
—Selena Chaivaranon

He did an awesome job of bringing in real world examples to the classroom in order to engage students more. For example, when we were going through the information security portion of the class - which to me, sounded really boring - Dr. Kauffman brought it to life. He brought in a couple of videos that broke down the concepts, and

## Open Resource Exam Rules

## Can Use, physical or electronic

- ▶ Notes
- ▶ Textbook(s)
- ▶ Editor, Compiler, IDE
- ▶ Oracle Javadocs
- ▶ Language Dictionary
- ▶ Locally stored webpages
- ▶ Your own code
- ▶ Textbook/Instructor code

## Cannot Use

- ▶ Internet Search
- ▶ Piazza discussion board
- ▶ Downloaded applets
- ▶ Chat
- ▶ Texting
- ▶ Email
- ▶ Communication with anyone except Instructor/Proctor

- ▶ Protect your work from theft
- ▶ You may be asked to show your GMU ID
- ▶ If you aren't sure of something, ask

Figure 26: Open Resource Exam Rules used for courses. Almost every exam and quiz I give uses some variant of these rules. The slide is displayed during the exam to remind students what resources are in bounds.

then would go into detail about how these intricate subjects impacted our everyday lives. He tied it into how it impacts email, online shopping, texting, and then went into a history lesson about how encryption was used back during war times to get important messages from one battle site to another. Now, instead of the topic being a bland subject matter, everyone was engaged, because the subject matter now impacted us all - we wanted to know more about how encryption kept our online credit card and shopping information safe. Plus history lessons are always fun! –Taylor Kohler

## 5.6 Potpourri

This section enumerates a few more critical elements of my teaching which don't fit well elsewhere. Relevance to my educational philosophy and some growth over the years are touched on in each case.

### 5.6.1 Assessments

I take a somewhat unique approach to assessments in all my courses. In almost all cases, I give open-resource assessments. That is, students can use a wide array of materials they have accumulated during a course during quizzes and exams. Figure 26 shows typical rules associated with open resource exams.

Allowing students to use their resources introduces some amount of risk to assessments. It is possible to attempt to google for exam answers or otherwise communicate with outside sources. In my four years teaching, I have detected this happening twice. When compared to the large number of students that I have administered exams to, I consider this a favorable trade-off due to the positive aspects of open resource exams I enumerate below.

When students are allowed to utilize notes, compiler, lecture slides, program documentation, and other tools, exams can become much more interesting and much better at assessing their true skills. I teach programming often and it always seemed a perfect sort of ridiculousness that one would in all cases use tools like a text editor and program interpreter aid the development of

code, all cases except exams where the penalty for failure was highest. When students can use their tools, their HW becomes training for the timed event which is the exam. If need be, they can write out a program quickly in their editor, compile, debug, and transfer. The cost to change one's mind in on a computer editor, to re-arrange pieces is minute compared to a mistake written on paper which must be erased and re-stated. The tools allow students to sketch out ideas first then commit when they have gained confidence.

There is a tremendous amount of minutia involved in computing: specific syntax, special symbols, particular names for library functions. Even with as much experience as I have, I am constantly referencing documentation as I remember there is some piece of code that does what I want but I cannot recall the name for it. Allow students to use their notes, textbook, and standard documentation frees them from worry about such details. They are important details, but I would much rather students focus on how to stitch existing pieces together to solve a problem than exactly what those pieces are called. Programming is an active art and having tools to facilitate that activity is what makes a programmer powerful.

All of this stems from my basic philosophy of assessment: figure out what you want students to be able to do, create assignments that exercise those skills, and examine them in on similar material. Few coders but the elite can get work done without tools. Indeed, few people in any job can get their job done without quick access to relevant resources. That means assessments today should focus on how well students have learned to use resources rather than how much low-level detail has been crammed into their heads.

My exams can be somewhat tedious to grade as they often involve my grading whole programs. With enough practice, I have learned to do this with relative speed though it seems every final exam grading season I find myself wondering whether grading a given problem will be enough to push me into true/false exams. So far, so good.

One other adaptation I have used is to offer "Mini-exams" in some courses, notably CS 100 and CS 499, samples of which appear on pages 67 and 69. These are an alternative to single, monolithic midterm and final exams and short frequent quizzes. Mini-exams are a 30-minute assessment with usually 4 per semester. This allows them to be administered relatively close to the material which they cover. Lower-stakes, frequent assessments are advocated for by several educational sources including "How Learning Works," a text I have worked out with my SIMPLE teaching development groups.

### 5.6.2 Gathering Feedback

I gather a regular feedback from students midway through a course to facilitate changes and at the end of the course. I have always been interested in asking course-specific questions so have often rolled my own surveys to complement the generic university evaluations. These "Customer Satisfaction Surveys" often have a humorous bent in an attempt to engage students and elicit more responses. A sample appears on page 71 with both the form and the results analyzed.

Typically I review midterm feedback with students in lecture to discuss corrections that will be made to address any issues that have arisen. I also provide the results online for both midterm and final feedback as I want students to know that I am listening and care about how the course is running for both their section and future sections.

Feedback from students has led to a number of policy and practice changes in courses I run.

- I try to include answers to most lecture exercises in slides or in code packs associated with those lectures now so that students have a reference of how to solve in-class activities.
- We experimented with using an online grading system for labs in CS 211 but dropped it due to student feedback reporting that it was too difficult to utilize.

- In CS 211 labs, we were able to advise some lab leaders that they needed to prepare more discussion material and prompt students to participate more. This is crucial training for future academics.
- In CS 100 I dropped a required online textbook that was rated very badly by students as high cost with little decent content. My initial fear about the text was confirmed by their feedback: it was mainly fluff they already knew.
- Based on feedback from CS 310 students, we now have a “Milestone Deadline” which breaks the large projects in those classes into smaller chunks to make them more manageable without sacrificing the overall size of the projects.
- Also based on feedback from CS 310 students I have become particularly sensitive to scheduling assignment deadlines around exams. An early gaffe of setting a due date on an exam date resulted in un-ambiguous declarations that I was trying to kill the entire class. Feedback noted and schedule permanently altered.

Improvement can only happen if I am willing to listen to criticism so I plan to continue soliciting feedback from students on all facets of the courses I teach.

### 5.6.3 Community

My ultimate goal in running a class is to create a learning environment where students feel responsible to themselves and others to participate. That means building a temporary community. Courses don't run too long, only a few months, so I try to hit the ground running, getting students to talk to me and to each other from day one.

Students that stick it through to the end of my courses know one another fairly well and know me as well. That connection echoes forward: I have written over 25 letters of recommendation for former students for scholarships, internships, and graduate school applications. An equal number have asked me to serve as a professional reference as they matriculate and move on to full-time work.

I am particular fond of students as they finish CS 310 with me as it ends the programming course sequence. As they transition towards upper-level courses, try to squeeze out the last course project, and fit in a few minutes of study time, I have found it beneficial to sponsor a celebration of the end of Fall semesters for CS 310 students. Lovingly dubbed “CodeFest”, this pizza/programming/pop music mix has served to cement the semester's achievements. During Fall 2016, Prof. Carver was intrigued enough by the concept to join us for several hours rendering assistance to students as the ground away on their last assignment. I was also present to provide pizza, caffeine, and as many answers as I could rattle off before the Engineering building closed at 11pm.

As for my own community, I have taken the following concrete steps to build the capacity of my department and colleagues.

- I was recruited by Jill Nelson to run a SIMPLE teaching development group in the computer science department. We meet regularly to discuss issues related to teaching our large, technical classes. We have read through most of “How Learning Works” by Ambrose et al. and are moving to study “Teaching and Learning STEM” by Felder and Brent.
- I have been an active member of the undergraduate studies committee which works on all things undergrad for our department. I have developed the CS Departmental Honors program and contributed to the department recommendations on IT learning outcomes for all students at GMU. I drew from my experience in CS 105 and CS 100 for the latter.



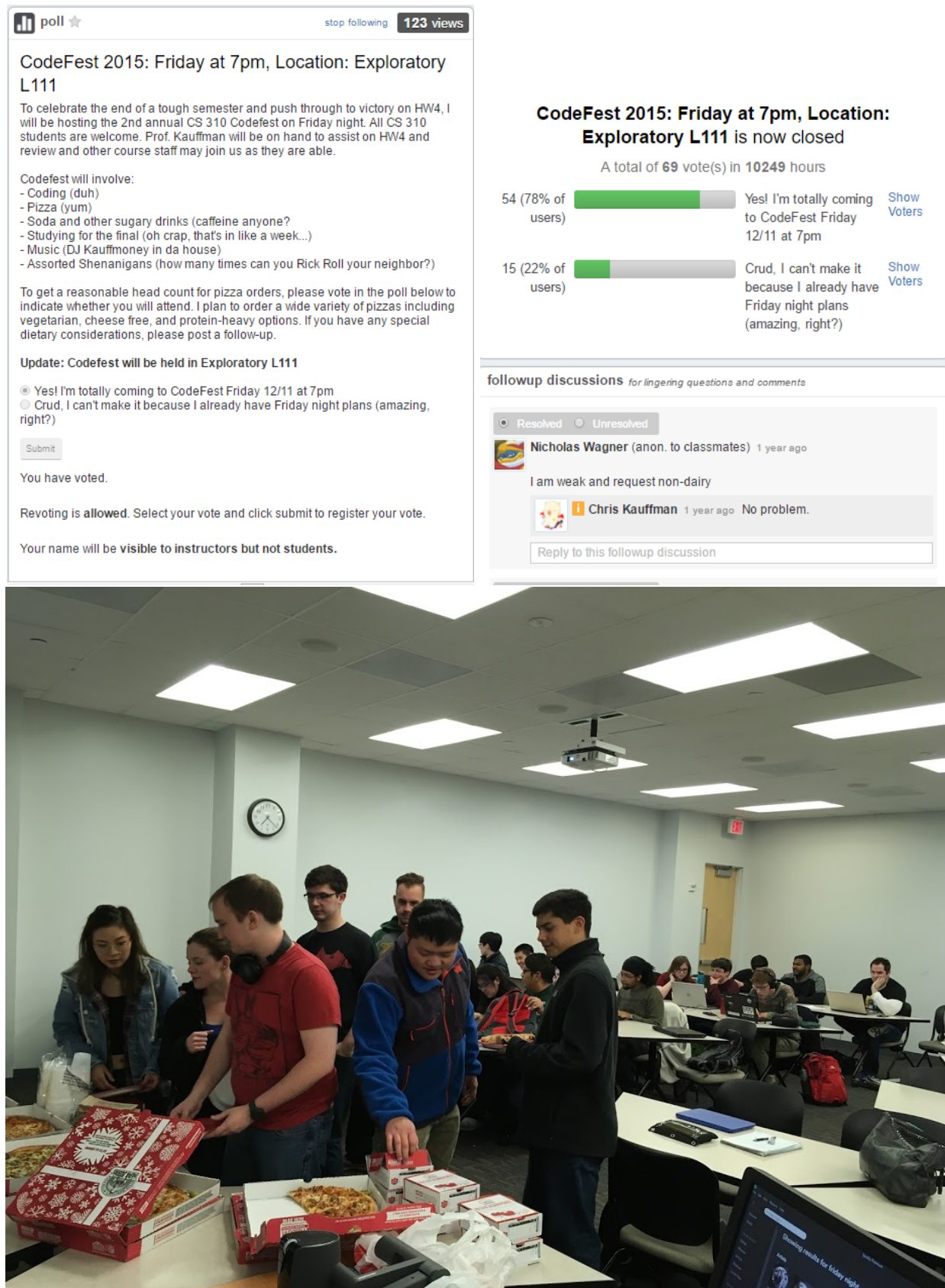


Figure 27: Top: Poll to count participation in CodeFest 2015 along with requests for Pizza. Bottom: CodeFest 2016, the third annual celebration of the end of CS 310 with pizza, music, and, of course, coding.

- I have contributed to a number of curriculum reforms in the undergraduate CS program including the recent development of a new course, CS 110, which will replace CS 105 next year. This course will plug gaps I helped identify during an curriculum analysis exercise we conducted to align ourselves with the standards set by the ACM, the authority on CS education.

Here are some closing comments from students on my efforts to build a community in every class.

I was a transfer student to Mason and during my first semester, felt a bit out of place. However, in professor Kauffmans Data Structures course everyone belonged. –*Ananya Dhawan*

Although I myself am not a computer science major, I was made to feel welcome in his [CS 100] classroom; he actively encourages and cultivates a rich and enjoyable learning environment, immersing his students in his lectures with great genius and genuineness. –*Selena Chaivaranon*

He made me understand that attending a university is about being a part of a community, making a positive impact, and developing relationships with your comrades, as well as your superiors, that help you all grow. *Taylor Kholer*

The clear fact that he cares about his students is exemplified by last semesters party. Professor Kauffman used his own time and money to organize a CodeFest pizza party for the Computer Science students in his Data Structures class. Students brought their code to work on with assistance from Professor Kauffman, other CS professors, and their fellow students. Professor Kauffman even spent extra time helping me debug unfamiliar code that wasnt even for his class. This was much needed relief right before finals arrived and a great way to celebrate the coming end of the semester. –*Will Crosswait*

.1in

Throughout my four years at Mason, Dr. Kauffman has been the professor who I look to for advice when it comes to my interests and aspirations in computer science. He has given me his opinions on many different fields of computer engineering like android and internet of things development. He is someone students can look up to and talk to about their aspirations and goals in life. –*Emmanuel Tagoe*

## 5.7 Sample Materials

This section contains some sample teaching materials which I use in several different classes. Each is described below.

**CS 310 Week 2 Lecture Slides** This slide deck is used to introduce students to Big-O notation which is used to describe algorithm complexity. It covers notation, theory, and the relation of Big-O to actual computer code. Included in the slides are exercises and diagrams to reinforce the concepts.

**CS 310 TripleStore Project** This project has been in use since I started CS 310. It is one of my shorter projects but it teaches an important lesson about using data structures to efficiently locate information. Students develop a small database program which has three columns of information. For this, they utilize a predefined data structure called a Red-Black tree. Diagrams within the assignment illustrate how to exploit the structure of this tree to quickly answer queries from users. Also present in the assignment are elements mentioned earlier such as a granular table of contents, structure of the project, and demonstration sessions.

**CS 211 CensoredWriter Lab** I developed this lab during CS 211 to provide a concrete activity that teaches the programming concept of inheritance. In it, students modify an existing class for which they cannot see the source code. The great strength of inheritance is that it still enables modifications to be made in this case which the students do to “censor” output of certain words.

**CS 100 Mini-Exam 4** I use mini-exams in CS 100, short evaluations interspersed throughout the semester to lower the stakes of the assessment and push it closer to the HW and student learning associated with the exam. This is the final mini-exam in CS 100 which covers some social aspects of computing such as data privacy along with some technical concepts such as random number generation. Mini-exams run for 30 minutes and are open-resource: students may use their notes, textbooks, and other resources while taking the exam.

**CS 499 Mini-Exam 1** I also used mini-exams in my CS 499 course as the course naturally lent itself to four main areas of study with one mini-exam for each. This mini-exam covers the first area of the class on theory and basics of constructing parallel algorithms.

**CS 100 Final Feedback Form and Results** I collect a lot of feedback, often through midterm and final “Customer Feedback” surveys which are anonymous and more customized to the course over what the generic university evaluations ask about. This survey starts with the questionnaire and shows the tabulated results for each question over a couple years of asking.

**CS 211 Midterm Feedback Results** This midterm feedback survey was given via blackboard in Spring 2015. The questions are repeated in the titles of each graph which shows fairly detailed information about which we were concerned. Particularly, Prof. Snyder and I were interested in how much students were collaborating during labs (allowed) and how useful they were finding the discussion their lab staff was facilitating. My background in data analysis is a great boon when plotting complex information which comes from such surveys.

## CS 310: Order Notation (aka Big-O and friends)

Chris Kauffman

Week 1-2

## Logistics

## At Home

- ▶ Read Weiss Ch 1-4: Java Review
- ▶ Read Weiss Ch 5: Big-O
- ▶ Get your java environment set up
- ▶ Compile/Run code for Max Subarray problem from first lecture

## Goals

- ▶ Finish up Course Mechanics
- ▶ Basic understanding of Big O and friends

## Announcement: UMD Diversity in Computing Summit

- ▶ <http://mcwic.cs.umd.edu/events/diversity>
- ▶ Keynote, talks, networking for current students
- ▶ Monday, November 7, 2016
- ▶ College Park Marriott Hotel, Hyattsville, MD
- ▶ \$35 for students before 10/1, \$50 after

*Through informative workshops and dynamic speakers, the Summit will emphasize inclusive computing efforts that address the positive impact that underrepresented groups have and will continue to have on the future of technology.*

## Course Mechanics

Finish up course mechanics from last time (first slide deck)

## How Fast/Big?

Algorithmic time/space complexity depend on **problem size**

- ▶ Often have some input parameter like  $n$  or  $N$  or  $(M, N)$  which indicates problem size
- ▶ Talk about time and space complexity as *functions* of those parameters
- ▶ **Example:** For an input array of size  $N$ , the maximum element can be found in  $5 * N + 3$  operations while the array can be sorted in  $2N^2 + 11N + 7$  operations.
- ▶ Big-O notation: bounding how fast functions grow based on input

## It's Show Time!

Not *The* Big O

## Just Big O

$T(n)$  is  $O(F(n))$  if there are positive constants  $c$  and  $n_0$  such that

- ▶ When  $n \geq n_0$
- ▶  $T(n) \leq cF(n)$

Bottom line:

- ▶ If  $T(n)$  is  $O(F(n))$
- ▶ Then  $F(n)$  grows as fast or faster than  $T(n)$

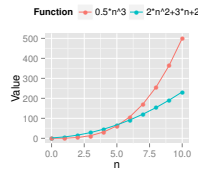
## Show It

Show

$$f(n) = 2n^2 + 3n + 2 \text{ is } O(n^3)$$

- Pick  $c = 0.5$  and  $n_0 = 6$

$n$	$f(n)$	$0.5n^3$
0	2	0
1	7	0
2	16	4
3	29	13
4	46	32
5	67	62
6	92	108
7	121	171



How about the opposite? Show

$$g(n) = n^3 \text{ is } O(2n^2 + 3n + 2)$$

## Basic Rules

- Constant additions disappear
  - $N + 5$  is  $O(N)$
- Constant multiples disappear
  - $0.5N + 2N + 7$  is  $O(N)$
- Non-constant multiples multiply:
  - Doing a constant operation  $2N$  times is  $O(N)$
  - Doing a  $O(N)$  operation  $N/2$  times is  $O(N^2)$
  - Need space for half an array with  $N$  elements is  $O(N)$  space overhead
- Function calls **are not free** (including library calls)
  - Call a function which performs 10 operations is  $O(1)$
  - Call a function which performs  $N/3$  operations is  $O(N)$
  - Call a function which copies object of size  $N$  takes  $O(N)$  time and uses  $O(N)$  space

## Bounding Functions

- Big O: **Upper** bounded by ...
  - $2n^2 + 3n + 2$  is  $O(n^3)$  and  $O(2^n)$  and  $O(n^2)$
- Big Omega: **Lower** bounded by ...
  - $2n^2 + 3n + 2$  is  $\Omega(n)$  and  $\Omega(\log(n))$  and  $\Omega(n^2)$
- Big Theta: **Upper and Lower** bounded by
  - $2n^2 + 3n + 2$  is  $\Theta(n^2)$
- Little O: **Upper** bounded by **but not lower** bounded by ...
  - $2n^2 + 3n + 2$  is  $o(n^3)$

## Growth Ordering of Some Functions

Name	Leading Term	Big-Oh	Example
Constant	1, 5, c	$O(1)$	2.5, 85, 2c
Log-Log	$\log(\log(n))$	$O(\log \log n)$	$10 + (\log \log n + 5)$
Log	$\log(n)$	$O(\log(n))$	$5 \log n + 2$
Linear	$n$	$O(n)$	$2.4n + 10$ $10n + \log(n)$
N-log-N	$n \log n$	$O(n \log n)$	$3.5n \log n + 10n + 8$
Super-linear	$n^{1+\epsilon}$	$O(n^{1+\epsilon})$	$2n^{1.2} + 3n \log n - n + 2$
Quadratic	$n^2$	$O(n^2)$	$0.5n^2 + 7n + 4$ $n^2 + n \log n$
Cubic	$n^3$	$O(n^3)$	$0.1n^3 + 8n^{1.5} + \log(n)$
Exponential	$a^n$	$O(2^n)$	$8(2^n) - n + 2$
Factorial	$n!$	$O(10^n)$	$100n^{500} + 2 + 10^n$ $0.25n! + 10n^{100} + 2n^2$

## Constant Time Operations

The following take  $O(1)$  Time

- Arithmetic operations (add, subtract, divide, modulo)
  - Integer ops usually practically faster than floating point
- Accessing a stack variable
- Accessing a field of an object
- Accessing a single element of an array
- Doing a **primitive** comparison (equals, less than, greater than)
- Calling a function/method but **NOT waiting for it to finish**

The following take more than  $O(1)$  time (how much)?

- Raising an arbitrary number to arbitrary power
- Allocating an array
- Checking if two Strings are equal
- Determining if an array or ArrayList contains() an object

## Common Patterns

- Adjacent Loops Additive:  $2 \times n$  is  $O(n)$ 

```
for (int i=0; i<N; i++){
    blah blah blah;
}
for (int j=0; j<N; j++){
    yakkety yack;
}
```
- Nested Loops Multiplicative usually polynomial
  - 1 loop,  $O(n)$
  - 2 loops,  $O(n^2)$
  - 3 loops,  $O(n^3)$
- Repeated halving usually involves a logarithm
  - Binary search is  $O(\log n)$
  - Fastest sorting algorithms are  $O(n \log n)$
  - Proofs are harder, require solving recurrence relations

Lots of special cases so be careful

## Practice

- Two functions to reverse an array. Discuss
- Big-O estimates of **runtime** of both
  - Big-O estimates of **memory overhead** of both
    - Memory overhead is the amount of memory in addition to the input required to complete the method
  - Which is practically better?
  - What are the **exact** operation counts for each method?

## reverseE

```
public static
void reverseE(Integer a[]){
    int n = a.length;
    Integer b[] = new Integer[n];
    for(int i=0; i<n; i++){
        b[i] = a[n-1-i];
    }
    for(int i=0; i<n; i++){
        a[i] = b[i];
    }
}
```

## reverseI

```
public static void
reverseI(Integer a[]){
    int n = a.length;
    for(int i=0; i<n/2; i++){
        int tmp = a[i];
        a[i] = a[n-1-i];
        a[n-1-i] = tmp;
    }
    return;
}
```

## Much Trickier Exercise

```
public static String toString( Object [ ] arr )
{
    String result = "[";
    for( String s : arr )
        result += s + " ";
    result += "]";
    return result;
}
```

- Give a Big-O estimate for the runtime
- Give a Big-O estimate for the memory overhead

## Multiple Input Size

What if "size" has two parameters?

- $m \times n$  matrix
- Graph with  $m$  vertices and  $n$  edges
- Network with  $m$  computers and  $n$  cables between them

## Exercise: Sum of a Two-D Array

Give the runtime complexity of the following method.

```
public int sum2D(int [][] A){
    int M = A.length;
    int N = A[0].length;
    int sum = 0;
    for(int i=0; i<M; i++){
        for(int j=0; j<N; j++){
            sum += A[i][j];
        }
    }
    return sum;
}
```

## What if I have no idea?

Analyzing a complex algorithm is hard. More in CS 483.

- Most analyses in here will be straight-forward
- Mostly use the common patterns

If you haven't got a clue looking at the code, *run it and check*

- This will give you a much better sense

## Observed Runtimes of Maximum Subarray

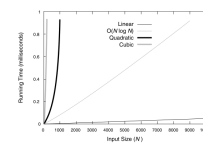
	Figure 5.4	Figure 5.5	Figure 7.20	Figure 5.8
$N$	$O(N^2)$	$O(N^2)$	$O(N \log N)$	$O(N)$
10	0.000001	0.000000	0.000001	0.000000
100	0.000288	0.000019	0.000014	0.000005
1,000	0.223111	0.001630	0.000154	0.000053
10,000	218	0.133064	0.001630	0.000533
100,000	NA	13.17	0.017467	0.005571
1,000,000	NA	NA	0.185363	0.056338

figure 5.10  
Observed running times (in seconds) for various maximum contiguous subsequence sum algorithms

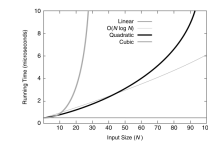
Weiss pg 203

## Idealized Functions

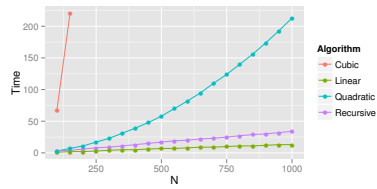
## Smallish Inputs



## Larger Inputs



## Actual Data for Max-Subarray



- ▶ Where did this data come from?
- ▶ Does this plot confirm our analysis?
- ▶ How would we check?

## Playing with MaxSumTestBetter.java

Let's generate part of the data, demo in  
w01-1-code/MaxSumTestBetter.java

- ▶ Edit: Running a main, n=100 to 100,000, multiply by 10
- ▶ Try in DrJava
- ▶ Demo interactive loop

## Analysis

## Linear

```
> summary(linmod)
```

```
Coefficients:
              Estim Pr(>|t|)
(Intercept)  7.26  <2e-16 ***
poly(N, 1)   16.25  <2e-16 ***
poly(N, 2)   -0.34   0.287
poly(N, 3)   -0.01   0.962
```

Why these coefficients?

## Quadratic

```
> summary(quadmod)
```

```
Coefficients:
              Estim Pr(>|t|)
(Intercept) 83.89  <2e-16 ***
poly(N, 1)   278.16  <2e-16 ***
poly(N, 2)    54.75  <2e-16 ***
poly(N, 3)   -0.24   0.562
```

## Take-Home

## Today

Order Analysis captures big picture of algorithm complexity

- ▶ Different functions grow at different rates
- ▶ Big O upper bounds
- ▶ Big Theta tightly bounds

## Next Time

- ▶ What are the limitations of Big-O?
- ▶ Reading: finish Ch 5, Ch 15 on ArrayList
- ▶ Suggested practice: Exercises 5.39 and 5.44 which explore string concatenation, why obvious approach is slow for lots of strings, alternatives

Last Updated: 2016-12-01 Thu 16:35

## CS 310 HW 4: TripleStore Database

- **Due: 11:59pm Friday 12/9/2016**
- Approximately 8.75% of total grade
- Submit to Blackboard
- There is no code distribution for this assignment([notes](#))
- There are no milestones for this project
- Final Test Cases: Available by Wed 11-30-2016
  - [RecordTests.java](#) 56 tests (Updated Thu Dec 1 16:34:57 EST 2016)
  - [TripleStoreTests.java](#) 45 tests
  - [HW4Tests.java](#) runs above tests

### CHANGELOG:

#### Thu Dec 1 16:35:04 EST 2016

Minor update to `RecordTests.java` to fix a bug in `doCompare(...)`.

#### Thu Dec 1 00:11:48 EST 2016

Test cases for HW4 are now available linked from the top of spec. `RecordTests.java` and `TripleStoreTests.java` exercise their respective classes while `HW4Tests.java` simply runs all the tests in these two.

#### Tue Nov 29 17:21:58 EST 2016

Some of the examples of `query()` results indicated that an `ArrayList` is returned. This should just be a general `List` which may be either an `ArrayList` or `LinkedList`

In response to a [@805 a section outlining how wild fields compare has been added](#).

## Table of Contents

- [1. Overview and Goals](#)
  - [1.1. TripleStore Database](#)
  - [1.2. TripleStore Examples](#)
  - [1.3. TripleStore Demo](#)
  - [1.4. Basic operations](#)
  - [1.5. Records](#)
  - [1.6. Fast Access and Wildcards](#)
- [2. Class Architecture](#)
  - [2.1. Project Files](#)
  - [2.2. Built-in Classes of Interest](#)
  - [2.3. Constraints](#)
  - [2.4. Record.java](#)
  - [2.5. TripleStore.java](#)
- [3. Record Class Implementation](#)
  - [3.1. Creation and Basic Functionality](#)
  - [3.2. Record.toString\(\)](#)
  - [3.3. Records with Wild Cards](#)
  - [3.4. Record.matches\(\)](#)
  - [3.5. Record Comparators](#)
  - [3.6. Wildcards and Comparisons](#)
- [4. TripleStore Class Implementation](#)
  - [4.1. TripleStore Constructor and Basic methods](#)
  - [4.2. TripleStore.toString\(\)](#)
  - [4.3. TripleStore.add\(\)](#)
  - [4.4. TripleStore.query\(\)](#)
  - [4.5. TripleStore.remove\(\)](#)
- [5. Grading](#)
  - [5.1. Final Automated Tests \(50%\)](#)
  - [5.2. Final Manual Inspection \(50%\)](#)
- [6. Final Manual Inspection Criteria](#)
  - [6.1. \(10%\) Record Design](#)
  - [6.2. \(10%\) Record Comparator Design and Use](#)
  - [6.3. \(20%\) TripleStore Method Runtime Complexities](#)
  - [6.4. \(5%\) Coding Style and Readability](#)
  - [6.5. \(5%\) Correct Project Setup](#)
- [7. Setup and Submission](#)
  - [7.1. HW Directory](#)
  - [7.2. ID.txt](#)
  - [7.3. Penalties](#)
  - [7.4. Submission: Blackboard](#)

## 1 Overview and Goals

### 1.1 TripleStore Database

Databases pervade the computing world as they provide an efficient way to store information that can be accessed and manipulated in a flexible fashion. Most database systems provide a way to create a *table* which has many entries (usually written as the rows) and many columns (the different fields of each row). Databases may contain multiple different tables with varying table sizes. This assignment centers around implementing a very simple database in Java called a *triplestore*, so named because it has a single table with many rows but only three columns ever. For the purpose of this assignment, these columns are named **Entity**, **Relation**, and **Property**. Two examples of triplestores are shown in tabular format below. The first triplestore is small and somewhat trivial while the second has immediate commercial applications

### 1.2 TripleStore Examples

#### Comical Example

	entity	relation	property
1	Willie	ISA	human
2	Alf	ISA	alien



	entity	relation	property
5	Alf	EATS	cat
3	Lynn	ISA	human
4	Lucky	ISA	cat
6	Lynn	EATS	veggies
7	Lucky	LIKESTO	purr
8	Lucky	EATS	catfood
9	Alf	EATS	veggies

**Commercial Example**

	entity	relation	property
1	1	ISA	boots
2	1	AVAILABLE	4
3	1	COSTS	32.50
4	1	SIZE	10
5	2	ISA	boots
6	2	AVAILABLE	6
7	2	COSTS	32.50
8	2	SIZE	9
9	3	ISA	shirt
10	3	COLOR	red
11	3	COSTS	11.99
12	3	SIZE	XL
13	3	AVAILABLE	2
14	3	MATERIAL	cotton
15	3	DESIGNER	Swankums
16	4	ISA	watch
17	4	ISA	accessory
18	4	AVAILABLE	1
19	4	COSTS	302.29
20	4	DESIGNER	PricePlusPlus
21	4	MATERIAL	gold
22	5	ISA	hat
23	5	ISA	accessory
24	5	MATERIAL	wool
25	5	COSTS	\$29.99
26	5	COLOR	gray
27	5	AVAILABLE	0
28	5	ISA	warm clothing
29	5	DESIGNER	Swankums

**1.3 TripleStore Demo**

The quickest way to get an immediate sense of how a `TripleStore` looks and feels is to examine the following demo done in DrJava's interaction pane. JGrasp provides the same functionality for experimentation.

In the demo, most results of assignments or object returns are shown on the following line by NOT putting a semicolon at the end of the line. This is the default behavior in DrJava while in JGrasp you must type a variable name on its own line to see its printed representation. Comments are also inserted to add some guidance to the demo

```
Welcome to DrJava.
> TripleStore t = new TripleStore();
> t

> // Empty TripleStore

> // --- ADD ---
> boolean b = t.add("Willie","ISA","human");
> b
true
> // Successful add returns true
```

2/5/2017

CS 310 HW 4: TripleStore Database

```

> t.add("Alf", "ISA", "alien");
> t
    Alf      ISA      alien
    Willie   ISA      human
> // Two Records in the TripleStore

> b = t.add("Willie", "ISA", "human");
> b
false
> // Duplicates are not allowed
> t
    Alf      ISA      alien
    Willie   ISA      human
> // Still only two Records in the TripleStore

> t.add("Lynn", "ISA", "human");
> t.add("Lucky", "ISA", "cat");
> t.add("Alf", "EATS", "cat");
> t.add("Lynn", "EATS", "veggies");
> t.add("Lucky", "LIKESTO", "purr");
> t.add("Lucky", "EATS", "catfood");
p> t.add("Alf", "EATS", "veggies");
> t
    Alf      EATS      cat
    Alf      EATS      veggies
    Alf      ISA      alien
    Lucky    EATS      catfood
    Lucky    ISA      cat
    Lucky    LIKESTO   purr
    Lynn     EATS      veggies
    Lynn     ISA      human
    Willie   ISA      human
> // Variety of things in the TripleStore

> // --- QUERY ---
> import java.util.*;
> List<Record> results;
> results = t.query("Alf", "ISA", "alien");
> results
[    Alf      ISA      alien]
> // query returns an List
> // Records match exactly, 1 result

> t.getWild()
""
> results = t.query("Alf", "ISA", "")
[    Alf      ISA      alien]
> // query with a wild card matched 1 record
> results = t.query("Alf", "EATS", "")
[    Alf      EATS      cat,    Alf      EATS      veggies]
> // query with a wild card matched 2 records
> results = t.query("Alf", "", "")
[    Alf      EATS      cat,    Alf      EATS      veggies,    Alf      ISA      alien]
> // query with several wild cards matched 3 records

> t.query("", "ISA", "human")
[    Lynn     ISA      human,    Willie   ISA      human]
> t.query("", "", "human")
[    Lynn     ISA      human,    Willie   ISA      human]
> t.query("", "", "cat")
[    Alf      EATS      cat,    Lucky    ISA      cat]
> t.query("", "ISA", "")
[    Alf      ISA      alien,    Lucky    ISA      cat,    Lynn     ISA      human,    Willie   ISA      human]
> // wildcards can appear in any of entity, relation, property

> // --- REMOVE ---
> int nrm = t.remove("Alf", "EATS", "veggies")
1
> // successful removal, changes database
> t
    Alf      EATS      cat
    Alf      ISA      alien
    Lucky    EATS      catfood
    Lucky    ISA      cat
    Lucky    LIKESTO   purr
    Lynn     EATS      veggies
    Lynn     ISA      human
    Willie   ISA      human

> nrm = t.remove("Alf", "EATS", "fruit")
0
> // unsuccessful remove
> nrm = t.remove("Alf", "EATS", "veggies")
0
> // record no longer exists

> nrm = t.remove("Alf", "", "")
2
> t
    Lucky    EATS      catfood
    Lucky    ISA      cat
    Lucky    LIKESTO   purr
    Lynn     EATS      veggies
    Lynn     ISA      human

```

2/5/2017

CS 310 HW 4: TripleStore Database

```

    Willie      ISA      human
> t.remove("","*", "*")
6
> t

> // wild cards can remove many records

> --- WILDCARDS ---
> t.getWild()
"*"
> // Can add wild card strings as records
> t.add("Alf", "ISA", "*")
> t
Alf      ISA      *
> t.add("Alf", "ISA", "whateva")
true
> t.add("Alf", "ISA", "alien")
true
> t
Alf      ISA      *
Alf      ISA      alien
Alf      ISA      whateva
> t.query("Alf", "ISA", "*")
[Alf      ISA      *      , Alf      ISA      alien      , Alf      ISA      whateva ]

> t.setWild("whateva")
> // The * is no longer the wild card, matches only equal strings
> t.query("Alf", "ISA", "*")
[Alf      ISA      *      ]

> // String "whateva" is not the wild card, matches anything
> t.query("Alf", "ISA", "whateva")
[Alf      ISA      *      , Alf      ISA      alien      , Alf      ISA      whateva ]
> t.setWild("whateva")
> t.getWild()
"whateva"
> t.query("Alf", "ISA", t.getWild())
[Alf      ISA      *      , Alf      ISA      alien      , Alf      ISA      whateva ]

```

#### 1.4 Basic operations

A TripleStore is like many other data structures in that it provides add, remove, and find functionality. Similar to the binary search trees discussed in class, no duplication is allowed in a TripleStore so `add()` will not add a record that is already present. Slightly more general than other data structures is that the find functionality, named `query()`, may return multiple results in a collection and the remove functionality may remove more than one record from the triplestore. The detailed functionality of each method is described in the implementation sections in the *section on TripleStore*

#### 1.5 Records

The records stored in the TripleStore will be objects of class `Record`. At its core, a `Record` is just a way cart around the `String` objects `entity`, `property`, `relation` in a triplet as shown in the tables above. `Records` are **immutable** in that, once they are created, they cannot change, in much the same way that when a `String` is created, it cannot be changed. This sacrifices some space and time efficiency for *much* simpler reasoning about data structures involving `Record`. The fields of a `Record` are accessible by methods named after the fields.

```

public String entity();      // Who
public String relation();    // How
public String property();    // What

```

In addition to these, every `Record` will have a unique identifier which is returned by the `id()` method. Whenever a `Record` is created, a new unique ID number is assigned to it which is accessible through the `id()` method.

```

public int id();             // Must be unique

```

#### 1.6 Fast Access and Wildcards

Each triplestore keeps track of a **wild card** string which allows flexible queries and removals that can handle multiple matching records. The default wild card is `*` (star) but it may be changed to any string using methods described below. A query involving a wildcard may match more than one record. Providing fast location of all `Records` that match a query involving wildcards is the primary focus of this assignment.

There are several options for arranging `Records` in memory, but providing a reasonably efficient combination of `add`, `query`, `remove` operations,  $O(\log N)$  suggests the use of balanced binary search trees.

In addition to this, `query()` calls which use wild cards may return multiple records. The sorted nature of binary search trees will come in handy here. For example, consider a binary tree which sorts each stored `Record` starting with the `String` `entity` field breaking ties by examining `relation` and breaking further ties with `property`. An example is shown below which does this in `Grasp` and uses an pre-built red-black tree in the java library. Note the color of nodes is shown accurately as red or black.

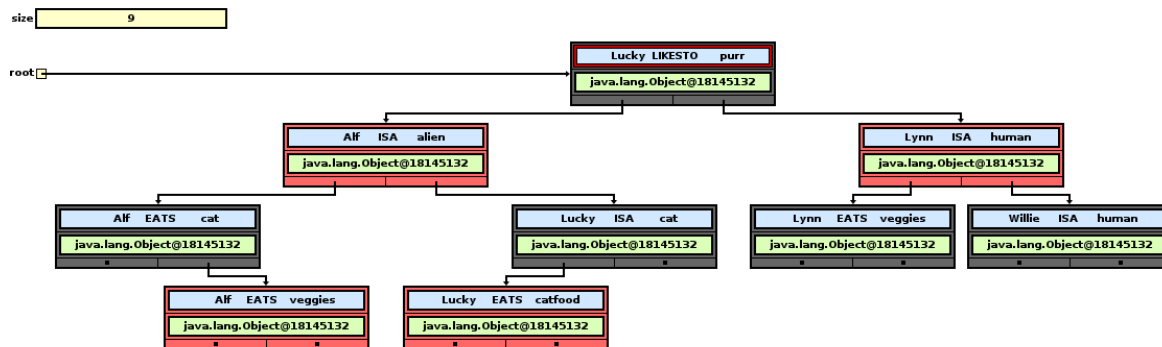


Figure 1: Red-Black Tree with Entity-Relation-Property Sorting

If `t.query("Alf", "*", "*")` is executed, an effective strategy is to find the "least" node involving `Alf` and begin an in-order traversal from that point. This would lead to the sequence of `Records`

```
Alf EATS cat; Alf EATS veggies; Alf ISA alien; Lucky EATS catfood; ...
```

which contains relevant records to the query. During the traversal, if each record is checked to determine if it **matches** the query, the first three clearly match while the fourth involving `Lucky` does not match. Since the tree is sorted, there can be no more `Alf` records and the method can stop the traversal and return the three records found that match the query.

This process involves the following steps.

1. Find the smallest matching record
2. Start an in-order traversal at that record
3. For each record in the traversal
  - o If the record matches the query, add it to the growing collection
  - o Otherwise there can be no more matching records so stop and return

Step 1 can be done in  $O(\log N)$  time as one simply needs to search our tree for the "spot" where the query would belong. Step 2 is  $O(1)$  if the tree has parent pointers or some facility for supporting in-order traversals of the tree efficiently (which is the case for production quality tree implementations). Finally, Step 3 visits each matching node plus one additional node that indicates there will be no other matches. If there are  $K$  matches, Step 3 takes  $O(K)$  time. Thus, the total complexity is  $O(K + \log N)$ .

Unfortunately, the above tree will not help us with a query such as `t.query("ISA", "*", "*")` as the tree is not sorted appropriately: the matching `ISA` records are spread throughout the tree. A simple solution to this is to use **additional** trees storing the same data but sorted differently. For example, the below tree stores the same data but sorted on relation, then property, then entity.

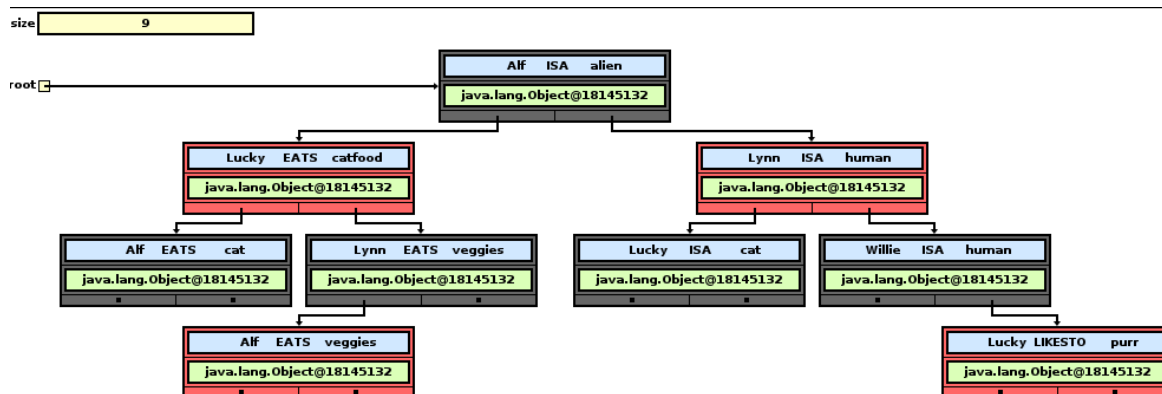


Figure 2: Red-Black Tree with Relation-Property-Entity Sorting

The records matching `t.query("ISA", "*", "*")` are all stored linked together starting with the root.

A data structure that facilitates fast lookup into a database is usually referred to as an **index**. When a user calls `t.query()` it is up to the `TripleStore` class to analyze the query to determine the most efficient means to find matching records, usually by selecting an index if one is available. **All queries to triplestore should be efficient** regardless of the query `query()` should meet the target  $O(K + \log N)$  complexity target. This will require several indexes.

The trade-off of this fast lookup is that when records are added or removed, all indexes must be updated. The cost of `add` will be  $O(\log N)$  as it deals with only a single record. The cost of `remove` is  $O(K \times \log N)$  as multiple records may need to be removed. There is also a space trade-off as the stored records require more data structures to facilitate fast lookup which means more memory is used.

## 2 Class Architecture

### 2.1 Project Files

2/5/2017

CS 310 HW 4: TripleStore Database

The following files are relevant to the HW. When submitting, place them in your [HW Directory](#), zip that directory and submit. **Always submit a zip file** not `tar.gz`, not `bzip`, not `7zip`, just vanilla zip files. For additional instructions see [Setup and Submission](#)

File	State	Notes
Record.java	Create	Triples stored in the TripleStore database. May subclass to support queries.
TripleStore.java	Create	Database with 3 columns. Stores records via <code>add()</code> and allows <code>find()</code> and <code>remove()</code> .
junit-c310.jar	Testing	JUnit library for testing. Copy over from previous projects.
ID.txt	Create	Identifying information

## 2.2 Built-in Classes of Interest

It is **strongly** recommended that you not try to implement balanced trees yourself. Aside from the obvious difficulty, this is not an analogous activity to what programming in the wild is like. Instead, consider the use of the following classes

### [java.util.TreeSet](#)

Implemented using a red-black tree under the hood, this is an incredibly useful class to become familiar with. While [TreeSet](#) is fairly complex, it provides nearly all the functionality required for this assignment without even the need to `extend` it. Pay particular attention to methods which provide a view of the tree as a `SortedSet` such as `tailSet()` which essentially give a "view" of a subset of the tree; an iterator can be obtained from this subset which will traverse the tree in order. The standard `add` and `remove` methods are also present for the `TreeSet`.

### [java.util.Comparator](#)

An interface that allows one to create objects which compare other objects. Any binary search tree will need a means of comparing objects and if the same objects are to be sorted in multiple ways, `Comparators` are the standard way to do this. Note that `TreeSet` has a constructor which takes a `Comparator` as an argument: any insertions or lookups use the given comparator as a way to navigate the sorted tree.

Mastering these two classes will involve reading their documentation carefully and experimenting in interactive loops or with your own compiled code. This is typically the case when using other people's code and is what developers in the wild do far more than actually writing their own code. It is worth the effort to make the project easier and to develop reading/understanding skills so that you don't re-invent the tree.

*Note:* You should not need to examine the source code for `TreeSet` to complete the assignment but the curious and ambitious wizard may find it enlightening.

## 2.3 Constraints

The only code you may use for this project falls into two categories

1. Classes in the Java standard library such as `TreeSet`
2. Classes and code you write yourself

There may be some triplestore java implementations out there but you are not to use them for this project; such action would constitute an honor code violation.

## 2.4 Record.java

```
// Immutable. Stores 3 strings referred to as entity, relation, and
// property. Each Record has a unique integer ID which is set on
// creation. All records are made through the factory method
// Record.makeRecord(e,r,p). Record which have some fields wild are
// created using Record.makeQuery(wild,e,r,p)
public class Record{

    // Return the next ID that will be assigned to a Record on a call to
    // makeRecord() or makeQuery()
    public static int nextId();

    // Return a stringy representation of the record. Each string should
    // be RIGHT justified in a field of 8 characters with whitespace
    // padding the left. Java's String.format() is useful for padding
    // on the left.
    public String toString();

    // Return true if this Record matches the parameter record r and
    // false otherwise. Two records match if all their fields match.
    // Two fields match if the fields are identical or at least one of
    // the fields is wild.
    public boolean matches(Record r);

    // Return this record's ID
    public int id();

    // Accessor methods to access the 3 main fields of the record:
    // entity, relation, and property.
    public String entity();

    public String relation();

    public String property();

    // Returns true/false based on whether the the three fields are
    // fixed or wild.
    public boolean entityWild();

    public boolean relationWild();

    public boolean propertyWild();

    // Factory method to create a Record. No public constructor is
    // required.
    public static Record makeRecord(String entity, String relation, String property);
```

2/5/2017

CS 310 HW 4: TripleStore Database

```

// Create a record that has some fields wild. Any field that is
// equal to the first argument wild will be a wild card
public static Record makeQuery(String wild, String entity, String relation, String property);

// Comparators that compare Records based on different orderings of
// their fields. The names of the Comparators correspond to the
// order in which they compare fields: ERPCompare compares Entity
// (E), then Relation (R), then property (P). Likewise for
// RPECompare and PER compare.
public static final Comparator<Record> ERPCompare;

public static final Comparator<Record> RPECompare;

public static final Comparator<Record> PERCompare;

    public int compare(Record x, Record y); return 0; }

}

```

### 2.5 TripleStore.java

```

// Three-column database that supports query, add, and remove in
// logarithmic time.
public class TripleStore{

    // Create an empty TripleStore. Initializes storage trees
    public TripleStore();

    // Access the current wild card string for this TripleStore which
    // may be used to match multiple records during a query() or
    // remove() call
    public String getWild();

    // Set the current wild card string for this TripleStore
    public void setWild(String w);

    // Ensure that a record is present in the TripleStore by adding it
    // if necessary. Returns true if the addition is made, false if the
    // Record was not added because it was a duplicate of an existing
    // entry. A Record with any fields may be added to the TripleStore
    // including a Record with fields that are equal to the
    // TripleStore's current wild card. Throws an
    // IllegalArgumentException if any argument is null.
    //
    // Target Complexity:  $O(\log N)$ 
    // N: number of records in the TripleStore
    public boolean add(String entity, String relation, String property);

    // Return a List of the Records that match the given query. If no
    // Records match, the returned list should be empty. If a String
    // matching the TripleStore's current wild card is used for one of
    // the fields of the query, multiple Records may be returned in the
    // match. An appropriate tree must be selected and searched
    // correctly in order to meet the target complexity. Throws an
    // IllegalArgumentException if any argument is null.
    //
    // TARGET COMPLEXITY:  $O(K + \log N)$ 
    // K: the number of matching records
    // N: the number of records in the triplestore.
    public List<Record> query(String entity, String relation, String property);

    // Remove elements from the TripleStore that match the parameter
    // query. If no Records match, no Records are removed. Any of the
    // fields given may be the TripleStore's current wild card which may
    // lead to multiple Records being matched and removed. Return the
    // number of records that are removed from the TripleStore. Throws
    // an IllegalArgumentException if any argument is null.
    //
    // TARGET COMPLEXITY:  $O(K * \log N)$ 
    // K: the number of matching records
    // N: the number of records in the triplestore.
    public int remove(String e, String r, String p);

    // Produce a String representation of the TripleStore. Each Record
    // is formatted with its toString() method on its own line. Records
    // must be shown sorted by Entity, Relation, Property in the
    // returned String.
    //
    // TARGET COMPLEXITY:  $O(N)$ 
    // N: the number of records stored in the TripleStore
    public String toString();

}

```

### 3 Record Class Implementation

```
public class Record
```

Databases contain "rows" and in the case of `TripleStore` each row comprises three fields which will be housed in instances of the `Record` class. It is an immutable class which carts around three strings. The class provides facilities for unique creation and creation of records which act as queries that can match other records. It also provides a set of comparators for the arrangement of `Records` in various ways in the `TripleStore`.

### 3.1 Creation and Basic Functionality

Fields and accessors

```
public int id();           // Must be unique
public String entity();    // Who
public String relation();  // How
public String property();  // What
```

Factory method

```
public static Record makeRecord(String e, String r, String p)
```

Notes

- You are free to specify your own constructors but testing code always uses the factory method `Record.makeRecord( . . )` which should return a record.
  - If any of the arguments `e`, `r`, `p` are `null`, throw an `IllegalArgumentException` with an informative message.
  - The ID number of a `Record` is accessible only through the `id()` method and never changes after creation.
  - The `id()` method must return an integer unique to every record but is not set by the user. A class level (`static`) private field is usually used for this kind of behavior.
- Example:

```
> Record r1 = Record.makeRecord("Alf", "ISA", "alien");
> Record r2 = Record.makeRecord("Alf", "ISA", "alien");
> r1.id()
45
> r2.id()
46
> r1.id() == r2.id()
false
```

### 3.2 Record.toString()

**Note:** the demos may not display records in exactly the right format. This section contains the expected format.

```
public String toString()
```

A specific format is required for `Record.toString()`.

- Each field is separated by 1 space.
- `entity` appears first and is right justified in a field of **8 characters**
- `relation` appears second and is right justified in a field of **8 characters**
- `property` appears third and is right justified in a field of **8 characters**
- Use of `String.format("%8s ")` or something similar is encouraged to create the formatted record string
- If any field exceeds 8 characters, take no special action acknowledging that this may cause tables of records to display undesirably
- `id` does not appear in `toString()`
- No special action needs to be taken if the fields of the record are wider than 8 characters: this may lead to ugly printing but fixing this problem is beyond the scope of project.

You will likely find the function `String.format()` useful for constructing building string representations.

Several records formatted as strings are below.

```
> Record r;
> r = Record.makeRecord("A", "B", "C");
> r
      A      B      C
> r.toString()
"      A      B      C "
> r = Record.makeRecord("Alf", "ISA", "alien");
> r
      Alf      ISA      alien
> r.toString()
"      Alf      ISA      alien "
> r = Record.makeRecord("12345678", "12345678", "12345678");
> r
12345678 12345678 12345678
> r.toString()
"12345678 12345678 12345678 "
// Take no special action when the fields are longer than 8 characters
> r = Record.makeRecord("1234567890", "123456789", "123456789012");
> r
1234567890 123456789 123456789012
> r.toString()
"1234567890 123456789 123456789012 "
```

### 3.3 Records with Wild Cards

```
public static Record makeQuery(String wild, String e, String r, String p)
```

This factory method returns a special kind of record, perhaps a subclass of `Record` which is able to *match* other records. If any of `e`, `r`, `p` equal the first argument `wild`, those fields should be marked as "wild" and can match any other string in `Record.matches()`.

```
public boolean entityWild()
public boolean relationWild()
public boolean propertyWild()
```

Each record has several simple accessors which determine if its fields are wild. A field is wild **only** if it exactly matches the wild card in `Record` according to the `String.equals()` method.

```
// Records created with makeRecord(..) never have wild fields
> Record notAQuery = Record.makeRecord("Alf", "*", "*");
> notAQuery.entityWild()
false
> notAQuery.relationWild()
false
> notAQuery.propertyWild()
false

// Queries created with makeQuery(..) have wild fields where they
// match the first argument string
> Record query = Record.makeQuery("*", "Alf", "*", "*");
> query
    Alf      *      *
> query.entityWild()
false
> query.relationWild()
true
> query.propertyWild()
true

// Any string can denote wild records;
> Record query2 = Record.makeQuery("wild", "X", "wild", "*");
> query2
    X      wild      *
> query2.entityWild()
false
> query2.relationWild()
true
> query2.propertyWild()
false
// despite being *, third field is not wild as the word 'wild' was
// chosen to denote wild fields in construction of query2
```

### 3.4 `Record.matches()`

```
public boolean matches(Record r)
```

- Determine if two `Records` *match*; return `true` if they do and `false` otherwise.
- Matching occurs when all the fields `entity`, `relation`, `property` match
- Fields match if either one or both is wild or both fields are exactly the same according to `String.equals()`
- Examples

```
> Record r1 = Record.makeRecord("Alf", "ISA", "alien");
> Record r2 = Record.makeRecord("Alf", "ISA", "alien");
> r1.matches(r2)
true
> Record r3 = Record.makeRecord("Alf", "EATS", "cat");
> r1.matches(r3)
false
> Record r4 = Record.makeQuery("*", "Alf", "ISA", "*")
    Alf      ISA      *
> r1.matches(r4)
true
>
> Record r5 = Record.makeQuery("*", "Alf", "*", "*");
> r1.matches(r5)
true
> r4
    Alf      ISA      *
> r5
    Alf      *      *
> r4.matches(r5)
true
```

### 3.5 `Record Comparators`

```
public static final Comparator<Record> ERPCompare
public static final Comparator<Record> RPECompare
public static final Comparator<Record> PERCompare
```

The purpose of these three objects is to allow `Records` to be arranged in different ways in a binary search tree. Each implement the `Comparator<Record>` interface so that they have the following method.



```
public int compare(Record r1, Record r2)
```

Each of the comparators compares fields of `Records` in a different order to determine which `Record` is sorted first. The orders are

- `ERPCompare`: entity relation property
- `RPECompare`: relation property entity
- `PERCompare`: property entity relation

Comparison starts with the first field listed. If they are equal, the next field listed is compared to break the tie, and if they are equal the final field is considered. The records are equal only if all of `entity`, `relation`, `property` match exactly.

For example given the records

```
Record abc = Record.makeRecord("a","b","c");
Record bca = Record.makeRecord("b","c","a");
Record cba = Record.makeRecord("c","b","a");
Record caa = Record.makeRecord("c","a","a");
```

The following results apply

Comparison	entity	relation	property	ERPCompare	RPECompare	PERCompare
compare(abc,bca)	abc < bca	abc < bca	abc > bca	< 0	< 0	> 0
compare(abc,cba)	abc < cba	abc = cba	abc > cba	< 0	> 0	> 0
compare(cba,caa)	cba = caa	cba > caa	cba = caa	> 0	> 0	> 0

It is **very** important that these comparators work correctly as they determine how trees which use them will be sorted. Make sure to test thoroughly.

Some additional examples are given below.

```
> Record abc = Record.makeRecord("a","b","c");
> Record abb = Record.makeRecord("a","b","b");
> Record bca = Record.makeRecord("b","c","a");
> Record cba = Record.makeRecord("c","b","a");
> Record.ERPCompare.compare(abc,abb)
1
> Record.ERPCompare.compare(abc,abc)
0
> Record.ERPCompare.compare(abc,abb)
1
> Record.RPECompare.compare(abc,abc)
0
> Record.RPECompare.compare(abc,abb)
1
> Record.RPECompare.compare(abc,bca)
-1
> Record.PERCompare.compare(abc,abc)
0
> Record.PERCompare.compare(abc,abb)
1
> Record.PERCompare.compare(abc,bca)
2
> Record.ERPCompare.compare(abc,cba)
-2
// +2 and -2 are not errors.
// Results need not be restricted to -1,0,+1
// so long as they abide by <0, ==0, >0
// for the proper cases. String.compareTo
// may be directly used in the comparators
```

### 3.6 Wildcards and Comparisons

When implementing `query`, one typically wants to find the "first" record that matches a query with a wildcard. This can be done by iterating through the tree from the beginning but such an approach would not meet the target complexity.

Instead, production trees like `TreeSet` provide facilities to obtain sorted views of the tree starting at different elements, for instance starting at the first stored element bigger than a given element. One can use this facility along with properly implemented comparators to accomplish the fast query for this project.

**If a comparator treats wildcards as less than any other string** then asking a properly sorted tree for the first entry bigger than the query will yield the first matching `Record` if one exists. One can then start an in-order traversal to find any other matching `Records` for the query results. This means the comparators need to be aware of the wildcard in `Record` and use it during comparisons.

Consider the tree sorted on `entity, relation, property` and the call `t.query("Lucky", "*", "*")`

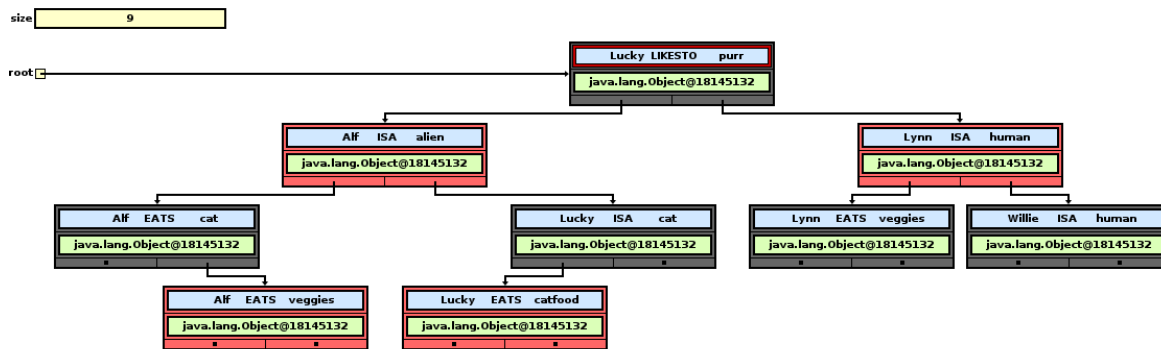


Figure 3: Red-Black Tree with Entity-Relation-Property Sorting

If the wildcards in ("Lucky", "\*", "\*") are considered less than the strings in the relation and property fields of the existing records involving Lucky are

Lucky	EATS	catfood
Lucky	ISA	cat
Lucky	LIKES TO	purr

then the first record "bigger" than the query according to the tree diagram will be Lucky EATS catfood and an in-order traversal from there will find all Lucky records. The query Lucky \* \* would fit "in between" Alf ISA alien and Lucky EATS catfood in the tree above if it were allowed to be added (which it is not).

The following examples illustrate the intention for wildcards in comparisons.

```
> Record r1 = Record.makeRecord("Lucky", "ISA", "cat")
> r1
  Lucky    ISA    cat
> Record r2 = Record.makeRecord("Lucky", "LIKES TO", "purr")
r2
  Lucky    LIKES TO    purr
> Record q = Record.makeQuery("?", "Lucky", "*", "*")
> q
  Lucky    *    *
> Record.ERPCompare.compare(r1, r2)
-3
> Record.ERPCompare.compare(q, r1)
-1
> Record.ERPCompare.compare(q, r2)
-1
> Record.PERCompare.compare(r1, r2)
-13
> Record.PERCompare.compare(q, r1)
-1
> Record.PERCompare.compare(q, r2)
-1
> Record r3 = Record.makeRecord("Alf", "ISA", "alien")
> r3
  Alf      ISA      alien
> Record.ERPCompare.compare(q, r3)
11
> Record r4 = Record.makeRecord("Lucky", "EATS", "catfood")
> r4
  Lucky    EATS    catfood
> Record.ERPCompare.compare(q, r4)
-1
```

Though the situation should never arise in its use of with Triplestore, two wild fields are considered equal to one another regardless of what their underlying string might be.

```
// All wild fields
> Record q = Record.makeQuery("?", "?", "?", "?")
> Record v = Record.makeQuery("?", "?", "?", "?")
> Record.ERPCompare.compare(q, v)
0

// Same wilds except for cat
> v = Record.makeQuery("?", "?", "?", "cat")
> Record.ERPCompare.compare(q, v)
-1
> Record.ERPCompare.compare(v, q)
1

// Two records with wild fields but different wild strings are
// equal to one another
> q = Record.makeQuery("?", "?", "?", "?")
> v = Record.makeQuery("wild", "wild", "wild", "wild")
> Record.ERPCompare.compare(v, q)
```

```
0
> q = Record.makeQuery("","*", "*", "cat")
> v = Record.makeQuery("wild", "wild", "wild", "cat")
> Record.ERPCompare.compare(v,q)
0
```

**Note:** The `Comparator` interface specifies in addition to the `compare()` method that `Comparators` should have an `equals(Object o)` method. All classes inherit `equals()` from `Object` and for this project, there is no need to over-ride the default `equals()` implementation, only that an implementation of `compare(r1,r2)` is provided.

#### 4 TripleStore Class Implementation

```
public class TripleStore
```

TripleStores are databases that have three columns (each row has three fields). Our implementation will enforce that each record in the database is unique. The operations supported are insertion, lookup/query, and removal based on a query. This implementation will support logarithmic time operations for all three operations. To achieve this performance, the database must store its `Records` in three separate balanced binary search trees. A good choice for the BST is `java.util.TreeSet` which is implemented as a Red-Black tree. `TreeSet` provides a variety of useful methods that you should review via its javadocs.

##### 4.1 TripleStore Constructor and Basic methods

```
public TripleStore()
```

Create a `TripleStore` Initialize any trees you will use in the constructor Make sure to pass in relevant arguments to the tree constructors such as `Comparators`.

In addition, specify a few simple methods to get and set the wild card for a `TripleStore`.

```
public String getWild()
public void setWild(String w)
```

- These methods allow the current wild card to be inspected and changed.
- The default wild card is **required** to be the asterisk string `*`
- The following example illustrates the independence of wild cards between separate `TripleStore` instances.

```
TripleStore t1 = new TripleStore();
t1.add("Willie", "ISA", "human");

TripleStore t2 = new TripleStore();
t2.add("Willie", "ISA", "human");
t2.setWild("@");

t1.query("Willie", "*", "*");
// [ Willie    ISA    human]

t2.query("Willie", "@", "@");
// [ Willie    ISA    human]

t1.query("Willie", "@", "@");
// []

t2.query("Willie", "*", "*");
// []
```

##### 4.2 TripleStore.toString()

```
// TARGET COMPLEXITY: O(N)
// N: the number of records stored in the TripleStore
```

Create a string representation of the `TripleStore` suitable for printing. This should be done by building a `String` containing the results of each `Record.toString()` which is currently stored and separating these with a `newline`. Proper implementation of [Record.toString\(\)](#) is essential for `TripleStore.toString()`

Notes

- Records **MUST** appear in sorted order based on a correct implementation of `ERPComparator`. See [Record Comparators](#) for details on this order. *Warning:* Some of the demo displays may not show records exactly in this ordering as the demos were generated using a prototype but it is requirement for your implementation.
- Conversion to a string should not repeatedly call `query` but instead directly walk through the underlying data likely via a traversal of one of the index trees.
- To meet the given complexity bound, you will need to find an efficient way to concatenate `Strings`. The typical use of `+` is **NOT** efficient and will result in a  $O(N^2)$  performance. Your textbook contains information on efficiently building `Strings` up.

##### 4.3 TripleStore.add()

```
// Target Complexity: O(log N)
// N: number of records in the TripleStore
```

Add a single record into the `TripleStore`

- Use the three argument `Strings` to create a valid `Record`
- It is **not** an error to add a `Record` with the current wild card
- If the `TripleStore` already contains an identical record, do not add the new duplicate information and return `false`
- Add the `Record` and ensure that any and all index trees are updated.

Examples from [1.3](#)

```

> TripleStore t = new TripleStore();
> t

> // Empty TripleStore

> // --- ADD ---
> boolean b = t.add("Willie", "ISA", "human");
> b
true
> // Successful add returns true

> t.add("Alf", "ISA", "alien");
> t
      Alf      ISA      alien
      Willie   ISA      human
> // Two Records in the TripleStore

> b = t.add("Willie", "ISA", "human");
> b
false
> // Duplicates are not allowed
> t
      Alf      ISA      alien
      Willie   ISA      human
> // Still only two Records in the TripleStore

> t.add("Lynn", "ISA", "human");
> t.add("Lucky", "ISA", "cat");
> t.add("Alf", "EATS", "cat");
> t.add("Lynn", "EATS", "veggies");
> t.add("Lucky", "LIKESTO", "purr");
> t.add("Lucky", "EATS", "catfood");
> t.add("Alf", "EATS", "veggies");
> t
      Alf      EATS      cat
      Alf      EATS      veggies
      Alf      ISA      alien
      Lucky    EATS      catfood
      Lucky    ISA      cat
      Lucky    LIKESTO   purr
      Lynn     EATS      veggies
      Lynn     ISA      human
      Willie   ISA      human

```

#### 4.4 TripleStore.query()

```

// TARGET COMPLEXITY: O(K + log N)
// K: the number of matching records
// N: the number of records in the triplestore.
public List<Record> query(String entity, String relation, String property)

```

Return a `List` of the records that match the given query parameters. Any of the arguments `entity`, `relation`, `property` may be the wild card for the TripleStore.

- To meet the complexity bound, you will need to identify which index to use for the initial search into the tree.
- Use methods of your tree (probably `TreeSet`) to get a view of the tree starting at a certain key and which allow the tree to be traversed in order. This will allow you to meet the target complexity.
- You'll likely need to use `Record.makeQuery(. . .)` to create a record with the wild fields matching the wild card associated with the triplestore.

Examples from [TripleStoreDemo](#)

```

> // --- QUERY ---
> import java.util.*;
> List<Record> results;
> results = t.query("Alf", "ISA", "alien");
> results
[      Alf      ISA      alien]
> // query returns an List
> // Records match exactly, 1 result

> t.getWild()
""
> results = t.query("Alf", "ISA", "")
[      Alf      ISA      alien]
> // query with a wild card matched 1 record
> results = t.query("Alf", "EATS", "")
[      Alf      EATS      cat,      Alf      EATS      veggies]
> // query with a wild card matched 2 records
> results = t.query("Alf", "", "")
[      Alf      EATS      cat,      Alf      EATS      veggies,      Alf      ISA      alien]
> // query with several wild cards matched 3 records

> t.query("", "ISA", "human")
[      Lynn     ISA      human,      Willie   ISA      human]
> t.query("", "", "human")
[      Lynn     ISA      human,      Willie   ISA      human]
> t.query("", "", "cat")
[      Alf      EATS      cat,      Lucky    ISA      cat]

```

2/5/2017

CS 310 HW 4: TripleStore Database

```
> t.query("","ISA","")
[ Alf ISA alien, Lucky ISA cat, Lynn ISA human, Willie ISA hum
> // wildcards can appear in any of entity,relation,property
```

#### 4.5 TripleStore.remove()

```
// TARGET COMPLEXITY:  $O(K * \log N)$ 
// K: the number of matching records
// N: the number of records in the triplestore.
public int remove(String e, String r, String p){
```

**Note:** `remove()` does not need to be as fast as `query()`: compare their target complexities carefully

- `query()`:  $O(K + \log N)$
- `remove()`:  $O(K \times \log N)$

Remove records from the TripleStore. Any of the arguments `entity`, `relation`, `property` may be wild. All records that match the parameters will be removed.

Return the number of elements removed which may be 0 if no records matched the parameters.

Examples from [TripleStoreDemo](#)

```
> // --- REMOVE ---
> t
  Alf  EATS  cat
  Alf  EATS  veggies
  Alf  ISA   alien
  Lucky EATS  catfood
  Lucky ISA   cat
  Lucky LIKESTO purr
  Lynn  EATS  veggies
  Lynn  ISA   human
  Willie ISA   human

> int nrm = t.remove("Alf","EATS","veggies")
1
> // successful removal, changes database
> t
  Alf  EATS  cat
  Alf  ISA   alien
  Lucky EATS  catfood
  Lucky ISA   cat
  Lucky LIKESTO purr
  Lynn  EATS  veggies
  Lynn  ISA   human
  Willie ISA   human

> nrm = t.remove("Alf","EATS","fruit")
0
> // unsuccessful remove
> nrm = t.remove("Alf","EATS","veggies")
0
> // record no longer exists

> nrm = t.remove("Alf","*","*")
2
> t
  Lucky  EATS  catfood
  Lucky  ISA   cat
  Lucky  LIKESTO purr
  Lynn   EATS  veggies
  Lynn   ISA   human
  Willie ISA   human

> t.remove("","*","*")
6
> t

> // wild cards can remove many records
```

## 5 Grading

Grading for this HW will be divided into three distinct parts:

- Part of your grade will be based on passing some automated test cases by an early "milestone" deadline. See the top of the HW specification for
- Part of your grade will be based on passing all automated test cases by the final deadline
- Part of your grade will be based on a manual inspection of your code and analysis documents by the teaching staff to determine quality and efficiency

### 5.1 Final Automated Tests (50%)

- JUnit test cases will be provided to detect errors in your code. These will be run by a grader on submitted HW after the final deadline
- Tests may not be available on initial release of the HW but will be posted at a later time
- Tests may be expanded as the HW deadline approaches.
- **It is your responsibility to get and use the freshest set of tests available**
- Tests will be provided in source form so that you will know what tests are doing and where you are failing.
- It is up to you to run the tests to determine whether you are passing or not. If your code fails to compile against the tests, little credit will be garnered for this section
- Most of the credit will be divided evenly among the tests; e.g. 50% / 25 tests = 2% per test. However, the teaching staff reserves the right to adjust the weight of test cases after the fact if deemed necessary.
- Test cases are typically run from the command line using the following invocation which you should verify works as expected on your own code.

file:///home/kauffman/teaching/cs310-F2016/hw/hw4-triplestore/hw4.html

14/16

```

UNIX Command line instructions
Compile
> javac -cp ../junit-cs310.jar *.java

Run tests
> java -cp ../junit-cs310.jar SomeTests

WINDOWS Command line instructions: replace colon with semicolon
Compile
> javac -cp ../junit-cs310.jar *.java

Run tests
> java -cp ../junit-cs310.jar SomeTests

```

### 5.2 Final Manual Inspection (50%)

- Graders will manually inspect your code and analysis documents looking for a specific set of features after the final deadline
- Most of the time the requirements for credit will be posted along with the assignment though these may be revised as the HW deadline approaches.
- Credit will be awarded for good coding style which includes
  - Good indentation and curly brace placement
  - Comments describing private internal fields
  - Comments describing a complex section of code and invariants which must be maintained for classes
  - Use of internal private methods to decompose the problem beyond what is required in the spec
- Some credit will be awarded for clearly adhering to the target complexity bounds specified in certain methods. If the specification lists the target complexity of a method as  $O(N)$  but your implementation complexity is actually  $O(N \log N)$ , credit will be deducted. If the implementation complexity is too difficult to determine due to poor coding style, credit will likely be deducted.

All TARGET COMPLEXITIES are worst-case run-times.

- Some credit will be awarded for turning in any analysis documents that are required by the HW specification. These typically involve analyzing how fast a method should run or how much memory a method requires and are reported in a text document submitted with your code.

## 6 Final Manual Inspection Criteria

### 6.1 (10%) Record Design

- The design of the `Record` class is documented via comments on class fields.
- It is clear how the notion of wild cards is supported.
- It is clear that any string can be used to create a query with wild fields and that there are no external dependencies of wildness on other classes such as `TripleStore`
- There is adequate documentation of the `matches(...)` method so that it is easy to understand how it operates.

### 6.2 (10%) Record Comparator Design and Use

- The static fields housing the comparators are present and usable
- It is clear how each comparator implements `compare(...)` differently to sort `Records` in alternative orders. Some comments and clear code are present achieve the different sorting orders.
- When creating trees that store `Records` in different orders, `TripleStore` adheres to the public interface of `Record` laid out in the HW specification. `TripleStore` should not use any classes that are internal to `Record` except via public fields mentioned in the HW specification.

### 6.3 (20%) TripleStore Method Runtime Complexities

```

// Target Complexity: O(log N)
// N: number of records in the TripleStore
public boolean add(String entity, String relation, String property)

```

- It is clear that no duplication of records occurs during add.
- Data is added to all internal trees to support fast queries later.
- The overall method clearly meets the target runtime complexity.

```

// TARGET COMPLEXITY: O(K + log N)
// K: the number of matching records
// N: the number of records in the triplestore.
public List<Record> query(String entity, String relation, String property)

```

- Code that which does tree selection is present and clear. This code decides among ERP, RPE, or PER, which tree will efficiently answer a query with the given wild card pattern and selects it.
- Methods of `TreeSet` are used effectively to meet the target runtime complexity for `query()`.
- Iterators are employed to avoid repeatedly searching through the tree and instead perform an in-order traversal to visit appropriate elements.
- The overall method clearly meets the target runtime complexity.

```

// TARGET COMPLEXITY: O(K * log N)
// K: the number of matching records
// N: the number of records in the triplestore.
public int remove(String e, String r, String p)

```

- The records that must be removed are determined efficiently potentially by using other methods of `TripleStore`
- Data is removed from all internal trees to support fast queries later
- The overall method clearly meets the target runtime complexity.

```

// TARGET COMPLEXITY: O(N)
// N: the number of records stored in the TripleStore
public String toString()

```

- Make sure to use an efficient means to construct the string representation of the Triple Store

#### 6.4 (5%) Coding Style and Readability

This is a larger project. It will require discipline and effort to keep track of how all the pieces fit together. Commenting your own code to keep track of the purpose of fields and methods will pay much higher dividends in this project than was previously the case. Your code will be inspected for clarity in the following categories.

- **Code Cleanliness (1%)** Indent and `{}` code uniformly throughout the program to improve readability.
- **Class Documentation (1%)** Each class has an initial comment indicating its intended purpose, how it works, and how it relates or uses other classes in the project.
- **Field Documentation (1%)** Each field of a class is documented to indicate what piece of data is tracked and how it will be used, regardless of visibility.
- **Method documentation (1%)** Each method has a short description indicating its intended purpose and how it gets its job done. (These are also needed for your own helper methods you choose to add).
- **Proper Use of Generics (1%)** The point of using Java generics is to get good compile-time checks and avoid runtime casting. Effective use of generics will mean very few runtime casts will be needed. To that end, runtime casts will be penalized unless there is no other way around them such as in the implementation of `ArrayList`. Unless otherwise noted, all classes required for HW do not require runtime casts.

#### 6.5 (5%) Correct Project Setup

Correctly setting up the directory structure for each project greatly eases the task of grading lots of projects. Graders get cranky when strange directory structures or missing files are around and you do not want cranky graders. The following will be checked for setup on this project.

- The *Setup* instructions were followed closely
- The *Project Directory* is named according to the specification
- There is an *Identity Text File* present with the required information in it
- Code can be compiled and tests run from the command line

### 7 Setup and Submission

#### 7.1 HW Directory

There is no code distribution for this assignment

Create a directory named *masonid-hwX* where *masonid* is your mason ID. My mason ID is `ckauffm2` so I would create the directory `ckauffm2-hw4`

This is your **HW directory**. Everything concerning your assignment will go in this directory.

#### 7.2 ID.txt

Create a text file in your HW directory called `ID.txt` which has identifying information in it. My `ID.txt` looks like.

```
Chris Kauffman
ckauffm2
G001234567
```

It contains my full name, my mason ID, and G# in it. The presence of a correct `ID.txt` helps *immensely* when grading lots of assignments.

#### 7.3 Penalties

Make sure to

- Set up your HW directory correctly
- Include an `ID.txt`
- Indent your code and make comments

Failure to do so may be penalized by a 5% deduction.

#### 7.4 Submission: Blackboard

**Do not e-mail the professor or TAs your code.**

Create a **ZIP file** of your HW directory and submit it to the course blackboard page. **Do not submit multiple files manually through blackboard** as this makes it hard to unpack large numbers of assignments. Learn how to create a zip and submit only that file

On Blackboard

- Click on the *Assignments* section
- Click on the HW1 link
- Scroll down to "Attach a File"
- Click "Browse My Computer"
- Select your Zip file

You can resubmit to blackboard as many times as you like up to the deadline

Author: Chris Kauffman and Richard Carver ([kauffman@cs.gmu.edu](mailto:kauffman@cs.gmu.edu))  
Date: 2016-12-01 Thu 16:35

Last Updated: 2016-02-17 Wed 23:30

## CS 211 Lab 5: Output, Inheritance, Censorship

- **Due: 11:59pm Sunday 2/21/2015**
- Approximately 1% of total grade
- Submit to [Blackboard](#)
- [Lab Exercises are open resource/open collaboration](#)
- Sections marked (TESTED) involve functionality that is tested by the automatic test cases.

CODE DISTRIBUTION: [distrib-lab05.zip](#)

CHANGELOG Empty

### Table of Contents

- [1. Rationale](#)
- [2. PrintWriter class](#)
- [3. CensoredWriter Overview](#)
- [4. CensoredWriter Class](#)
- [5. CensoredWriter Constructors \(TESTED\)](#)
- [6. CensoredWriter.transform\(\)\(TESTED\)](#)
- [7. CensoredWriter Printing functions \(TESTED\)](#)
- [8. Notes](#)
- [9. Getting Credit for this Lab](#)
- [10. Testing Locally Using JUnit](#)

### 1 Rationale

Writing output to files is an essential skill in any computing environment. This exercise will familiarize you with `PrintWriter` which is one means to write to files and to the terminal.

Very occasionally one must write a new inheritance hierarchy from scratch, but much more frequently one must adapt an existing class to suit the situation. In java this is done via inheritance and overriding certain methods of the parent class. While doing so, all capabilities of the parent class are still available.

Completing this exercise will familiarize you with file output using `PrintWriter` and how to inherit from `PrintWriter` to affect how output is produced.

#### Associated Reading

##### Building Java Programs

Chapter 6.4 covers file output using the `PrintStream` class, an alternative to the `PrintWriter` that will be used here. Everything described there about `PrintStream` applies equally to `PrintWriter` here.

Chapter 9 discusses inheritance. Of particular relevance are overriding parent class methods but invoking them during the child method using the `super` keyword. You will also want read the discussion on calling parent class constructors using `super`.

##### CS 211 Lab Manual

Chapter 6 covers basic use of `PrintWriter` for output to files as well as a few other classes of note for I/O (`ObjectOutputStream`/`ObjectInputStream`) are particularly cool but not necessary for the lab).

Chapter 7 covers inheritance and how to construct your own hierarchies. The discussion of overriding methods is useful for this lab.

##### PrintWriter versus PrintStream

The observant programmer will recall that `System.out` is a `PrintStream` discussed in the textbook above, while we will focus on its close relative `PrintWriter`. The difference between these two is quite subtle and the [subject of discussion on a StackOverflow thread](#). Curious students may be interested in perusing this thread but it is optional reading.

#### Videos

[Videos linked here](#) provide an overview of inheritance along with demonstrations and examples. Review these if you struggle to follow the discussion below.

### 2 PrintWriter class

Java provides output facilities through various classes. You are likely very familiar with `System.out` by now as using its output methods `print/println/printf` have been the primary means of putting text on the screen. Inspecting the [System class](#) you will see that `out` is a `static` field and is an instance of the [PrintStream class](#) which endows it with many methods such as `print/println/printf`.

There come times when one must do output to files rather than the terminal. For this task, we will use the [java.io.PrintWriter](#) class. It is quite similar to `System.out` in that `PrintWriters` have `print/println/printf` methods. However, when constructing a `PrintWriter`, the constructor determines the destination of output. Should a `File` be passed in, all output is directed the file. This is demonstrated in the provided `PrintWriterDemo.java` file.

```
// Demonstrate the PrintWriter class for doing output to arbitrary
// locations. The most common use is to print to files using either
// new PrintStream("filename.txt")
// or new PrintStream(new File("filename.txt"))
//
// However, one can also print to standard output (terminal) using the
// constructor
// new PrintStream(System.out)
//
// Just make sure not to close standard output.
```



```
import java.io.*;
public class PrintWriterDemo {
    public static void main(String args[])
        throws FileNotFoundException // WTF?
    {
        PrintWriter out = new PrintWriter(new File("myfile.txt"));
        // PrintWriter out = new PrintWriter("myfile.txt"); // Alternative constructor that also writes to a file
        // PrintWriter out = new PrintWriter(System.out); // Write to the screen instead
        out.println("Sweet foutput");
        out.printf("An int: %d\nA double %.1f\nA string: %s\n",
            1, 2.5, "Three");
        out.close(); // May close System.out (bad)
    }
}
```

This program writes some simple output to the file `myfile.txt` and after the program executes, there will be such a file in the directory the program was run in.

Alternatively, output can be directed to the screen instead if the `PrintWriter` is constructed with `System.out`. This is analogous to `Scanners` which can read from `System.in` or from a `File` or from a `String`. The ability to choose between output destinations is demonstrated in the `PickyOutput.java` file which asks the user where they would like to write and creates an appropriate `PrintWriter`.

```
// Ask the user whether to print stuff to the screen or to a file.
// Create a new PrintWriter using different constructors, System.out
// or String filename.

import java.io.*;
import java.util.Scanner;

public class PickyOutput {
    public static void printStuff(PrintWriter out){
        out.println("Sweet output");
        out.printf("An int: %d\nA double %.1f\nA string: %s\n",
            1, 2.5, "Three");
    }

    public static void main(String args[])
        throws FileNotFoundException // WTF?
    {
        Scanner in = new Scanner(System.in);
        System.out.print("Where do you want to print? ");
        String filename = in.next();
        PrintWriter pw;
        if(filename.equals("stdout")){
            pw = new PrintWriter(System.out);
        }
        else{
            pw = new PrintWriter(filename);
        }
        printStuff(pw);
        pw.close(); // could close System.out
        System.out.println("Done with output"); // This will not print if using System.out!!
    }
}
```

**Note:** `PrintWriters` do not always push their output to the destination immediately. This requires

- When printing to a file call `close()` to ensure all output is pushed to the file
- When printing to the screen, one may need to call `flush()` to get text to appear. In subclasses, `flush()` can be added to `print/println` to do this.
- Make sure **NOT** to call `close()` on a `PrintWriter` that is writing to the screen: this will close `System.out` and prevent ALL further output from being printed to the screen.

### 3 CensoredWriter Overview

Occasionally in history there has arisen the need to *censor* communications. An automatic means of doing this is to scan output and block or replace certain key words that appear. We will set up a class `CensoredWriter` which performs this task: any output that contains a specified string pattern will have the pattern replaced by a censor string. The solution will involve

- Making `CensoredWriter` a child class of `PrintWriter`
- `CensoredWriter` will then have `print` and `println` methods
- `CensoredWriter` will change what the `print` and `println` methods do: it will transform input to them.
- Transformations will be done in a special `transform()` method of the `CensoredWriter`. This is a method that the parent class `PrintWriter` does not have.
- Transformations of output will utilize `String` class methods which will replace matched string patterns with the censored text
- Transformed text will be passed to the parent class `print` and `println` methods which will perform the printing.
- Several constructors for `CensoredWriter` will be provided so that output can be sent to the screen or to a file.
- All other methods of `CensoredWriter` will be inherited from `PrintWriter` and will behave identically to the parent class.

The intended use for `CensoredWriter` is demonstrated below in an interactive session.

```
Welcome to DrJava. Working directory is /lab03
> CensoredWriter out = new CensoredWriter(System.out, "frack");
```

2/5/2017

CS 211 Lab 5: Output, Inheritance, Censorship

```

> out.println("When looking for fossil fuels, these days it is efficient to frack for natural gas")
When looking for fossil fuels, these days it is efficient to %!^*#@ for natural gas.

> out.println("Wait, why can't I say frack now?")
Wait, why can't I say %!^*#@ now?

> out.println("What the frack!?!")
What the %!^*#@!?!

> out.println("Hey, frack you, you stupid fracking censor. What about free fracking speech?")
Hey, %!^*#@ you, you stupid %!^*#@ing censor. What about free %!^*#@ing speech?

> out.println("Poop. Didn't see that one coming did you?");
Poop. Didn't see that one coming did you?

> out = new CensoredWriter(System.out, "poop");
> out.println("Big steaming piles of poop on you.");
Big steaming piles of %!^*#@ on you.
> out.println("Frack. :-p");
Frack. :-p

> out = new CensoredWriter(System.out, "[Pp]oop|[Ff]rack");
> out.println("What's the matter? Can't you censor frack and poop at the same time?")
What's the matter? Can't you censor %!^*#@ and %!^*#@ at the same time?

> out.println("Fracking regular expressions")
%!^*#@ing regular expressions

```

The last few lines demonstrate that the censored string is actually a *regular expression* allowing for a variety of patterns to be captured. **You do not need to know anything about regular expressions** to complete this lab. Instead, replacements will be done using `String` methods which will handle regular expressions without any effort on your part. We may discuss RegExs later in the course as they are very useful as evidenced by the above ability to replace multiple matching words. Investigate on your own if you are curious.

It is a simple matter to redirect output for a `CensoredWriter` to a file as well. Here is an example.

```

// Writing to a file
> out = new CensoredWriter("somefile.txt", "frack");
> out.println("When looking for fossil fuels, these days it is efficient to frack for natural gas");
> out.close();
// somefile.txt now contains the censored text

```

## 4 CensoredWriter Class

```
public class CensoredWriter extends PrintWriter
```

A class which acts like a `PrintWriter` but filters output of certain string patterns. It has several required elements.

- Constructors to create `CensoredWriter`s that write to `System.out` and to files. These constructors also accept a string pattern to censor
- A `transform()` method which will transform input strings for output. This method is `public` to allow for testing but is primarily used internally.
- Overrides of `print` and `println` methods that `transform()` input text before calling the `super` class versions of these methods.

### Required Fields

None, but you will likely need two internal fields to track (1) the string pattern to be censored and (2) the string to replace censored text by.

The expected replacement string is

```
%!^*#@
```

and all censored text is replaced with that string.

### Class Structure

The rough structure of the `CensoredWriter` class is given below.

## 5 CensoredWriter Constructors (TESTED)

### Terminal Output Constructor

```
public CensoredWriter(OutputStream o, String c)
```

Construct a `CensoredWriter` which writes to the given output stream `o` and censors the pattern in `c`. The censored writer will then write its output to this output stream and all instances of the pattern `c` in output will be replaced by the replacement string (above). Several things are worth noting here.

- `System.out` is of type `PrintStream` which is a descendant of `OutputStream`. So this constructor will make it possible to create a `CensoredWriter` which writes censored text to the terminal. This will look like

```
CensoredWriter out = new CensoredWriter(System.out, "frack");
out.println("Some text which contains frack and frack.");
```

- The `PrintWriter` class from which `CensoredWriter` descends already has a constructor which takes an output stream. This constructor can be accessed using the `super` keyword.
- If you call a super class constructor in a child class constructor, **Java has an arbitrary rule** that it must be the first thing that the child class constructor does, so the first line of your constructor will likely be `super(...)`;
- This constructor and the other two constructors **should be very short**, on the order of 1-2 lines.

### File Output Constructors

```
public CensoredWriter(File f, String c) throws Exception{
    public CensoredWriter(String s, String c) throws Exception{
```

Construct a `CensoredWriter` which writes to a file which is the argument in the first case or is named in the second case. Calls to `print` and `println` for the `CensoredWriter` will then print to a file. The contents will be flushed to file when the `close()` method is called. Again, the second argument is the string pattern to censor by replacing it with the replacement string.

Note again that `PrintWriter` has similar constructors to the above so these constructors are very simple to write by invoking the parent constructing using `super`.

### 6 CensoredWriter.transform() (TESTED)

```
public String transform(String s)
```

This method is used mainly to transform desired output to remove any censored patterns. It is public to allow for easy testing. The input string `s` is transformed so that all instances of the censored pattern should be removed. The resulting string is returned.

Here is an example use in DrJava. Note that `transform()` returns a string which is shown with double quotes around it in the interactive loop.

```
> CensoredWriter out = new CensoredWriter(System.out, "poop");
> String s = out.transform("Everybody poops");
> s
"Everybody %!^*#@s"
> out.transform("Everybody poops")
"Everybody %!^*#@s"
out.transform("I'd like to see the poop deck. Can you arrange a poop deck tour?")
"I'd like to see the %!^*#@ deck. Can you arrange a %!^*#@ deck tour?"
```

#### Hints

- Browse the [String](#) class methods to find a method that will replace all instances of some pattern string with another string.
- Using the `String` methods should make this method short, 1-2 lines long.

### 7 CensoredWriter Printing functions (TESTED)

```
public void print(String s)
public void println(String s)
```

The `CensoredWriter` prints things differently than its parent `PrintWriter` so it must **override** the parents existing `print` and `println` methods.

- Rather than printing the raw string that is passed as an argument, `CensoredWriter` should `transform()` the string first.
- The modified string is then passed to the parent class's `print` method. This will require use of the `super` keyword to do things as the parent does.

#### Examples

```
> CensoredWriter cw = new CensoredWriter(System.out, "whine|whining");
> cw.println("No whining")
No %!^*#@
> cw.println("Only wimps whine when the assignment is hard so stop whining and start coding")
Only wimps %!^*#@ when the assignment is hard so stop %!^*#@ and start coding
> cw.transform("Only wimps %!^*#@ when the assignment is hard so stop %!^*#@ and start coding")
"Only wimps %!^*#@ when the assignment is hard so stop %!^*#@ and start coding"
> CensoredWriter cw = new CensoredWriter("somefile.txt", "whine|whining");
> cw.println("Only wimps whine when the assignment is hard so stop whining and start coding")
> cw.close();
```

### 8 Notes

- When printing to the screen, one may need to call `flush()` to get text to appear. In subclasses, `flush()` can be added to `print/println` to do this.
- The code for `CensoredWriter` is very short. The reference implementation is 31 lines long which includes closing curly braces. If you find you are writing lots of code, you are doing something wrong.

- Make sure to do a little research on the `super` keyword. It is used to access the parent or super class, in this case `PrintWriter`. One can call constructors for the parent class using `super(constructor, args, here)` and invoke methods of the super class using `super.someMethod(with, some, args);`

## 9 Getting Credit for this Lab

**To receive credit for this lab you must submit a zip file of your files to Blackboard.** Credit will be assigned based on the fraction of tests you pass. You may test your programs locally as well to ensure you know what your score will be ahead of time. Most students work on labs until they pass all tests then submit their code.

**Grading Process:** Graders will download your zip file, unzip it, change into the resulting directory and run a series of commands to compile and test your code. The exact commands are given in the below section. You can verify what your lab score will be by running those commands yourself. This is particularly relevant if you are using an IDE other than Dr. Java: **it is your responsibility to verify that the tests pass on the command line.** Test failures during grading will receive no credit and will not be examined for regrading.

After verifying that your code runs, zip your lab directory and submit it to Blackboard under the corresponding lab submission location.

## 10 Testing Locally Using JUnit

Review Lab 01 for details of how to run tests locally as graders will use test cases as a way to determine the correctness of your code. As an example, code passing all tests will compile and produce test output which looks like this.

```
aphaedrus [ckauffm2-216-lab05]% javac -cp junit-cs211.jar:. *.java
aphaedrus [ckauffm2-216-lab05]% java -cp junit-cs211.jar:. Lab05Tests
JUnit version 4.12
.....
Time: 0.058

OK (6 tests)
```

**Note:** The test cases for Lab 5 create some test output files which follow the name pattern `testfile_X.txt`. These are only used during testing and may be safely deleted if desired. They are regenerated each time the tests are run.

---

Author: Mark Snyder, Chris Kauffman ([msnyde14@gmu.edu](mailto:msnyde14@gmu.edu), [kauffman@cs.gmu.edu](mailto:kauffman@cs.gmu.edu))  
Date: 2016-02-17 Wed 23:30

Name: \_\_\_\_\_

NetID: \_\_\_\_\_

G#: \_\_\_\_\_

**CS 100: Mini-Exam 4**

Fall 2015

George Mason University

Exam period: 30 minutes

Points available: 40

Weight: 7.5% of final grade

**Problem 1 (5 pts):** Sam is writing a program that will allow users to play the card game Solitaire. He is frustrated though that each time he runs his program, he always gets the same sequence of cards during the game. Describe why this is probably happening.

**Problem 2 (5 pts):** Suggest a way that Sam could “fix” his problem such that when he starts his Solitaire program right now he gets one sequence of cards and when he plays the game later or tomorrow he gets a different sequence of cards.

**Problem 3 (5 pts):** Describe a computer simulation that is used to make decisions that impact the lives of a large number of people.

**Problem 4 (5 pts):** List two types of private digital information which private companies routinely gather on customers. The information should not be something customers volunteer such as their name or email address but is instead information which the companies gather by observing customer activity. Explain what motivates companies to gather this information and how it is typically used.

NetID:

---

**Problem 5 (5 pts):** Describe a reasonably complex task which was historically performed by humans but which nowadays is handled via a **machine learning or statistical** algorithm in place of human intelligence.

**Problem 6 (5 pts):** Using 1 computer processor it takes 4 minutes to finish sorting a long list of numbers. **True or False:** If 4 computer processors are used to sort the numbers, it should only take 1 minute to finish sorting. **Justify your answer.**

**Problem 7 (10 pts):** **Argue for or against** the statement below. Use examples we discussed in class to support your argument.

Statement: *Government can easily increase both security and privacy at the same time.*

Name: \_\_\_\_\_

NetID: \_\_\_\_\_

G#: \_\_\_\_\_

**CS 499 Parallel Computing: Mini-Exam 1**

Spring 2016

George Mason University

Exam period: 30 minutes

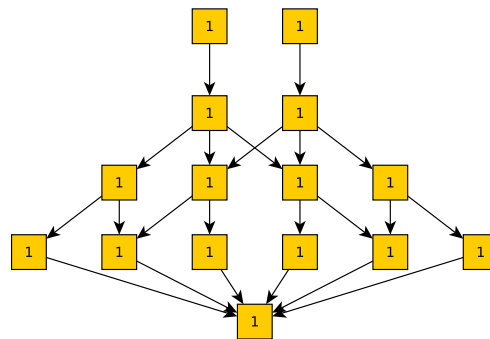
Points available: 40

Weight: 6.25% of final grade

**Problem 1 (10 pts):** We have spent considerable time examining the 2D mesh and hypercube network topologies for parallel computers. Contrast these two and describe the tradeoffs of using one versus the other. Give at least one concrete difference of a type of parallel operation/group communication for which the two networks have different performance.

**Problem 2 (10 pts):** Compute the following statistics for given task dependency graph.

- \_\_\_\_\_ Maximum degree of concurrency
- \_\_\_\_\_ Critical path length
- \_\_\_\_\_ Assuming infinite available processors, maximum achievable **speedup** over one processor
- \_\_\_\_\_ The minimum number of processes needed to obtain the maximum possible speedup
- \_\_\_\_\_ Maximum achievable speedup for 2 processors
- \_\_\_\_\_ Maximum achievable speedup for 4 processors



---

 NetID:
 

---

**Background:** The maximum sub-array problem seeks a contiguous chunk of an input array which sums to a larger number than any other sub-array. Some of elements of this array may be negative. Answers include the start and end indices of the maximal sub-array along with its sum. One solution to the maximum sub-array problem is given in pseudocode nearby.

```
function maxSubArray(int A[]){
    bestSum=0, bestStart=-1, bestEnd=-1
    for start=0 to (A.length-1) {
        curSum = 0
        for end=start to A.length {
            curSum += A[end]
            if(curSum > bestSum){
                bestSum=curSum, bestStart=start, bestEnd=end
            }
        }
    }
    return bestSum,bestStart,bestEnd
}
```

**Problem 3 (15 pts):** Describe a **distributed memory** parallel version of the `maxSubArray` function above. Include the following elements along with justifications for each.

- What data is communicated between the  $P$  cooperating processors and when does it transmitted?
- What tasks will each processor perform? Which loops in the provided code are parallelized?
- Is the workload balanced between the processors in your approach?
- Where will the final answer be stored at completion of the function?

**Problem 4 (5 pts):** Describe what the following fragment of MPI code is doing. Identify any problems you see with the fragment, particularly problems that would prevent the program from finishing for certain numbers of processors running the code.

```
int a[10], b[10], proc_id, total_procs;
MPI_Comm_rank(MPI_COMM_WORLD, &proc_id);
MPI_Comm_size(MPI_COMM_WORLD, &total_procs);
...
if (proc_id % 2 == 0) {
    MPI_Send(a, 10, MPI_INT, proc_id+1, 1, MPI_COMM_WORLD);
}
else {
    MPI_Recv(b, 10, MPI_INT, proc_id-1, 1, MPI_COMM_WORLD
            MPI_STATUS_IGNORE);
}
```



### CS 100: Final Customer Satisfaction Survey

**Anonymous** feedback survey: indicate below how the course went for you.

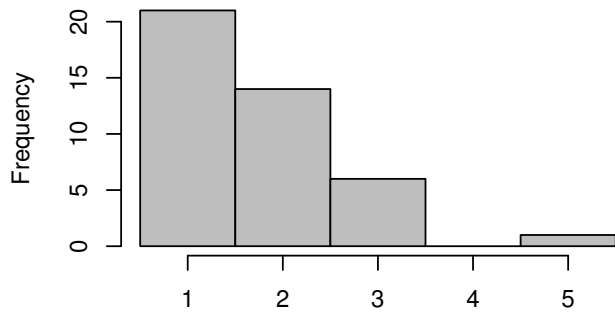
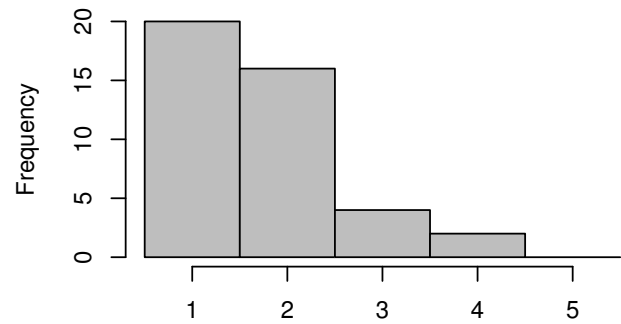
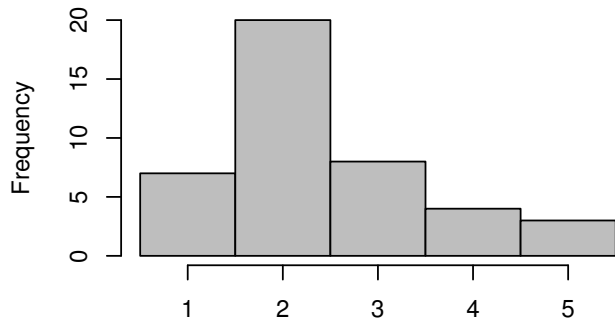
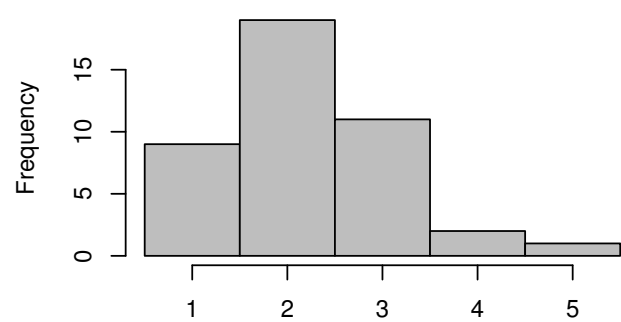
	Epic Win	1	2	3	4	5	Fail!
1.	The grading for the HW was fair.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	I lost credit on HW and feel robbed.
2.	After this class, I have a good idea about what constitutes an algorithm and what goes into solving a problem with a computer.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Computers and how they work are as mysterious as ever to me.
3.	The homeworks were fun and educational.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	The homeworks were soul-crushing labors which taught me nothing.
4.	I learned a ton about how computers work and got \$1,344 of value from this course (3 credits, in-state tuition)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	OMG, that's how much it cost!?! Refund please!

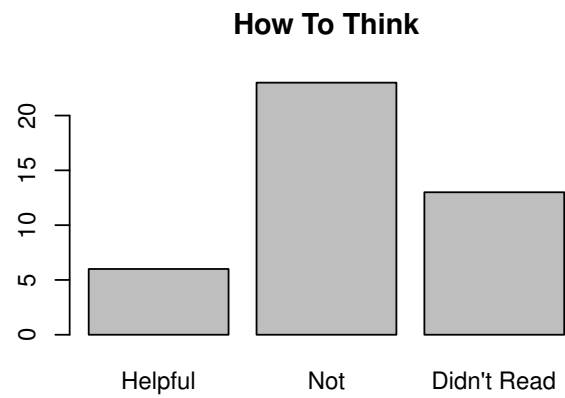
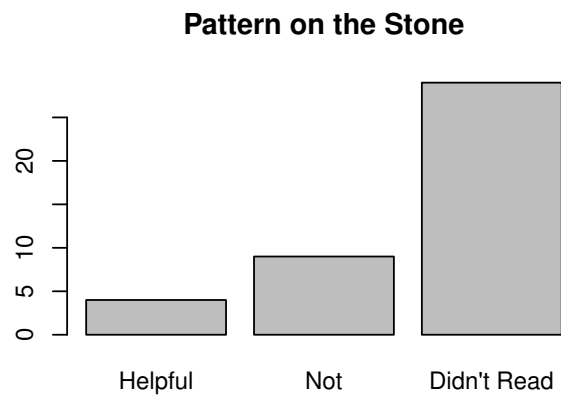
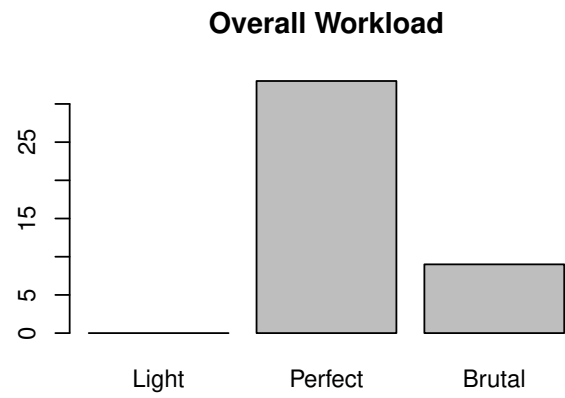
Check the response most accurately reflecting your experience.

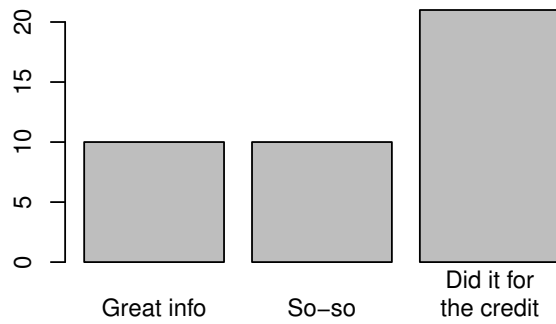
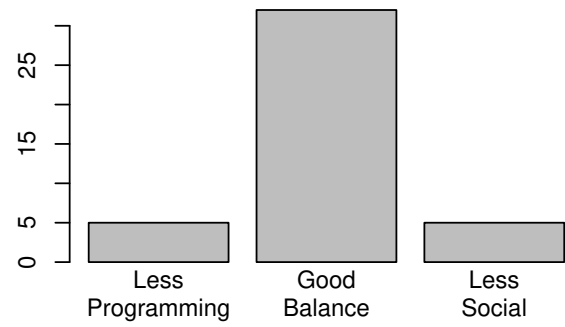
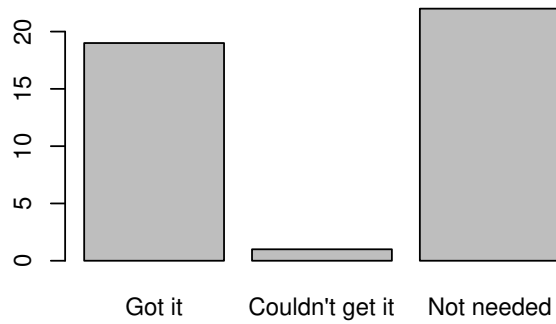
5.	<input type="checkbox"/> We covered material too slow	<input type="checkbox"/> Our speed was Goldilocks: just right!	<input type="checkbox"/> OMG, we went so fast I have whiplash
6.	<input type="checkbox"/> The overall workload for the course was super light	<input type="checkbox"/> The workload felt just right for a 3-credit course	<input type="checkbox"/> You're seriously trying to kill me with this amount of work. Seriously.
7.	<input type="checkbox"/> I read <i>The Pattern on the Stone</i> and it's great	<input type="checkbox"/> I read <i>Pattern</i> but didn't find it helpful for the class	<input type="checkbox"/> I didn't read <i>Pattern</i>
8.	<input type="checkbox"/> Reading <i>How to Think Like a Computer Scientist</i> was helpful for learning about Python	<input type="checkbox"/> We should have read/done something else to assist in learning about Python	<input type="checkbox"/> I didn't do much in the way of studying how to program in Python
9.	<input type="checkbox"/> Reading <i>Zyante: Computing Technology for all</i> provided good additional info on computing for me	<input type="checkbox"/> <i>Zyante</i> occasionally had interesting tidbits but not too much info	<input type="checkbox"/> I only read <i>Zyante</i> because I had to for credit
10.	<input type="checkbox"/> We spent too much time during the semester on programming and algorithms	<input type="checkbox"/> We had a great balance between programming and social impact of computation	<input type="checkbox"/> Forget the social impact stuff and let's do more programming problems
11.	<input type="checkbox"/> I needed help and I got it through, office hours, and/or the discussion board	<input type="checkbox"/> I needed help but Chris and the TAs never seemed available	<input type="checkbox"/> I didn't need much help for the class

In the future, *always* do this in CS 100:

In the future, *never* do this again in CS 100:

**HW grading fair vs Not!****Good overview of CS vs Not!****HW fun and informative vs Not!****Got tuition's worth vs No Way!**



**Zyante Computing for All****Programming vs Social Impact****Outside help**

## CS 100: Final Freeform Feedback

Fall 2015

**In the future, ALWAYS do this in CS 100:**

Count	Comment
<b>Fall 2015</b> (43 responses)	
14	Great lecture style, examples, activities, current events
7	Bonus cards
6	Python programming was great
5	Maintain personal humor, energy, approachability, awesomeness
3	Encryption and security
3	Code.org coding blocks
2	Videos in class
2	Social impact stuff
2	Open-resource exams
	Discussion board
	HTML assignment and posting web site
	Mini-exams vs big exams
	Term paper was a great way to codify what I learned
<b>Fall 2014</b> (31 responses)	
5	In-class exercises to practice and build understanding, very helpful
3	More of programming next time
3	Comedic videos on social issues and visuals
2	Good homework assignments
2	Use Code.org to introduce programming
	Fair grading erring on the easy side
	Good organization
	Exam review was helpful
	Use a real programming language like Python
	Keep the humor in class
	General overview of computing in life was helpful

### In the future, NEVER do this again in CS 100:

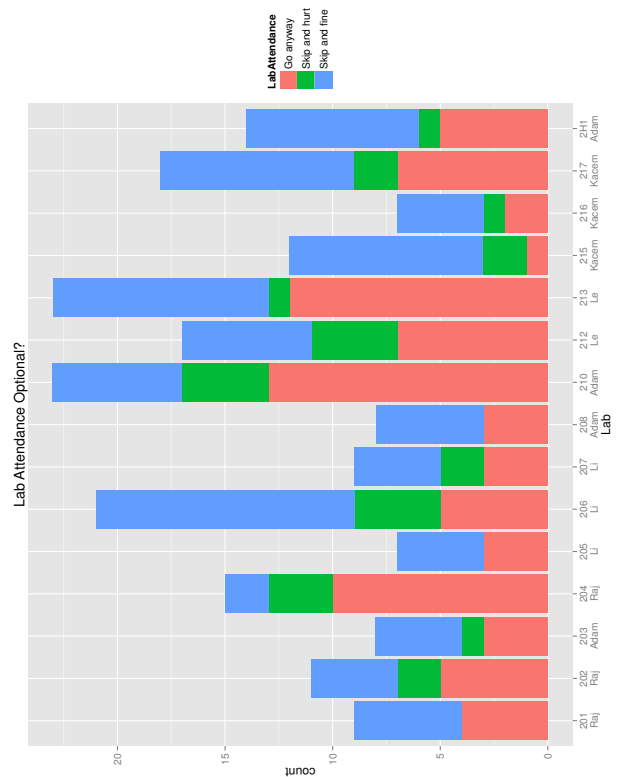
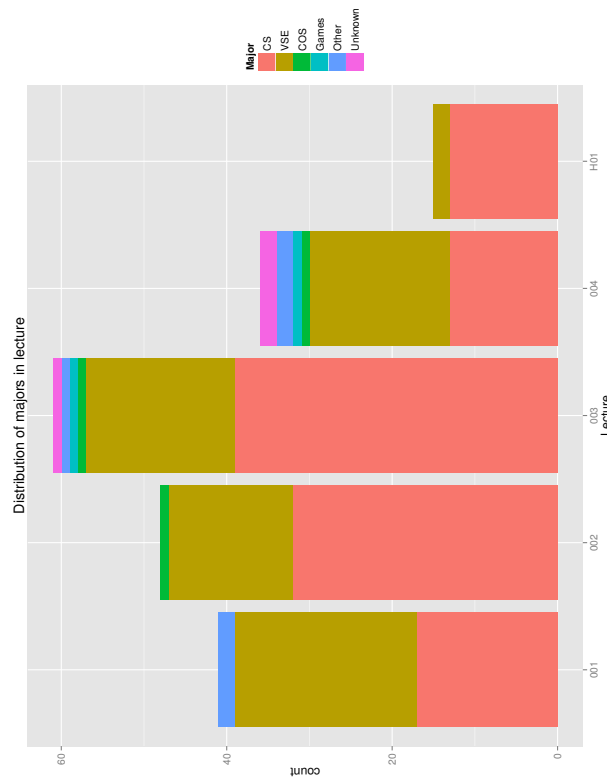
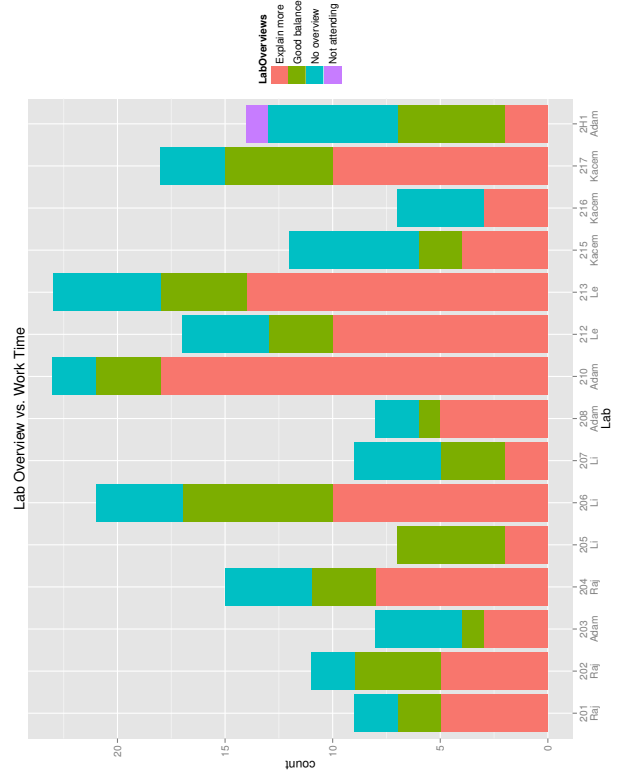
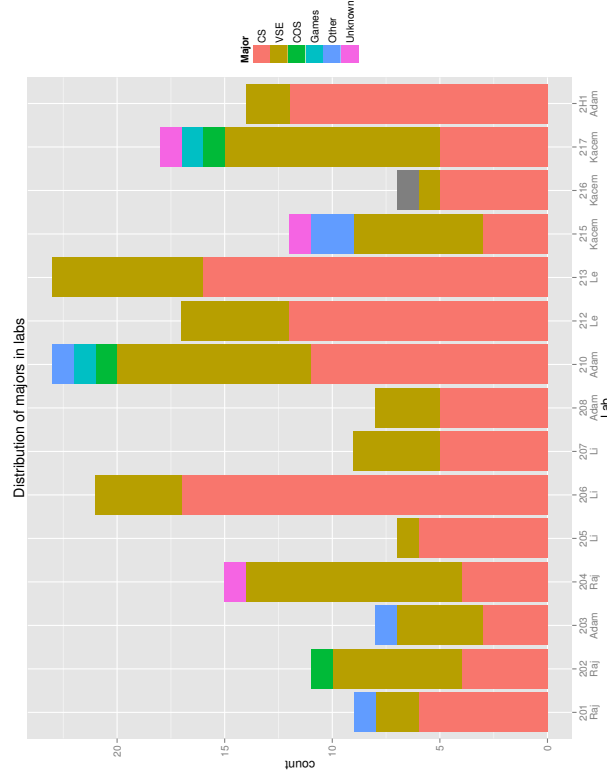
Count	Comment
<b>Fall 2015</b> (43 responses)	
7	Zyante was lame
7	Term paper too demanding
4	Python too hard, not useful
3	Term paper plus final exam too much
3	Python assignment on lists
2	Introduce the term paper earlier, too much due in last week
2	HTML assignment: too hard to post web site
2	Too many books required/suggested
	Mini-exams were tough
	Class ran late
	Group project rather than term paper
	Pattern on the Stone was lame
	Lecture examples not sufficient for HW, needed office hours
<b>Fall 2014</b> (31 responses)	
3	Make HWs easier for beginners
2	Don't discuss of social issues so much
2	Don't use "Pattern" as main textbook, too philosophical
	Don't use Python again
	Don't require buying the textbook
	Don't use hot seats for bonus credit
	Don't Berate students for being too afraid to speak up

### General Feedback

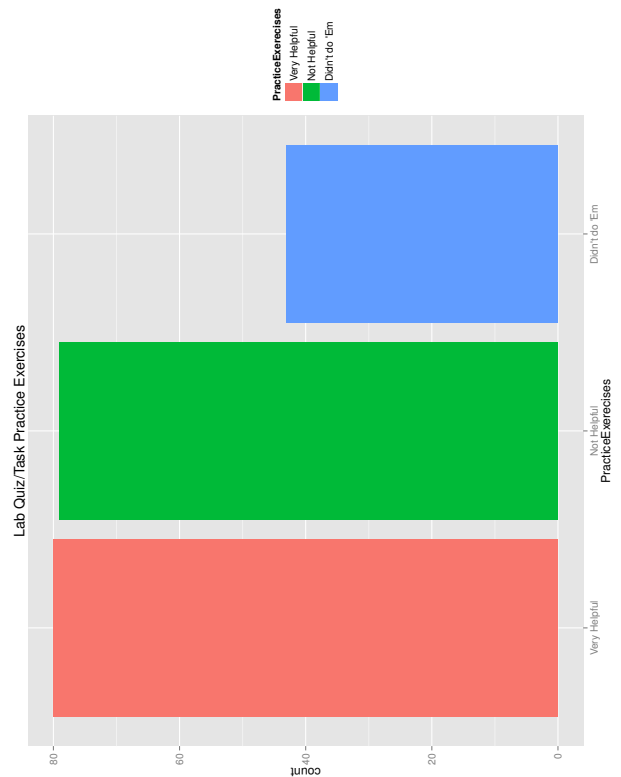
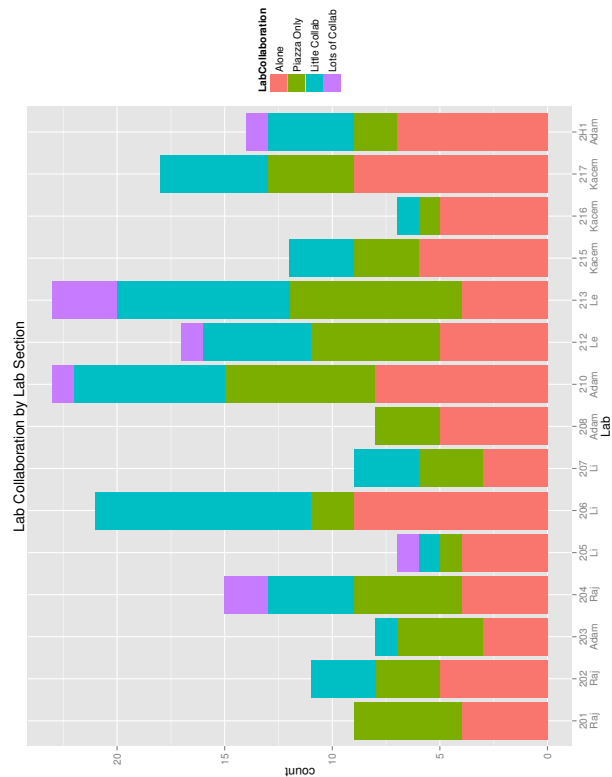
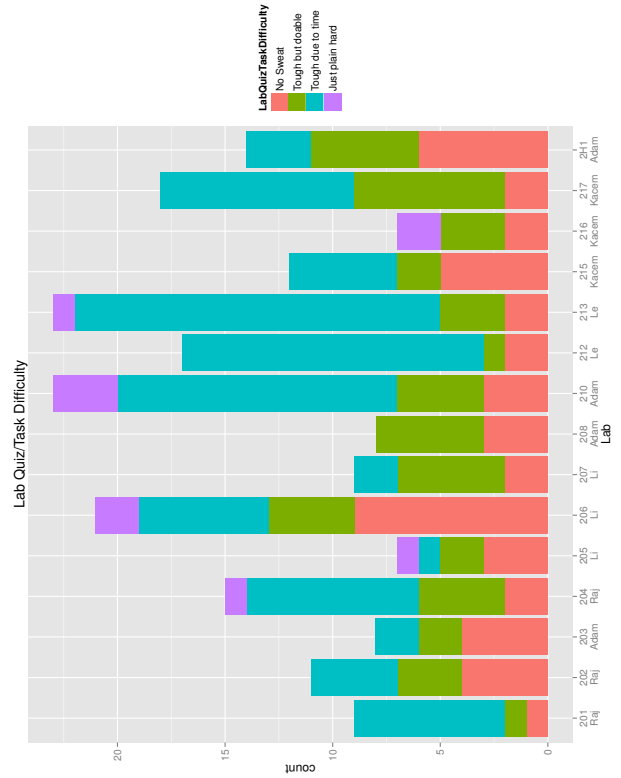
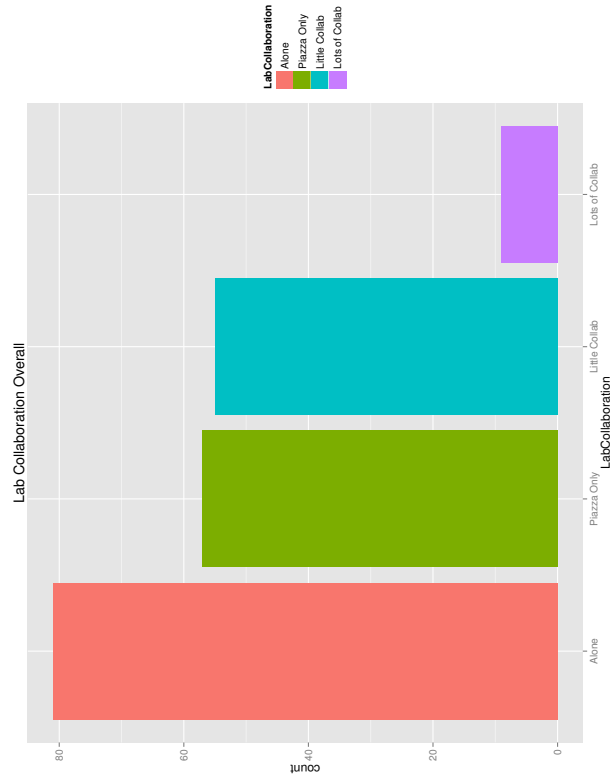
Count	Comment
<b>Fall 2015</b> (43 responses)	
4	Explain more on Python and talk about HW codes more in class
2	Grading a bit harsh
2	People who complain it's too hard should apply more effort
	Want more feedback on mini-exams
	Give out-of-class opportunities for bonus cards
	Shorten HWs, give less time to do them, have a few more HWs -> better
<b>Fall 2014</b> (31 responses)	
2	Slow down lectures some
	Require partners and assign them to make it easier to find them
	Mix theory and programming practice more in class
	Definitely got my money's worth: difficulty was more like a 200-level course
	Explain HW more in class

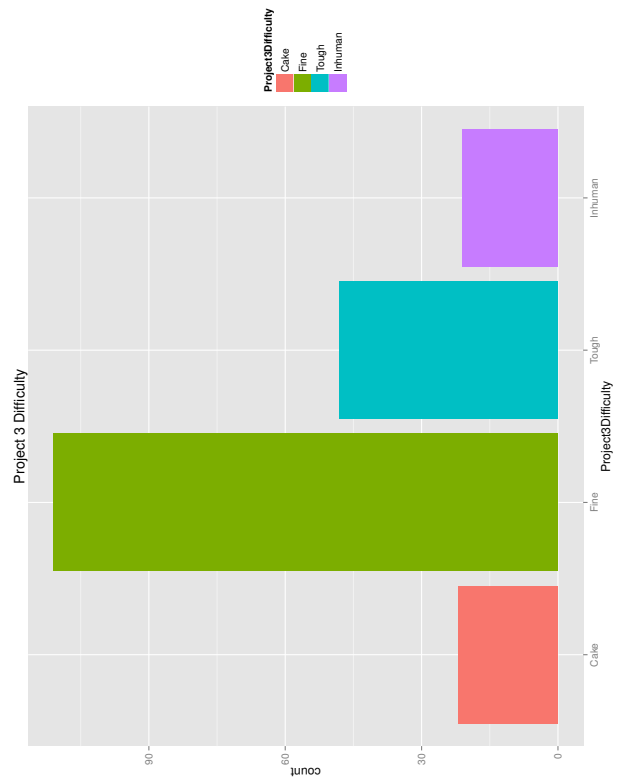
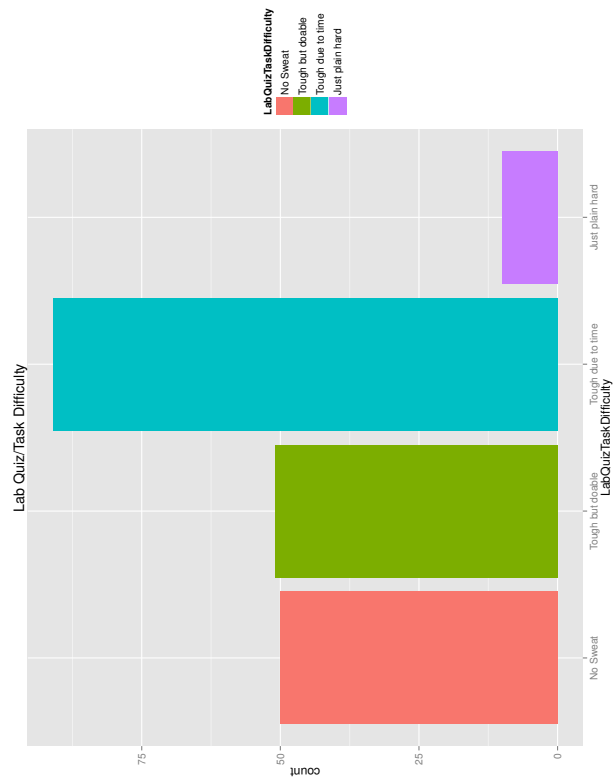
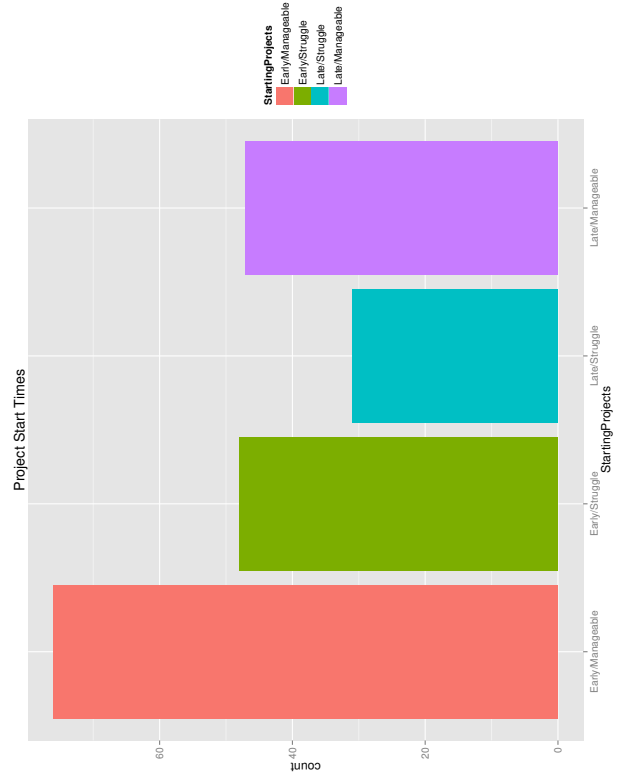
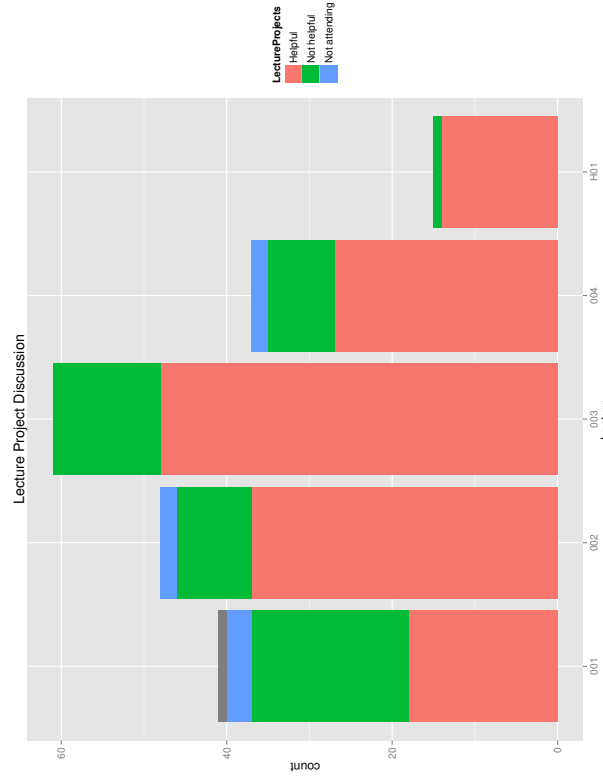
## Advice to Future Students

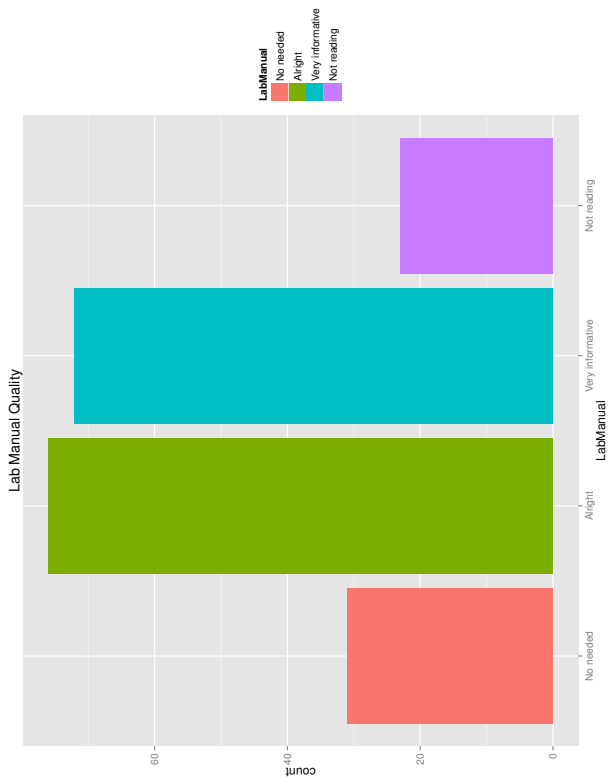
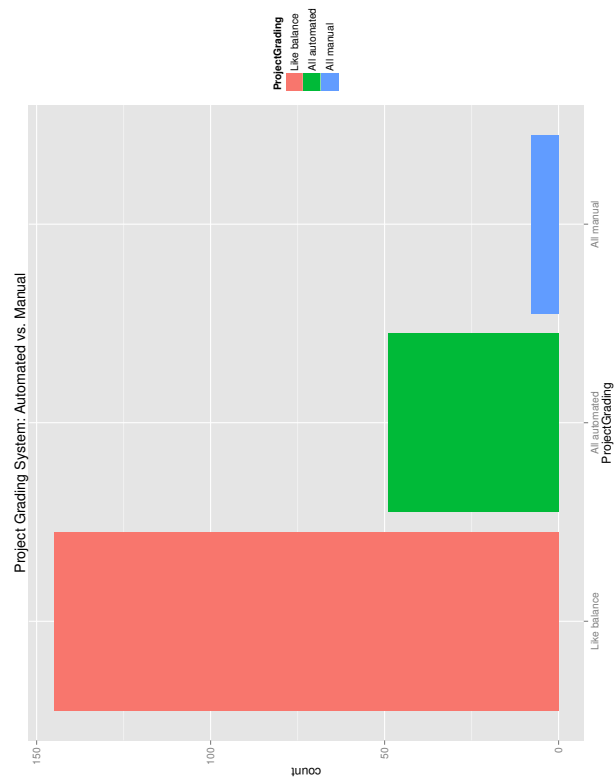
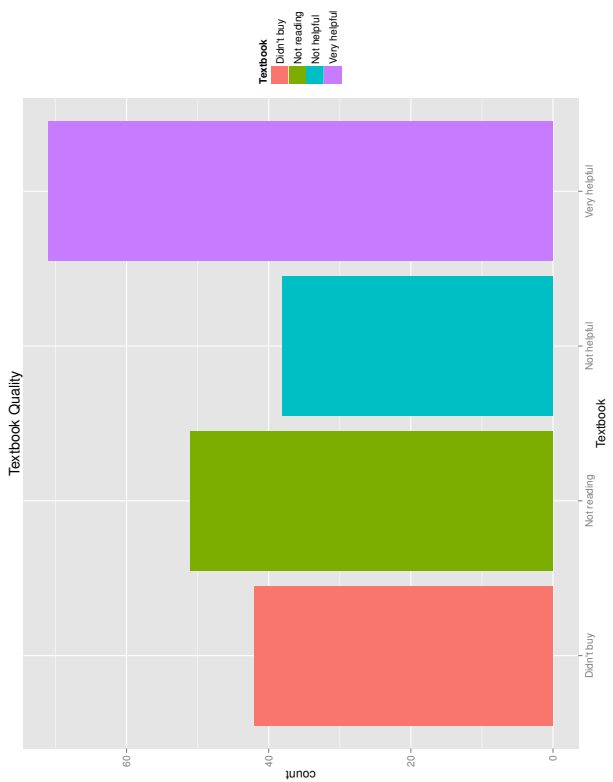
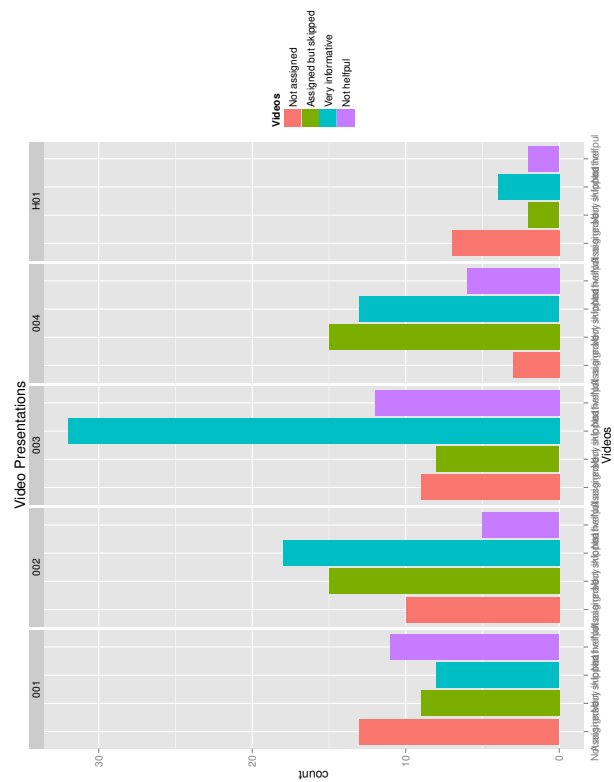
Count	Comment
	<b>Fall 2015</b> (43 responses)
3	Don't procrastinate on HW or term paper: start early!
2	Take notes as they are useful for exams
3	Don't skip classes or sleep in class
	Visit TAs to get help
	Sit in the hot seat rows: helps you focus
	Do extra Python practice

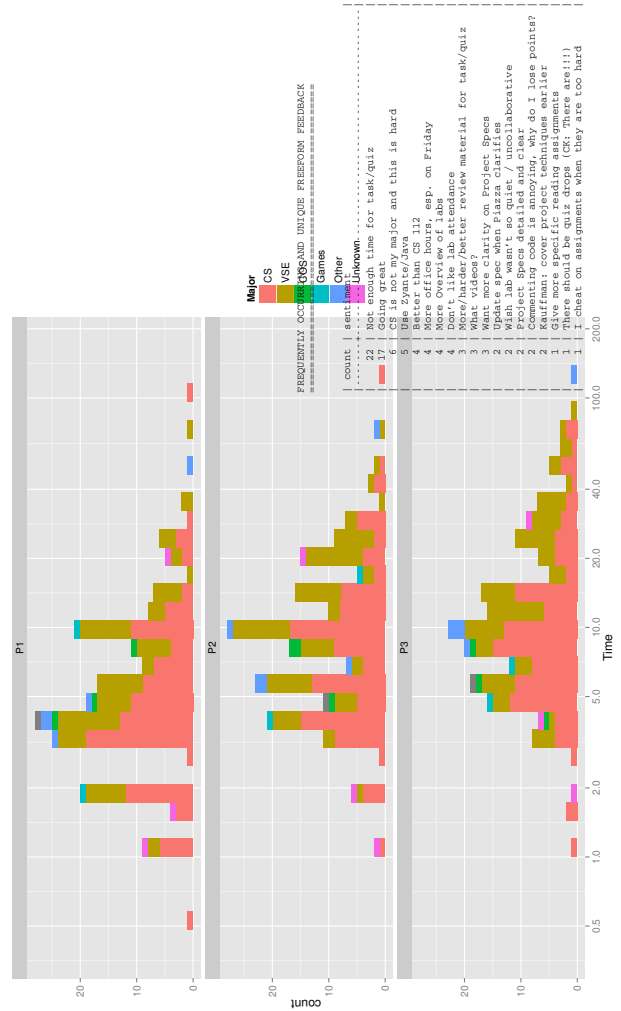
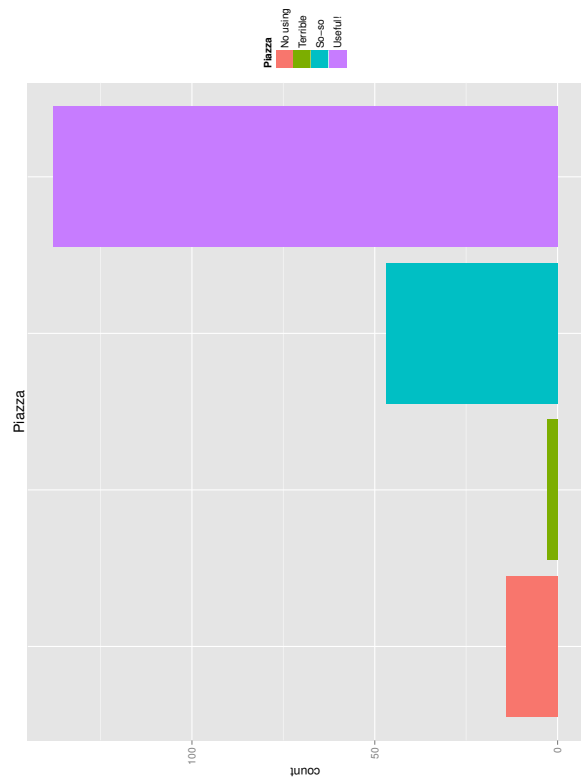
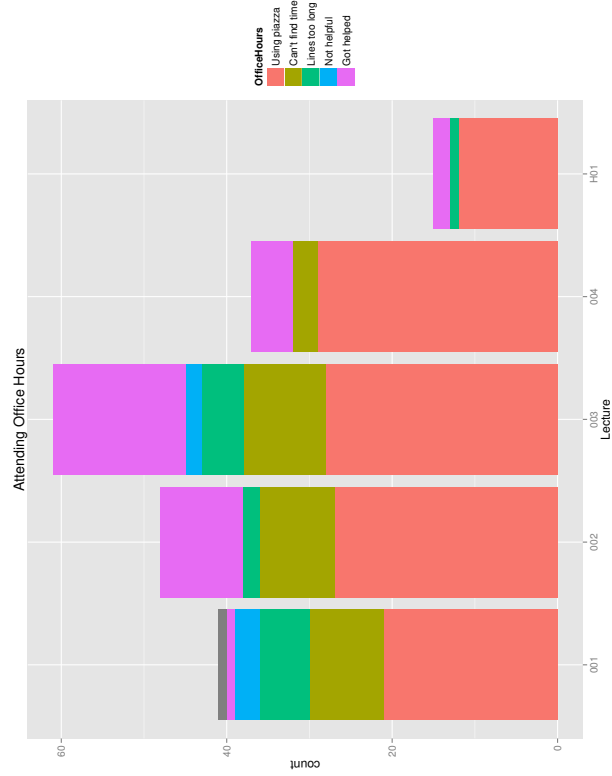












Course#	Teaching <sup>1</sup>	Course <sup>2</sup>	#Rates <sup>3</sup>	#Stdnts <sup>4</sup>	#Sects <sup>5</sup>	Course Title / Comment
CS100	4.61	4.07	85	114	3	Principles of Computing
CS105	4.77	4.24	568	654	19	Computer Ethics and Society
CS211	4.78	4.53	140	299	6	Object-Oriented Programming
CS222	4.75	4.46	105	145	4	Computer Programming for Engineers
CS310	4.70	4.44	244	364	8	Data Structures
CS499	4.84	4.84	25	33	1	Parallel Computing
<b>All</b>	4.74	4.34	1167	1609	41	Overall average of all ratings
<b>Dept.</b>	4.24	4.06				Overall Department Average ratings

- Ratings are averages over all students who provided ratings in a course.
- Teaching<sup>1</sup> measures answers to the prompt: “My overall rating of the teaching” rated from 1 (worst) to 5 (best)
- Course<sup>2</sup> measures answers to the prompt: “My overall rating of this course” rated from 1 (worst) to 5 (best)
- #Rates<sup>3</sup> counts the total number of students who provided paper evaluations over all courses
- #Stdnts<sup>4</sup> counts the total number of students who were enrolled each courses
- #Sects<sup>5</sup> counts the total number of sections of a course which was rated

Figure 28: *Summary of All Teaching Ratings over all classes*

## 6 Evidence

This section presents some additional evidence of the impact my teaching. It includes information on my numerical ratings from several few sources as well as testimonials from colleagues and students. This complements the student comments on various aspects of my teaching which accompanied sections of my Reflection. The section ends with the closing statement for this document.

### 6.1 Summary of Teaching Evaluations

Figure 28 shows a summary averaged by course over all my courses while Figure 29 shows evaluations for all courses I have taught at GMU.

While I am generally more trusting of the feedback given to me by students who are willing to slog through and entire course and attend the last week to fill out university evaluations, it is always important to consider alternative sources. The most prominent such source for college teaching is the notorious site, RateMyProfessor.com which, despite the bias inherent in any such rating site, gives some additional confirmation of my characteristics as an instructor. Figure 30 shows two samples of information available there which I feel confirm independently the lasting effects I have on students.

Term	Course	Sec	#Rates <sup>1</sup>	#Stdnts <sup>2</sup>	Teaching <sup>3</sup>	Dept <sup>4</sup>	Course <sup>5</sup>	Dept <sup>6</sup>	Course Title
Summer 2016	CS310	Bo1	28	37	4.82	3.88	4.54	3.8	Data Structures
Spring 2016	CS105	1	32	35	4.66	4.11	4.25	3.97	Computer Ethics and Society
	CS105	2	31	35	4.74	4.11	4.13	3.97	Computer Ethics and Society
	CS211	2	28	73	5	4.11	4.75	3.97	Object-Oriented Program Honors
	CS499	2	25	33	4.84	4.11	4.84	3.97	Parallel Computing
Fall 2015	CS100	2	26	34	4.73	4.18	4	4.03	Principles of Computing
	CS100	1	27	41	4.78	4.18	4.44	4.03	Principles of Computing
	CS310	1	39	58	4.77	4.18	4.3	4.03	Data Structures
	CS310	3	36	60	4.86	4.18	4.5	4.03	Data Structures
Summer 2015	CS222	Bo1	30	39	4.87	4.39	4.52	4.16	Computer Program for Engineers
Spring 2015	CS105	3	27	35	4.85	4.14	4.37	3.96	Computer Ethics and Society
	CS105	4	31	35	4.84	4.14	4.3	3.96	Computer Ethics and Society
	CS105	5	29	35	4.86	4.14	4.41	3.96	Computer Ethics and Society
	CS105	5	29	35	4.86	4.14	4.41	3.96	Computer Ethics and Society
	CS105	6	30	35	4.9	4.14	4.43	3.96	Computer Ethics and Society
	CS211	2	35	59	4.71	4.14	4.45	3.96	Object-Oriented Programming
	CS211	4	22	48	4.64	4.14	4.23	3.96	Object-Oriented Programming
Fall 2014	CS100	1	32	39	4.38	4.17	3.81	3.91	Principles of Computing
	CS310	2	31	53	4.65	4.17	4.28	3.91	Data Structures
	CS310	3	26	38	4.5	4.17	4.46	3.91	Data Structures
Summer 2014	CS222	Bo1	26	33	4.54	4.32	4.19	4.17	Computer Program for Engineers
	CS310	Bo1	29	32	4.31	4.32	4.17	4.17	Data Structures
Spring 2014	CS105	3	26	34	4.73	4.19	4.08	3.98	Computer Ethics and Society
	CS105	4	35	35	4.57	4.19	4.14	3.98	Computer Ethics and Society
	CS105	5	32	35	4.84	4.19	4.34	3.98	Computer Ethics and Society
	CS105	6	30	35	4.67	4.19	4.2	3.98	Computer Ethics and Society
	CS211	1	28	64	4.79	4.19	4.7	3.98	Object-Oriented Programming
	CS211	Ho1	15	19	4.67	4.19	4.4	3.98	Object-Oriented Program:Honors
Fall 2013	CS105	1	30	35	4.93	4.3	4.3	4.12	Computer Ethics and Society
	CS105	2	30	35	4.9	4.3	4.43	4.12	Computer Ethics and Society
	CS105	3	31	34	4.65	4.3	4.13	4.12	Computer Ethics and Society
	CS105	7	30	32	4.9	4.3	4.43	4.12	Computer Ethics and Society
	CS310	2	28	44	4.79	4.3	4.57	4.12	Data Structures
	CS310	3	27	42	4.81	4.3	4.78	4.12	Data Structures
Summer 2013	CS222	Bo1	21	26	4.71	4.67	4.37	4.36	Computer Program for Engineers
Spring 2013	CS105	5	31	32	4.68	4.29	4.13	4.13	Computer Ethics and Society
	CS211	1	12	36	4.83	4.29	4.58	4.13	Object-Oriented Programming
Fall 2012	CS105	1	32	34	4.56	4.27	3.88	4.08	Computer Ethics and Society
	CS105	7	20	33	4.7	4.27	4.1	4.08	Computer Ethics and Society
	CS105	8	32	35	4.75	4.27	4.06	4.08	Computer Ethics and Society
	CS222	1	28	47	4.86	4.27	4.71	4.08	Computer Program for Engineers

- Ratings are averages over all students who provided ratings in a course.
- #Rates<sup>1</sup> counts the total number of students who provided paper evaluations over all courses
- #Stdnts<sup>2</sup> counts the total number of students who were enrolled each courses
- Teaching<sup>3</sup> measures answers to the prompt: “My overall rating of the teaching” rated from 1 (worst) to 5 (best)
- Dept<sup>4</sup> same as 3 but averaged over the entire CS department
- Course<sup>5</sup> measures answers to the prompt: “My overall rating of this course” rated from 1 (worst) to 5 (best)
- Dept<sup>6</sup> same as 5 but averaged over the entire CS department

Figure 29: Teaching Ratings for from all courses

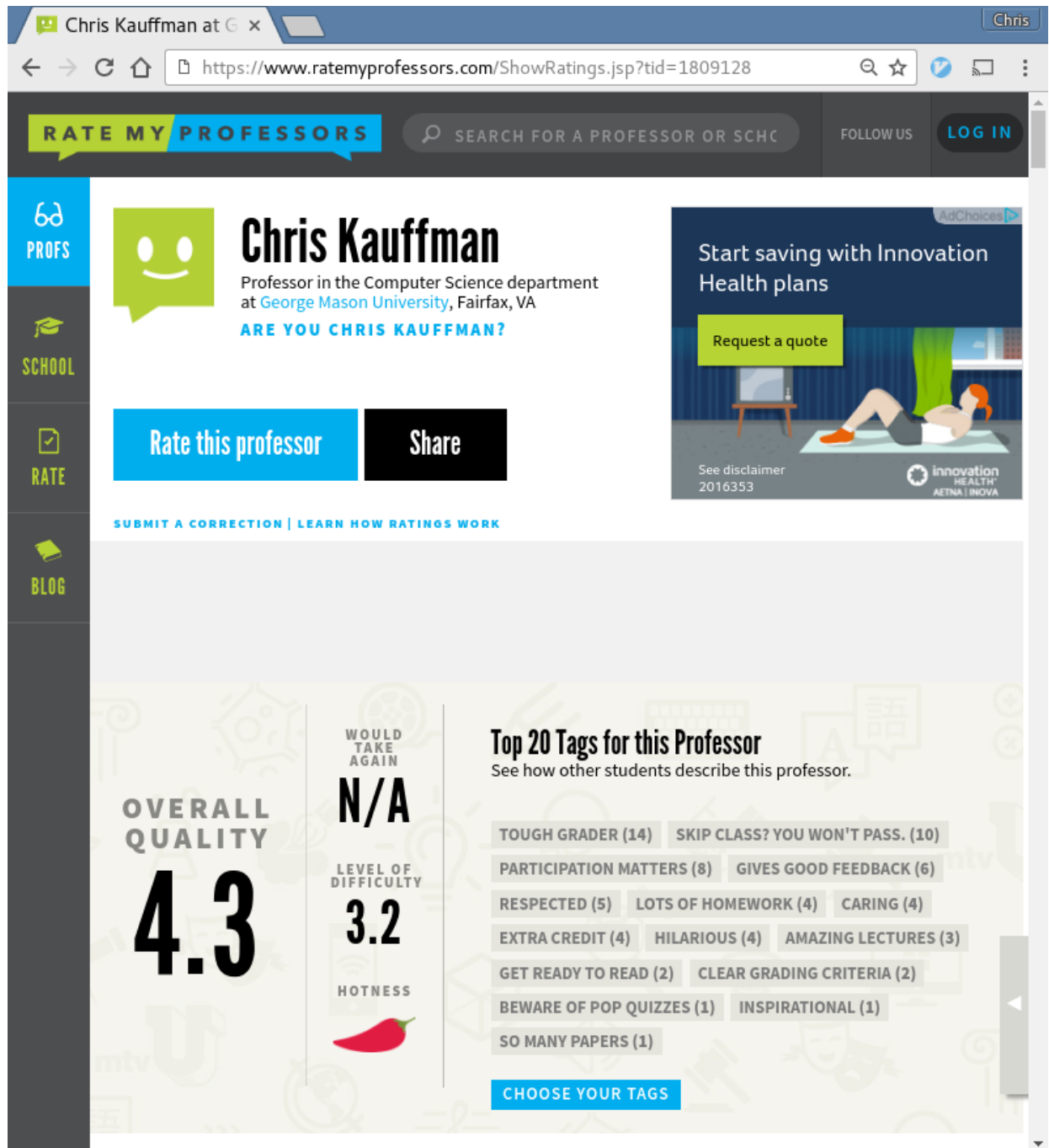


Figure 30: RateMyProfessor.com average rating and “tag cloud” which summarizes recurring teaching qualities attributed to me.

The screenshot displays the RateMyProfessors.com interface for a user named Chris Kauffman. The page shows three separate student reviews for different classes. Each review includes a date, a star rating for 'Overall Quality' (all 5.0) and 'Level of Difficulty' (all 4.0), and a written comment from the student. The reviews are for CS499, CS310, and CS211.

Date	Class	Overall Quality	Level of Difficulty	Student Feedback
05/06/2016	CS499	5.0	4.0	One of my favorite professors at GMU. His material is challenging but his focus is always on teaching concepts rather than wrote memorization. Attending and participating in class will prepare you for his open resource exams. You will actually get a sense of growth and accomplishment out of his classes rather than just learning to take a test.
04/30/2016	CS310	5.0	4.0	It's a very tough and rigorous class, but he is there to help you, where if help is needed his office hours are /invaluable/. He has his best interest in his students, being one of the few professors seriously invested in his students in the CS program. Take him if you can. Difficult, but worth the experience.
04/16/2016	CS211	5.0	4.0	Kauffman is awesome, but his projects are long and hard. I need to spend at least 3 hours every day working on his project. His exams were not too bad though. All exams were open books, notes, code he provided, or code you have written, and you can also use IDE to debug your code during exams. Highly recommend him!

Figure 31: *RateMyProfessor.com* feedback from a few students in different classes.



## 6.2 Testimonials

In the Reflection portion of this portfolio, I have interspersed student testimonials regarding specific parts of my teaching. This was in part to show evidence that the teaching techniques I have refined over my career have been well-received by students and aided their learning. Using testimonials there also allows me to adhere at least to the letter of the law that this section, devoted entirely to testimonials, remain reasonably short as by general solicitation for comments from former students produced a volume of affirmations that would not have adhered to the desired limits in this section.

Instead, this section focuses mainly on testimonials from colleagues who were kind enough to provide letters describing their knowledge of my teaching.

- Sanjeev Setia, Department Chair, CS, who has ultimate supervision of my work.
- Pearl Wang, Associate Chair, CS, who has influence me significantly through our interactions in the Undergraduate Studies committee and via our many, heart-to-hearts on working at the university.
- Mark Snyder, Coordinating Instructor, CS, with whom I have collaborated on CS 211 often, discussed teaching issues with in our SIMPLE CS development group, and mulled over department curriculum often. Mark is a dear friend and the hardest working teacher at GMU hands down.
- Richard Carver, Coordinating Instructor, CS, who has worked with me to offer sections of CS 310 which overlap topics, assignments, and schedule. A venerable force at GMU, Richard recently retired and I will miss his insight.
- Jill Nelson, NSF SIMPLE Project Leader, ECE, who recruited me to work lead a SIMPLE teaching development group in computer science. Jill is an outstanding teacher herself and cares deeply about improving education in STEM.



Department of Computer Science  
Volgenau School of Engineering  
4400 University Drive, MS 4A5, Fairfax, Virginia 22030  
Phone: 703-993-1530; Fax: 703-993-1710  
URL: <http://www.cs.gmu.edu/>

January 29, 2017

Dear Members of the Mason Teaching Excellence Award Committee,

I am writing this letter to express my strong support for my colleague **Dr. Chris Kauffman's** nomination for a Mason Teaching Excellence award.

Chris joined the CS department at Mason as a Term Assistant Professor in Fall 2012 after receiving his PhD from the University of Minnesota. During the last five years, Chris has not only been an outstanding teacher, but also an active participant in departmental discussions, and someone who has had a large (positive) impact on our undergraduate programs.

I discuss below Chris's contributions in more detail.

### **Outstanding Teaching**

Since starting at Mason, Chris has worked closely with other faculty teaching our three introductory programming courses to ensure that the sequence of topics covered throughout aligned well. He works extremely well with others and has effectively coordinated with colleagues to increase the rigor of the courses and improve the utilization of course lab meetings led by graduate teaching assistants. This resulted in a variety of concrete materials for GTAs to use during labs and guidance to them on how to effectively help students during labs.

Complaints from faculty teaching upper-level courses have diminished greatly since Chris began teaching the lower level courses. Through his efforts and coordination with other teaching faculty, our CS majors are much better prepared after working through the introductory sequence of courses. This benefits the students and the entire department.

I review the student evaluations for computer science professors on a regular basis and have never had to worry about Chris: his evaluations have been nearly uniformly outstanding, among the best in the department. In recognition of his outstanding teaching, Chris was awarded the department's Teaching Excellence Award in Spring 2015.

### **Curriculum Development**

Chris has developed several new courses for the department. A few years ago, we wanted to explore introducing a computing course for non-majors that covered the

basics of computing. Chris had been instructing CS 105: "Computer Ethics and Society" regularly and so was a natural choice to help develop the course and offer it. The resulting course, CS 100: "Principles of Computing", has proved very successful with students commenting positively on the practical skills that they acquired and the energy he brings to the subject.

Chris also developed a special topics course CS 499: "Parallel Computing" that was extremely well received by the upper-level students who enrolled in it. The course is likely to become a permanent fixture in the department in the next few years. Finally, he has contributed to developing a new introductory course which we will begin teaching next year to ground our CS majors in the basic tools, skills, and ethics required in our discipline.

### **Student Mentoring**

Chris is the faculty advisor for Patriot Hackers, a student organization which enhances student computing at our school by discussing and educating students about computer security issues. Patriot Hackers has a wide following computer science, applied information technology, and other students who meet regularly. Chris has provided them support by obtaining computing resources, liaising with the university community for outreach, and give advice to the group leadership on how to preserve continuity from year to year.

### **Service Contributions to CS Undergraduate Programs**

As mentioned earlier, Chris has made many contributions to the department through his active participation in the undergraduate studies committee. In addition, Chris was one of the primary forces behind the creation of a departmental honors program which we started this year. The proposal he wrote for it was extremely well thought-out and among the best I have seen come through the department. It passed the faculty vote unanimously on account of his excellent documentation and explanation for it during the meeting.

In summary, Chris Kauffman is a great teacher and an outstanding young faculty member in every respect. He has my strongest recommendation for the Mason Teaching Excellence award!

Sincerely,



Sanjeev Setia  
Chair, Computer Science Department  
George Mason University



Department of Computer Science  
Volgenau School of Engineering  
4400 University Drive, MS 4A5, Fairfax, Virginia 22030  
Phone: 703-993-1530; Fax: 703-993-1710  
www: <http://www.cs.gmu.edu/>

January 30, 2017

Center for Teaching and Faculty Excellence  
Office of the Provost  
George Mason University

It is a pleasure to provide this letter in support of Professor Chris Kaufmann's application for the GMU Teaching Excellence award. Chris joined the Computer Science department in Fall 2012 as a Term Assistant Professor. Earlier that year, it was my responsibility to host his faculty interview visit, and at our very first meeting, I was thoroughly impressed by his enthusiasm and interest in undergraduate computer science education. Since that time, Chris has proven to be an integral member of our faculty and has contributed not only by providing teaching support for our curriculum, but also for the many efforts he has made towards improving it. Some of his exceptional contributions are summarized below.

As a faculty member, Chris has teaching and service responsibilities associated with the delivery of our two undergraduate computer science programs. He has served as faculty advisor to a large percentage of our undergraduate majors and consistently volunteered for our open house and high school recruitment events. He has mentored many of our undergraduate majors and taught a range of courses from large section introductory programming to senior level advanced classes. Furthermore, he has strongly supported the department's efforts to expand our undergraduate research opportunities for computer science students.

In my role as chair of our CS Undergraduate Studies Committee, I have interacted with Chris regularly since he joined this committee after his arrival. As a member, he has proved to be a reliable source of information for what has been happening in our introductory CS courses, and he has consistently led efforts to improve the teaching of these courses.

For example, as part of our undergraduate programs' continuous improvement efforts, we regularly collect and review faculty reports on learning outcomes achievement. Chris's reports always provide informative insights on what happened in his classes and have contained useful suggestions on how to improve teaching effectiveness in subsequent offerings. His recommendations have been highly valued by his peers and have been adopted within our curriculum.

Chris has also assisted with our efforts to improve overall education in the CS department by applying his expertise in data analytics to sift through student performance reports for patterns related to learning outcomes achievement. Often this was done with minimal prompting, just a casual inquiry from him of, "Oh, you have some data: could you share it with me, I'm interested!" The reports he has provided over the years were extremely valuable for our recent BSCS ABET accreditation review and helped us achieve re-accreditation this past year.

Just recently, I asked Chris to investigate whether we could establish a departmental Honors program for our majors. Several other engineering departments were considering this as well, and our most talented CS students would definitely benefit from such opportunities. Chris took to the task with vigor, meeting with representatives from the Honors College to ensure our program complimented theirs rather than competing with it. He also met with many faculty members to get input on how the program should be

shaped. Through his hard work, he managed to find a solid middle ground among more than forty widely varying faculty opinions. His Honors program proposal passed both our Undergraduate Studies Committee and a full department faculty vote unanimously, giving us the means to encourage and recognize exceptional CS students.

Chris has demonstrated his dedication to teaching in other ways as well. I have been a frequent attendee at SIGCSE, the largest CS education conference and encouraged Chris to attend to meet other educators. He did so several times and has been extremely active at the conferences, attending as many sessions as possible to gain knowledge that he could bring back to the classroom. For example, he attended a workshop on parallel computing education at SIGCSE to help prepare for a new undergraduate course he was developing on the subject. This new course was first offered here at Mason as a special topics class and was extremely well received by students. With his assistance, it will become a permanent course offering for our seniors in the future. He also attended many SIGCSE sessions on flipped classrooms to assess whether he should try the technique in his own classes and sat in on a session about conducting educational research in order to assist our department project that is evaluating self-pacing techniques applied to our programming courses.

The CS department also regularly offers a research inquiry course developed with OSCAR support - CS 390: Research and Project Design Principles in Computing. The course features a series of guest lectures on different research topics. Chris has been invited many times to speak about his background in machine learning and bioinformatics. He led well-conceived and thought-provoking discussions during several semesters. He also volunteered several times to attend the end-of-semester student presentations in CS 390 to offer feedback and contribute to decisions on best project awards. His encouraging words and interest were much appreciated by students in the class.

Chris was similarly instrumental in the creation and formulation of a new Mason Core course, CS 100: Principles of Computer Science that was designed to increase awareness of computing concepts and computational thinking for non-majors. This course aligns with national efforts to promote STEM education in the K-12 arena. Chris has taught and refined this course over several semesters and his ideas have been adopted into its syllabus.

These contributions show clearly that Chris has an intense interest in teaching and a devotion to improving his own pedagogy. We are extremely lucky and grateful to have him on our computer science faculty. His many contributions are highly valued and have been recognized by our students and his colleagues.

With regards,

Dr. Pearl Y. Wang  
Associate Chair & Associate Professor  
Computer Science Department 4A5  
email: pwang@cs.gmu.edu  
phone: 703-993-1527

I am a Term Assistant Professor here at George Mason University. I am pleased to write to you today on behalf of Chris Kauffman. He is disciplined, inventive, and assertive. He's also the best friend by far I've made of a colleague on this campus, for everything from his personality and work ethic to his ability to bring people together. While shining in his role as teaching faculty, he has fostered a strong atmosphere for undergraduate research at the same time. I've had the opportunity to see his entire experience here at GMU, as I was hired for the same role just one year before him. We also coordinate each spring to teach our Object Oriented Programming course, and it has given me valuable insight into his teaching philosophy and practice. He has excellent instincts as a teacher, he understands his students, and he constantly works to hone his craft. He is a positive influence on the students in his class and in research opportunities, on other professors in his department, and as a friend.

Chris teaches many classes as needed by the department, and brings his specific talents to bear. Sometimes these courses were not direct matches to his background, such as our GenEd ethics course (CS105, "Computer Ethics and Society"). Other times, he's stepped up to provide the initial offering of a course like our general overview for non-majors (CS100, "Principles of Computing"). He has always stepped up as the department has called upon him. I'd say though that much of his teaching effort has gone into our Object Oriented Programming (OO) course and our Data Structures course, taught as second and third programming courses at GMU. The OO course is where I've had the best chance to witness his philosophy of teaching in action, as we've coordinated each spring on our sections together. He fosters a high level of motivation and expectations of our students, and he makes some great projects that truly get students excited. I'd taught the course once before his arrival, and with his input the course immediately became stronger in seemingly every way – better use of time meeting with GTAs each week; larger, more interesting projects that pushed our students further; better connection to the following Data Structures course; good refinements to topic ordering and adding a few more topics here and there. He also influenced how we grade assignments, bringing in outside ideas about student motivation - we now have half of the assignment graded by provided test cases, and the other half by qualitative attributes that are manually checked. There is always a solid reasoning behind his suggestions, too – for instance, he believes that students need experience seeing larger code bases and interacting with them; this is exactly what happens in the workforce, day after day. He therefore intentionally makes projects requiring it, balanced against the need for them to create and experience new patterns of program structure on their own. I know his Data Structures course is similarly strong, in seeing the students after the end of the semester, and in seeing the same sorts of complex, involved, and engaging projects. Chris also offered a Parallel Programming course last spring as a senior elective, bringing his research expertise to the classroom. The course is a great example of what a senior special topics course should be: a way to push students' boundaries, work on real problems, learn applicable skills, and bring the professor's experience directly to the students. Students were clearly excited and invested in the class – I'd hear them talking about his class during breaks in my own! I hope this course becomes a regular offering, as it would be the strongest course to fit the "concurrent programming in a senior elective" requirement.

Many of our best students have worked with Chris on undergraduate research projects during his time here, through OSCAR. He has worked with students on an educational game about AIDS, upgrades/maintenance on an open source code editor that we use in class, performance overhauls for handling large datasets for statistics in the R programming language, and others. These show his

commitment to continuing research and improvement, as well as his commitment to including students in research. As teaching faculty, his job does not require or reward him for any of that; Chris is just a force to be reckoned with, plowing ahead all on his own internal drive, because it's worthy work. He finds opportunities to make an impact, and he makes it happen. His office is just down the hall, and there is often a pile of students hanging out, studying up for a test, having an impromptu lecture session hammering out more details from class, or visiting for various undergraduate research project sessions. They often visit in groups – this reaches more students, and also creates a more cohesive community for learning. He supports the Patriot Hackers student group on campus (teaching industry-level cyber security skills), and holds extra review sessions before finals, with an atmosphere of camaraderie and community. In class, it's clear he focuses on giving students chances to try, to learn, and to show what they know. He encourages participation by offering extra credit for asking and answering questions via a system where students get points (physical playing cards) for answering questions, but they have to sit near the front to do so; students can also sit further back and observe without fearing being called upon any particular day. It's a semester-long chance to participate and nudge grades upwards (the most cards earning the maximum extra credit), perhaps far more by their being present and participating than by what portion of credit they actually earn through the cards. Other ways he has encouraged students' growth is by offering students multiple paths on an assignment to get to the maximum score – either by doing a lot of simpler practice and some harder tasks (good for weaker students who need the practice), or by going straight to a larger list of harder tasks, which suits the advanced students. Managing the needs of a wide array of students is challenging, and he consciously addresses them simultaneously. Chris learns new teaching techniques and tactics, and brings them into the classroom. From attending the SIG-CSE conference (Special Interest Group – Computer Science Education) most years to catch up on the current academic research for Computer Science, to on-campus activities by leading a small reading group about teaching (part of the SIMPLE project on campus), Chris continues to push his own capabilities and lift up others with him. This is where I have had the best chance to continue my own education, all thanks to his inviting me to participate. Through other more social activities such as a department running club and playing squash on campus, he meets individuals from other departments and then makes connections and builds out his network of contacts. His presence is felt far beyond his own classrooms.

Overall, Chris is one of the strongest teachers in our large department. We rely upon him, and he always delivers. He brings a personal touch, a studied and researched approach to his classroom, and seemingly always has time and a smile for the endless waves of students. He repeatedly has had visible positive impacts on our department, our students, and other professors. Maybe it's part of his Midwestern upbringing – even before his first day of classes, he offered to help *me* move! Happily I've been able to return the favor since, but the point is that it's important to understand that Chris is always looking out for those around him. He is always selfless, always growing as an educator. We are lucky to have him here at GMU; he's built a strong foundation for himself and is reaching his potential by taking each opportunity he can find to improve himself and the community around him. I believe he represents not only the best GMU has to offer, but also the best spirit of higher education as well, and for this reason I consider him thoroughly deserving of GMU's Teaching Excellence Award.

Mark Snyder

Department of Computer Science

Dear Madam or Sir:

Tuesday, January 31, 2017

I was a faculty member in the Computer Science department for 27 years before retiring this January 2017. For the past 4 years, Chris Kauffman and I taught sections of the same courses. These courses were part of a three-course sequence on programming at the beginning of the computer science curriculum. The majority of the instruction students receive on "how to program" is in these courses.

Learning how to program is difficult for students. They acquire sufficient programming skills only after many hours of practice. Teaching students how to program is also very difficult. Instructors are required to design a series of programming assignments that cover the requisite programming language concepts and require the solution of increasingly difficult problems. These assignments are a critical factor in determining how far students advance in the first two years of their computer science education.

Chris did a masterful job planning and preparing these programming assignments. In general, instructors draw on their personal experience to generate programming problems. Careful planning is needed to ensure that problems are neither too easy nor too hard to solve. Typically, an assignment asks students to spend 20-30 hours of programming time over a three-week period. Chris' assignments were not just challenging, they were interesting and fun, and motivated the hard work required to solve the problems. Chris develops his assignments by iteratively writing and refining programs until they are suitable for an assignment. This creative process integrates the programming concepts and problem solving skills to be covered with just the right amount of complexity, in an interesting way. Chris does this better than any computer science instructor that I observed at George Mason.

Each assignment must be carefully specified so that students know exactly what they are required to do. For complex problems, writing clear assignment specifications is almost as difficult as solving the problem. Chris' specifications were well-written technical documents that provided all of the information the students needed to complete the assignment.

For each assignment, Chris provided a set of test cases that the students used to assess the correctness of their programs. Each of the several hundred test cases was itself a small program that automatically tested the correctness of one small aspect of a student's solution. Teaching assistants used the test cases to grade student programs. The tests themselves also acted as a clear specification of what the instructors expected the program to do. The written assignment-specification and the test cases together ensured that students understood the assignment.

Students can become frustrated with difficult assignments. Chris made sure that students could get help and make progress towards a solution. Students had access to undergraduate teaching assistants and Piazza discussion boards. Chris also held an end of semester "Codefest," which was a Friday night work and play session, complete with music, pizza, and drinks. The Codefest supplied students with help and hope at the end of a long semester, and affirmed that learning was fun.

Richard Carver  
rcarver@gmu.edu





The Volgenau School of Engineering  
Department of Electrical and Computer Engineering  
4400 University Drive, MS 1G5, Fairfax, Virginia 22030  
Phone: 703-993-1569; Fax: 703-993-1601

January 30, 2017

Dear Members of the Selection Committee,

I am thrilled to write this letter in support of Dr. Christopher Kauffman's nomination for George Mason University's Teaching Excellence Award. I am an Associate Professor in the Department of Electrical and Computer Engineering and a 2014 recipient of Mason's Teaching Excellence Award. I know Chris through his participation in a faculty development project that I lead in collaboration with several faculty members from Mason's College of Education and Human Development. The primary goal of our NSF-funded project is to broaden the use of evidence-based interactive teaching and active learning practices in science, technology, engineering, and mathematics (STEM) college teaching. We aim to advance adoption of interactive teaching through long-term faculty development groups that are built on the SIMPLE framework, which encourages instructors to make small changes to their teaching over time and revise those changes with feedback from a community of instructors in their field. SIMPLE faculty development groups operate within several STEM departments at Mason. SIMPLE groups meet regularly to provide accountability and support for participants as they learn about and implement interactive teaching methods.

During the first year of the SIMPLE project, Chris participated in a SIMPLE faculty development group that included faculty from electrical engineering, computer science, and bioengineering. He was a dedicated and enthusiastic member of the group, sharing ideas for how to enhance teaching and proposing topics to discuss. After witnessing his passion for teaching and for student learning, I asked him if he would be willing to lead a computer science-focused faculty development group the following year. He agreed with minimal coercion, and he now leads one of the most active SIMPLE groups at Mason. Chris recruited four CS faculty members to join him, and the group has been going strong since Fall 2015. He selects books to scaffold the group's discussions and provides resources and encouragement to members as they try new evidence-based teaching strategies in their courses. When Chris and I cross paths, he is always excited to tell me about the new things he is doing in his classroom and what challenges in CS education have been discussion topics in his faculty development group.

Beyond the CS group, Chris has also been an incredible asset to the SIMPLE project as a whole. As part of the 2016 Innovations in Teaching and Learning Conference, we organized a SIMPLE Poster Session to feature the teaching innovations implemented by members of SIMPLE groups at Mason. Chris presented a poster describing his use of "hot seats" to increase student engagement in large courses. The poster received considerable traffic, and Chris was happy to share his experience and insights with interested instructors. In addition, Chris served as a discussion leader during our SIMPLE Summit, an interactive meeting designed to bring together SIMPLE faculty development participants from across disciplines.

In all my interactions with Chris, I am impressed by his remarkable dedication to teaching and to improving his students' learning. He has gone above and beyond in his classroom and in his contributions to furthering teaching development at Mason. I feel very fortunate that he is such an active part of the SIMPLE project, and I look forward to continued collaboration in future years. If I can provide any additional information, please do not hesitate to contact me.

Sincerely,

A handwritten signature in black ink, appearing to read "Jill K. Nelson", with a long horizontal line extending to the right.

Jill K. Nelson  
Associate Professor  
Department of Electrical and Computer Engineering  
George Mason University

### 6.2.6 Student Successes

The following are excerpts from student letters of support for this portfolio which specifically mention the lasting impact I have had on their academic careers and futures.

**Emmanuel Tagoe** CS 211 with Dr. Kauffman impacted my life here at Mason the most. It helped me land my first internship, which then opened up many opportunities for me. Dr. Kauffman's projects for CS 211 were also very challenging and provided me with the necessary skills and knowledge to be successful in my second internship, in which I had to write Java code for android development

**Bridget Lane** The work he does is greatly appreciated by alumni like me, who understand that part of the reason we can succeed in the workforce today is because of what he does in his classroom every day.

**Ananya Dhawan** I am a current Graduate student at George Mason University pursuing my PhD in Computer Science and hold a Bachelor's of Science in Computer Science from GMU as well. It was during my undergraduate studies that I had my first introduction to Professor Kauffman, and from the very first lecture itself, I saw his enthusiasm for both teaching and learning.

**Jimmy Boddien** It is worth mentioning that professor Kauffman truly wishes for his students to learn, which is why he employs a variety of means to ensure that they are given whatever is necessary to facilitate their assimilation of the lecture material... Because of all of this, I can earnestly claim that professor Kauffman has positively influenced my academic life at George Mason University.

**Mohammad Ahmad** His enthusiasm sparked a deep interest in [parallel computing] for me. And when the time for interviewing and deciding on my first job rolled around, I knew exactly what I wanted to do. Professor Kauffman challenged me to become a better student and lit the spark for a lifelong curiosity that will undoubtedly help me throughout my career.

**Cycliya Schultz** Over the course of approximately two years, I worked as an undergraduate research assistant under professor Kauffman's guidance. I cannot praise him more highly for his mentorship in that regard. He empowered me to succeed by ensuring I had access to critical resources, while providing me with many opportunities to contribute to the project with my own ideas and explorations. He continually strove to balance helpful support with confident trust in my abilities, and in so doing, enriched my experiences of both academic and professional endeavors.

**Briana Abrams** Professor Kauffman has showed that he truly cares about his students and their success in his class and has clearly invested time and effort to ensure our success.

**Selena Chaivaranon** The [CS 100] term paper assignment in particular offered me the opportunity to practice research strategies and the process of synthesizing and discussing literature, an experience which I feel helped prepare me to write outside of the context of computing, including sociological literature reviews, a university capstone research project, and most recently, my research project for URSP.

### 6.2.7 Student Thank Yous

This section ends with a small collection of thank-yous I have received unsolicited from students over the past few years. The thank yous are from

- Emily Owen who took CS 100 with me during Fall 2015. Emily was enjoying the class greatly when she had a suffered a concussion which required her to take some time off. I worked with Emily to ensure that she could finish the class which she appreciated greatly.
- Shakeh Simanian who took CS 211 with me in spring 2014. She was an excellent student and programmer who suffered a health incident during the final exam; I recognized something was wrong and asked her to make up the final later for which she was grateful.
- Alison Scott who also took CS 100 during Fall 2015. Alice was initially somewhat disengaged from the class but over time my enthusiasm seems to have won her over.

Dear Professor **Kauffman**,

I **MENTO** express my sincere **gratitude** for the additional hours spent **helping** me retain the basics of programming in **CS100**.

Fall semester (2015) I endured numerous obstacles physically (concussion), mentally, and emotionally that hindered my academic performance.

However, the **extra support** of your office hours provided me <sup>with</sup> the opportunity to **truly** succeed in CS100. I gratefully appreciate your **time** and **motivation**.

**Cheers!**

**Emily Owen**

Dear Professor Kauffman,

I just wanted to write you a quick note to thank you for all you've done & to let you know that I greatly appreciate the opportunity that you gave me & the time you took to send me the e-mail, ask me what happened, let me take the test & to grade it again for me. I've learned so much from you & I hope that I'll have the opportunity to take future courses with you.

Thank you for being a remarkable teacher and I am very grateful that I've had the pleasure of being your student.

Once again, thank you for your time & help.

Sincerely,

Shakeh S.

Professor Kauffman,

wish upon a snowflake

Thank you for an interesting semester! I definitely learned a lot and had fun along the way. Happy Holidays, have fun with your giant Wreath!

-Alice Scott

Figure 32: Unsolicited student thank yous.

### 6.3 Closing Statement

Perhaps you have slogged through to the end of this gargantuan, carefully noting my assorted assignments, assessments, and developments. Or perhaps you have simply skipped to this end note to see what final words I will use to convince you of my teaching prowess.

I will say this: being a hard person to convince of anything myself, I leave it to the preceding evidence of this document to make my case to you. As for me, those that have spoken to me about my teaching have done so unequivocally.

- Colleagues have shown extremely strong support with kind words and affirmations of my passion for teaching. I am, of course, skeptical of them as none of them have actually had to *take* a course from me but their words are appreciated nonetheless.
- Students in aggregate on university feedback forms rated me favorably. One anonymous rating site characterizes me as the tough but fair instructor who will deliver “the real CS experience.” This is a persona of which I am proud of and will always seek to promulgate.
- Students have at regular intervals expressed gratitude spontaneously to me for my contributions to their learning. This touches me deeply.
- Most gratifying of all, when I asked former students to help with this document by providing me with letters of support, I received an overwhelming response, 14 pages of praise from 12 students.

This last item is ultimately what convinces me I am on the right track as an educator. Students will forget my name sooner or later. They will move beyond the specific skills I teach. They will find wiser, closer mentors. The pizza I bought will be lost in the sands of time. But hearing that in this brief period I have positively impacted their direction in life enough for them enough to say so, this convinces me to keep cracking jokes in lecture and hatching devious projects.

Looking forward, I have a variety of goals associated with education. I feel my teaching itself is in a strong place but there are deficiencies in the supporting tools and structures associated with CS education. For example, we just don’t have a good tool to automatically draw data structures which appear in student programs. This would drastically simplify several aspects of teaching. Also, there are few texts that emphasize the relation between writing programs and manipulating the working memory of a program. And there are endless discussions of the effects new forms of automation (read: self-driving cars) will have on all humans. All these are gaps I am interested in filling.

At the moment, however, it is time to go to bed. Thank you kindly for your interest.

Cheers!

A handwritten signature in black ink that reads "Chris Kauffman". The signature is fluid and cursive, with the first name "Chris" and last name "Kauffman" clearly legible.

Christopher Kauffman