That Most Important Intersection

Lane A. Hemaspaandra

Department of Computer Science,
University of Rochester, Rochester, NY 14627, USA
`http://www.cs.rochester.edu/u/lane`

Abstract. This article describes the design and components of a Knuth-style problem-solving course for *undergraduate* students, including example problems. The hope is that the course approach is flexible and adaptable enough that it will be relatively portable, and yet will provide a course that for both teacher and students falls in that most important intersection: the intersection of what one can do quite well, and what one can do with much passion and joy.

To Juraj Hromkovič, with deepest thanks for his contributions to both theoretical computer science and computer-science education, and with best wishes for the future.

1 Preface: That Most Important Intersection

A Warning. This is not an article schooled in the rigor of or terminology of educational research, or based on well-designed surveys and studies. Rather, this article consists of some personal, informal, first-person reflections on bringing a Knuth-style research-immersion course to *undergraduates*, and is an attempt to make more generally available the course design, so that others can, if they wish, adopt it and adapt it to their own styles and their schools' needs. In doing so they may, if the stars align, create a course that is a joy for its teachers and students alike, and that helps the students realize that their abilities at problem-solving are far more powerful than they had previously realized.

1.1 In Prydain

One of my favorite book series as a child was Lloyd Alexander's *Chronicles of Prydain* series [2]. If you don't know of it, I'd urge you to find and read it. My guess is you'll be glad you did, and not just because you can then give it as a wonderful gift to your child-aged relatives.

One might at first think the series is merely yet another coming-of-age tale wrapped as is common in what at the start is a magical world. But within the series, one book really jumps out from the rest, and from the genre as whole, as having a message both quite insightful and brutally realistic.

H.-J. Böckenhauer et al. (Eds.): Hromkovič Festschrift, LNCS 11011, pp. 568–589, 2018.
https://doi.org/10.1007/978-3-319-98355-4_33

In particular, the fourth book in the series is *Taran Wanderer* [3]. In it the protagonist, Taran, starts as an assistant pig-keeper (although to an oracular pig). But Taran doesn't know who his parents are, and feels as if he doesn't even know who he is—whether (in the feudal-flavored setting of the series) high-born or low-born—or what he should be doing in life. So he goes on a journey, seeking his origins, seeking his identity, and in the process trying his hand at many occupations. And at each occupation, it turns out—lucky him—that he is remarkably talented. He turns out to be truly talented in smithing, weaving, and so on. And yet at each, he realizes it just isn't right for him—it simply isn't what he wants to do. Finally, he tries his hand at pottery, and he truly loves it. It is what he wants to do. Yet he slowly comes to realize that this is the one thing he has tried that he is not truly talented in. He could stick with it and be quite competent, but he'd never really produce work that transcends the mundane. And at the end of this whole process, he realizes (with the help of a perhaps not-so-enchanted water puddle that is claimed to show one's true self yet shows him only his own reflection) that it doesn't matter to him whether he is high-born or low-born, and that being an assistant pig-keeper is something he both is specially talented at and loves.

There is a lesson there for everyone. There is nothing better in life than spending one's time on things that one loves—things that truly resonate—and that one is lucky enough to have the talent to do excellently.

And in each of our young lives, we wander seeking that most important intersection, even if we do don't consciously think about the fact that that is what we're doing. If we're unfortunate or unwise or faced with circumstances that constrain us, perhaps we give up and settle, maybe not even realizing that that is what we have done. But if we're very fortunate indeed, perhaps we find our own personal most important intersection, where talent and passion coincide, and we can find joy.

Admittedly, this most-important-intersection view is too rosy and oversimplified. Perhaps Taran would have loved, and been even more talented at, arrow-making. And most of us don't possess utterly sublime talent at anything. Even if we did, passions and skills can shift or diminish over time.

So, perhaps, what one should more plausibly try—even while hoping for far more—is to find an area for which one's passion is great, and one's talent/ability is quite strong relative to what one's talent is in other areas where one senses one would feel passion. Even that is still a wonderful, difficult intersection to find.

1.2 In Academia: Research

Theoretical computer science, my own specialty, is an area where people relatively clearly have found so very well their most important intersection. One looks around, and pretty much everyone truly loves what they are doing. And although we all vary in our talent, and the advance of knowledge needed to solve the hardest questions of the field is a long game that may span many lifetimes, and most people are doing work that has real meaning and depth, and shows their (and their coauthors') fingerprints and flair.

I much suspect that the same can be said within academia for the other subfields of computer science, and—though these are beyond what I have extensive contact with—the other sciences and all of research-oriented academia. In some sense, the process of becoming a professor at a research-oriented school preselects people lucky enough to have found a personal most important intersection in what they are devoting their lives to.

1.3 In Academia: Teaching

However, teaching may be a different story. Especially at times when class sizes are very large, and when the pressure to bring in large amounts of external funding is great, many computer scientists in research universities may not view teaching their courses as either a joy or an area where they have exceptional talent. That is tragic not just for the professors, but also for the students. When a professor loves teaching a course, that shows through to the students, and even if the area is not initially the students' favorite, the students surely will learn far more about the area in a course passionately taught by a professor who loves the course's design and material than they will from one who does not.

The goal of this article is to present the design of a course that I think almost any computer-science professor who is a researcher would be passionate about teaching, and likely could teach very well. In particular, this article presents the flavor of a course my school offers that is an *undergraduate* version of the type of research-immersion course that Donald Knuth famously taught for many years to Ph.D.-level students at Stanford. Creating a course of this sort at one's school would, I suggest, add a class that professors will be falling over each other to teach, and that they will teach skillfully and well. I'll argue below that this course also tends to draw passion from students, and to reveal to them that their ability to study problems—even extremely hard problems—is far greater than they, or most of their professors, previously realized.

2 An Undergraduate Knuth-Style Research-Immersion Course

2.1 Introduction

In the 1970s and 1980s, Donald Knuth famously taught at Stanford's Ph.D.-program a problem-solving seminar, which in effect functioned as a sort of research-immersion course. Professor Knuth would give a problem to the students that was so open and hard that he himself did not know what the answer was. Then the class in groups would immerse itself in solving that problem, over the course of a few class sessions, with the groups reporting their progress at each class session. In a typical term, the class covered five problems. The stress was on *collaborative* problem-solving by groups, with cross-group intermediate discussions to share progress and information. It was a lovely, valuable approach. These classes were widely viewed by the students as extremely important to their careers—a trial by fire in meeting the unknown, in interacting with groupmates and with friendly-rival groups, and in doing one's best to brush the unknown back a bit. Happily, these courses were documented through a series of Stanford technical reports, each titled (the titles differ only in a hyphen) "A Programming and Problem-Solving Seminar" or "A Programming and Problem Solving Seminar," and coauthored by Professor Knuth and the TA, containing transcripts of the class sessions. Those reports make revelatory reading.[1]

The Stanford problem seminar was such a good idea that other CS departments started offering this type of course. My own department did, in the form of our "CSC 400: Graduate Problem Seminar." Of course, each school had its own taste and twists, and even in those places where the courses started out very much like the Knuth course, over time the nature of the course often shifted. My own department, pressed by the desire of many to have students start on research with their faculty advisor as early as possible, has made the main research-time focus of our graduate problem seminar course be an *individual* research project that is usually supervised by the student's advisor. That of course is valuable, but is quite different from the core focus on *collaborative* problem-solving of the Knuth course. Our graduate problem-solving course has many additional goals assigned to it, such as teaching students to avoid plagiarism. In some years, it has been used as a place to introduce all first-year Ph.D. students to the department's specialized hardware toys.

In the mid-1990s, we decided to create an *undergraduate* problem seminar, and some of us wanted to model that course closely on not our own graduate-level version but rather on the marvelous course that Professor Knuth had taught at Stanford, and in particular with a very strong stress on collaborative solving of open research questions—often questions so hard that the professor has little hope of a full "solution" being reached (rather, the hope is merely that the class will find a way to make some progress or gain some insights).

Having taken Professor Knuth's course myself, I was very enthusiastic about this approach, as were a number of my colleagues. Over the years, I have taught this course thirteen times—the first being 1997 and the most recent being 2016—the most of any of my department's faculty members. It took quite a lot of pleading to get the assignment that often, because the course is so much fun to teach that many of my colleagues also want to teach it. Each person teaches it in his or her own way and indeed to his or her own design, and that is a good

[1] However, getting to the reports can be a bit difficult. If one knows the report's number, one can find the report by digging down by year and then number within http://i.stanford.edu/pub/cstr/reports/cs/tr/. For example, the 1989 report numbered 1269 (report STAN-CS-89-1269) is available online at http://i.stanford.edu/pub/cstr/reports/cs/tr/89/1269/CS-TR-89-1269.pdf. That report contains the years and numbers of the seven previous technical reports, which are 77-606, 79-707, 81-863, 83-989, 83-990, 85-1055, and 87-1154. The reports also are generally available on archival microfiche copies at Stanford's library, e.g., search for "Programming Knuth Seminar" at https://searchworks.stanford.edu.

thing. But as colleagues we have of course shared our materials when the person coming after us wished that input. In particular, much credit is due to my dear friend and colleague, Professor Chris Brown, who taught the course the first times it was offered, and then very generously shared with me his meticulously crafted collection of course material.

In my offerings of the course, I have always tried my very best to make the experience be as close as possible to the insanely high expectations and open-research-problems flavor of Professor Knuth's graduate course. Indeed, a student who took both our graduate problem-solving course and our undergraduate problem-solving course commented that the undergraduate version is by far the more demanding.

2.2 Components: Overview

The components that my undergraduate research-immersion course has had over the years, not all being part of each year, are the following.

- The projects.
- The crazy-literature critiques.
- The Computer Science Gladiators Day and the Lightning Research Days.
- The one-on-one meetings.
- The quizzes.
- The guest lectures and the readings.

The following sections focus in turn on these components, giving actual examples for those cases where examples are important in understanding the flavor of the component.

2.3 Component: The Projects

2.3.1 Flavor

The Flow of Each Project. Each project spans four class sessions. At the first I present the project—ideally a problem that is as new and open to me as it is to the class—and divide the class into groups. The ideal number of groups is four, and the ideal size of groups is 3–4, with 5–6 being livable if the course size requires it. When the class is too large for even this, one can make 6–8 groups, and each of what usually would be class sessions 2, 3, and 4 of a given project becomes two class sessions, each with half the groups presenting; so each project then would span seven class sessions. However, for the rest of this section, I will generally assume the case of four groups and four class sessions per problem.

The group names are typically fun. For example, in many years we have gone with Gryffindor, Hufflepuff, Ravenclaw, and Slytherin [21]. More recently, the Great Houses from *A Game of Thrones* [18] have been fun for the students. They will often have the house sigil and the house words on their slides, and will work in clever plot allusions. And if a group starts piping "The Rains of Castamere" through the room's speakers, beware!

At the second and third class sessions, each group gives an informal presentation, on slides, of its progress to date, and the other groups, the TA, and I ask questions to the presenting group. This is meant to simulate real-world computer-science workshops, where researchers studying a subarea or problem meet to share progress and ideas.

The fourth class session models a computer-science conference session, complete with a session chair, strict time-keeping warnings, and so on. Each of the four groups on that day turns in a formal research paper and gives a polished, formal 18-minute (as the class time slot is 75 minutes) PowerPoint or Beamer conference talk on their paper/project.

Thus each project provides a time-compressed microcosm of what professors do in a research project.

The Time Needed Between Sessions. The spacing of the sessions is important to get right. Typically, the (three) gaps between sessions 1, 2, 3, and 4 will be at least one week each, and sometimes (especially between sessions 3 and 4) one and a half or two weeks. That gives the students enough time to actually perform their investigations and make progress. Any less time risks demoralizing them. Indeed, even at 1–2 weeks between meetings, that still is a brisk pace. Not many CS professors would want to be expected to make progress and have a new slide show on their progress every week or two!

Instilling Self-confidence in the Students. Students are often at first surprised that the course expects them to tackle open issues—that they are not sitting in a room and being lectured to and taking a midterm and a final on the content of the lectures. But they truly rise to the challenge of researching hard, open issues. It is sort of a *Field of Dreams* thing: If one lets students know that one expects hard work and creativity from them, and that one truly believes they have the ability to have insights even on hard problems, they remarkably often come through brilliantly. And I do believe in students because for more than 20 years I've seen the students truly grow and flourish, even within a single term. It is an honor and treat to be a part of that process.

But how does one convey that belief and confidence in them *to* the students, and let them know that they should share it? How does one convince them that such purported confidence is not just empty words?

A key way to let the students know that they can do far more than they might at first realize is that I explain to them at the start of the course that in traditional courses the professor tends to be an expert on the subject and has taught it many times before, but for the students the material of traditional classes is all new. So in traditional classes the playing field is extremely unlevel, and thus it is hardly surprising that students may feel intimidated by the apparently effortless command of the subject that the professor (sometimes) demonstrates. I point out to them that when the professor was sitting in classrooms 5 or 10 or 30 years ago, the then-student now-professor may well have felt just as intimidated by the command shown by the then-professor as the students themselves do now in their traditional classes. I then point out to them that in this undergraduate problem-solving course, the problems they are looking at are new and open—to them, but equally so to the TA and to me, and to the best of my knowledge to the entire world. The students, the TA, and I are partners in investigating these new questions—we are all peers in the investigation.

Relatedly, I let them know that, every year that I teach the course, I am impressed and humbled by the speed, energy, and insights of students in the class. And lest they think this implausible, I mention, early on, the outright glowing superiority of students in displaying outputs graphically and in writing simulations and experiments; the students know very well that they do these things far better than the typical faculty member, and so this rings true.

And continuing that, I often describe to them some of the most interesting projects done in previous years; and how the direction of one of those projects was so exciting that a few years later RPI was trying to capture the rights to other attacks on the problem; and how each of the class's groups brought wonderful approaches to the problem; and how in most years many students in the class put papers into the arXiv.org Computing Research Repository, and often in their papers refute previous papers in the archive that made overbold claims; and how some students have even produced field-advancing refereed journal publications from the course. In brief, I make it clear to the students that they can very reasonably believe that they can do exciting, valuable research in the course, because in previous years the students who started the course with precisely the same set of worries and self-doubts did do wonderful things.

And so the students do feel confidence, and they do excel. I delight in sharing with my colleagues what clever approaches and results the students are discovering in the course.

Of course, to help this all happen, it is important to choose the projects to be—though open and hard—ones where a variety of skills can be employed. That way, students with strengths as varied as CS theory, mathematical modeling, programming, heuristics, simulations, and visualization can plausibly each find their own way to contribute to their group's attack on the problem.

The Range of Topics. As a professor, there is a great temptation to focus the course all in one's own area, and there have been some offerings of the course that had that flavor. (I'm reminded of a professor at one of my earlier schools, who when assigned to teach the graduate complexity course spent the entire term teaching the professor's own quite different material, and then during the last week or two skimmed through all of complexity. The local joke then became—except the name has been changed here—that there was never any question what course Bill would teach... just what number it would have.)

Myself, when teaching this course I spread the project's topics over the department's three main focus areas. For example, in the first years I taught the course, we would have one project each in AI, systems, and theory. (In years when there are four groups in the class, and thus four class sessions per project, I usually have three projects plus a separate honors project—aka the crazy-literature critique—that does not itself take any class sessions.)

However, in my more recent offerings, that has changed a bit. I have the first project be math- or theory-inspired. And the second project is either AI- or systems-inspired. For the third project, I let each group choose its own project. Doing so lets the group focus on something the group itself is very excited about exploring. It also forces the group's members to negotiate among themselves, comparing skills and interests to find a project that works for the group as a whole.

For that third, self-chosen project, there are two phases. In the first phase, the students in the group formulate a project proposal, and present it to the class as a talk on slides, and also hand in the proposal on paper (and it is graded as a certain portion of that project's grade, e.g., 20% of the project's grade is due to Phase 1 and 80% is due to Phase 2). The class, the TA, and I then give immediate feedback. Sometimes a listener will point out that the proposed work already exists in the literature; or that it is already in progress at major research groups or companies that are putting in an enormous effort; or that it is just hopelessly ambitious to even get a start on within the time frame; or that the hardware required is beyond what the department can provide. After that, the group is allowed to if it wishes modify its approach, or its project, or even to completely change projects, in light of the feedback it received. (Since the Phase 1 presentations take a day or two, the self-chosen project involves five rather than four class sessions if there are four groups, and eight or nine sessions rather than seven if there are more than four groups.)

The topics the students choose are often utterly thrilling, ranging from natural language processing to video processing, to issues in systems, to algorithms challenges, to machine learning applications.

One group of students, David Klein, Kyle Murray, and Simon Weber, as their self-chosen project studied automated recognition of which programming language a code segment/fragment seen in isolation may be from. They wrote a lovely paper [15], and their work turned out also to be of interest to a group in industry, and that led to contact between that industry group and one of the students.

A Sample Twist. Sometimes, I throw in an unexpected twist. For example, a teaching newsletter to which I subscribe, *The Teaching Professor*, had an article suggesting that in the middle of group projects one should shift some students between groups to model the way technical workers move between companies in the real world. This seemed an excellent idea. However, at the time many tech companies were being *merged* in the real world, so I tweaked the idea to make it fit even better with the current news. In the course, I announced mid-project that Gryffindor and Hufflepuff had been merged and that Slytherin had bought Ravenclaw. There were both technical and cultural clashes as the groups integrated, but in the end, excellent projects emerged—and some lessons about combining approaches.

2.3.2 Project Example: 3-Dimensional Randomized Solitaire Bingo

This particular project was from a year in which the course was so large—31 students—that we had not four but eight groups on each project. This meant that the project involved not four but seven class days, since each of the two "workshops" and the "conference" needed two days, with four groups' talks each day. And so in that term, the course had just two projects during the term, plus the honors project (aka the crazy-literature critique) for those taking the honors version of the course.

And here is the project statement that was handed out on the day I introduced the project to the class, slightly edited and condensed.

Project 1: 3-Dimensional Randomized Solitaire Bingo

2014/1/29

CSC 200/200H: Undergraduate Problem Seminar Spring 2014
Instructor: Lane A. Hemaspaandra TA: Joe Izraelevitz

Due: March 3, 2014, 1:59PM (*even if your group will be one of the ones presenting on 3/5 rather than 3/3*—please note that carefully so you don't get a zero after having done lots of work!!!); also, fyi, the first intermediate workshops on this will be 2/10 and 2/12, and the second intermediate workshops on this will be 2/24 and 2/26.

The University of Rochester's (UR's) President Seligman has a problem. He has been challenged by the Rochester Institute of Technology's (RIT's) President Destler, to provide an evaluation of a certain "game" (called RITtaire3D, and which we will describe in detail below). Being no fool, President Seligman, after inspecting the description President Destler provided him of the game, suspects that in fact the RIT president has in mind some actual product in which a randomly chosen nonfailed "sector" of that product fails at each time step, and when a complete "line" of sectors has (over time) failed the product goes up in flames. (This seems a very bad feature of a product, but not all products are perfect. And apparently President Destler is trying to get a handle on how long one can expect to run the product before it self-immolates.) Nonetheless, filled with pride at the brightness of UR's students, President Seligman has agreed to undertake the analysis and has, of course, delegated it to his most powerful secret resource—the students of CSC200/200H.

Your task (each group's task, that is) is to write (following the rules of the course) a research paper exploring this game and in particular the issues mentioned below. This paper should be written in LaTeX. As mentioned above, we'll call the game RITtaire3D (which stands for 3-dimensional RITtaire).

The game RITtaire3D is played on an N by N by N board (N can be 1, 2, 3, etc.). To be clear and build the N into the name of the game, let us during this project when being careful write RITtaire3D$_N$ when speaking of the N by N by N version. The player during the first step chooses a random cell (each with equal probability, e.g., the player flips a fair N^3-sided die) and puts a marker on it. During the second step, the player chooses a random cell from among all cells that do not yet have a marker on them (each chosen with equal probability, e.g., the player flips a fair $(N^3 - 1)$-sided die), and puts a marker on it. And so on. The game ends when the player has N markers in a row—this can be in the height, width, or depth axes, or can be along any of diagonals that exist if one coordinate of one of the dimensions is fixed, or can even be one of the 4 such (well, 1 such if $N = 1$ as they degenerate there) diagonals that go between diagonally opposed corners of the overall cube-shaped board.

What we are interested in is how long it takes to win. So, in this project, you should study that issue. Note that there are many questions to potentially explore: What is the quickest possible win time? What is the longest possible win time? And, most centrally, what can you prove about the expected (average) win time?

It would be great if you can state and prove crisp theorems. Perhaps you'll completely resolve some issues. If not, perhaps you'll prove upper and lower bounds, and will work hard to make them as strong as possible. Or if on an issue you are failing to get any traction, maybe you'll program up a simulation, and see what insights that gives. Also, note that these issues can be studied in various ways. One issue is what values hold for a particular, fixed N, e.g., $N = 1$ (pretty easy!), or $N = 2007$, or $N = 2$, or etc. Very nice would be to get a precise formula that holds over all N, e.g., "For each $N \in \{1, 2, 3, \dots\}$, the quickest win time in $\mathrm{RITtaire3D}_N$ is N" (which by the way happens to be a true theorem—so you see, "theorem" isn't as intimidating a word as one might think; go prove some!). But if you can't get a precise formula, you might want to deal with the *asymptotics* (e.g., "$\mathrm{ExpectedWinTimeInRITtaire3D}_N = \Theta(BLORT)$," where for BLORT will be some expression in terms of N that you have brilliantly proven to hold in your paper, or perhaps you'll not get a Θ result but will provide $\mathcal{O}$ and Ω results), or perhaps you will find/prove upper and lower bounds on particular Ns. Basically, poke around, do your best to converge to strong results by proving at first what you can, and then strengthening more and more and more. We'll discuss the flavor of this in class a bit on the starting date, and of course, as always, we'll before the presentation day have two intermediate day-pairs (February 10 and 12, and February 24 and 26) during which you share, present, defend, discuss, etc. your ideas/progress/etc. (so half the groups will speak on 2/10 and the other half will speak on 2/12, and similarly regarding 2/24 and 2/26).

Good luck, and do well. President Seligman has put the honor of UR in your hands!

(And a bit of help. Most research projects build on the shoulders of those who have come before. In this case, it turns out that some wonderful researchers—the students of the Spring 2007 CSC200/CSC200H class—have looked at the *2-dimensional* analog of the question we're studying. Although trying to get answers from those people is cheating, you of course may well want to look carefully at their papers on this, and any other related work you can find in the literature. I'm making one or two of their papers available, having previously checked with them that it is ok to share them to help future students, via our web page (they will appear there late this evening or early tomorrow). Of course, and this is *very* important, when your papers draw on ideas/results/etc. from an existing paper, whether theirs or any other, you are ethically obligated to clearly and openly cite the paper you are drawing on and to attribute to it whatever you are using from it. In the case of earlier CSC200 papers (except those that are published through a journal or through arXiv.org, as those should be simply cited in those forms), the right citation is to, as your bibliography entry, list the authors, title, and date of the paper, and call it "unpublished manuscript.")

2.3.3 Project Example: Restricted Domain Dialogue Agents: Pawlicki-in-a-Box

Here is a second project example. It is from 2008, and is of an AI flavor, and indeed a rather standard-ish one aside from it being quite hard to actually do as the role of an Undergraduate Program Director and the questions the system might need to respond to are not really as limited as the project's title might make them seem. Briefly, the students are asked to write a program for an agent that fulfills the advising role of the department's Undergraduate Program Director (Ted Pawlicki, who despite the lighthearted accusations of embezzlement in the assignment, is a treasure of a colleague and would never take as much as a poker chip from anyone... except perhaps at a poker table).

This project's flow was a bit different from the standard flow. As usual, there was a day for introducing the project, and then a "workshop" day of talks, and then another "workshop" day of talks. But a week after the second workshop, instead of moving right to the day of "conference" talks, we had a day where the students could not only demonstrate the software systems they had built, but could put to the test the systems the other groups had built. And in the next class after that, we had the traditional end-of-project "conference"-talk day.

Here is the project statement that was handed out on the day I introduced the project to the class, except slightly edited for clarity and space.

Project 3: Restricted Domain Dialogue Agents: Pawlicki-in-a-Box

2008/3/26	Version 1.1
CSC 200/200H: Undergraduate Problem Seminar	Spring 2008
Instructor: Lane A. Hemaspaandra	TA: Stan Park

Due Date Information. Due April 21, 2008, 4:49PM (that is the due date/time for the paper and for the archive including program/README/makefiles/sample runs/etc.), and (this is the due date/time for your talk slides) April 23, 2008, 4:49PM. Note very carefully the slight departure from our normal process. As usual your group's paper and program must be sent to us by 4/21/4:49pm—see the class info document for full details on how to send things in. And each group's program will be demonstrated/tested by each other group and then the writing group itself during that 4/21 class session. And for the 4/23 class session each group will in that class session give a "conference" presentation on their paper.

The two intermediate discussion dates will be April 7th and April 14th.

Project. UR's President Seligman has a problem. Again! He has proposed expansions that total the better part of a billion dollars. And he had careful financial plans as to how to achieve this. But a new audit shows that things will fall far short, money-wise.

After some energetic checking, the President has found out the key cause of the predicted shortfall. Shortly after a CS faculty member was used to help improve the payroll software (there is no record of which one helped, although the four comment lines among the 150,000 lines of new programming are each signed "TFP," surely some deep code and one that the school's best minds have so far been unable to crack), CS faculty member Thaddeus ("Ted") F. Pawlicki's salary suddenly jumped by a multiplicative factor of well over one thousand. By a complete coincidence, Ted has just announced his departure for a recently endowed chair at the very prestigious University of We-Don't-Have-an-Extradition-Treaty-with-the-USA. Unfortunately, due to remaining deficiencies in the payroll software, if the President refills Ted's position, the new hire must start at no less than Ted's most recent salary.

So the President has reluctantly decided to leave the position open. Through shifting the teaching of some of the department's less rigorous courses to the Interpretive Dance Program and some course-assignment shuffling regarding the remaining courses, the President has managed to handle the course-coverage loss. However, he is deeply worried about the loss of Ted's famed advising skills. So who does he call? Yes, the students of CSC200, who so very recently rose so well to the task of studying the counting of kings in graphs. In particular, President Seligman asks you to write a "Pawlicki-in-a-Box" program. This program must handle the advising of all freshman/freshwoman and sophomore CS premajors. Its goal is to provide a strong level of advice, support, and insight to such students, by fulfilling the conversational role that Ted fulfilled when students walked into his office.

Here is the framework. You do not have to tackle speech recognition or speech synthesis at all. We'll use a totally text-oriented interface. Also, you may assume that this version of Ted doesn't have access to students' advising records (although if you want, your programs can keep files that stay around between runs of the program, and thus allow it to become better and better as it gets more and more experience), and so during the conversation with "Ted," "Ted" will have to get from the student (whom you may assume to be a freshman/freshwoman or sophomore) who has come to see him whatever information "Ted" needs to identify what help the student wants and to provide advice to the student on that issue. In particular, when the program starts up, we'll assume that a student has just walked into "Ted"'s office. Your program should then (always) greet the visitor by printing out the formulaic two lines:

```
> Hello.  What can I do for you today?
> (over)
```

(Note: Every line "Ted" outputs must start with the > character and then a space. This will make it easier to visually scan the conversation logs.) More generally, both "Ted" and the student will signal that they are done with what they are currently saying by having a line of input that simply says "(over)" in the student's case and "> (over)" in "Ted"'s case (and the student is allowed to choose to interchangeably use "(over)" and "(o)" and "."—the student might often choose to type "." as it is faster and easier). Note that the student in reply might not give "Ted" all the information "Ted" needs, and might not even pose any coherent issue, and so "Ted" may have to enter into a dialogue with the student. As to what that dialogue might look like and how long it might be, well, as students, you'll have a general idea of that. (Among the types of issues students might naturally come to "Ted" for are issues of what courses to take, prerequisites, getting ready to declare the major, summer jobs, courses, and much more.) And in the 4/21 class you'll have a chance to show off your program's strengths—and to have your program put energetically to the test by other groups.

There will be a second set of special phrases that when on a line by themselves by convention will indicate that the student is leaving "Ted"'s office; "(over and out)" and "(oo)" and "bye" may be used interchangeably for that. "Ted" will never end a conversation on its own; only the student can invoke those session-ending key-phrases. (All of these key-phrases are case-sensitive, by the way. "(OO)" on a line by itself is not a valid sign-off but rather is just a strange parenthetical exclamation to "Ted.")

Of course, writing a perfect such program is (currently) pretty much impossible. In fact, even doing it nondisastrously is a wildly daunting task that is probably also pretty near impossible and would fill any senior AI researcher with terror. (So be very thankful that you are not senior AI researchers; no one likes being filled with terror. In fact, naïveté can be an asset: Róbert Szelepcsényi, as an undergraduate, resolved—in the affirmative—the huge open issue of whether the context-sensitive languages are closed under complementation probably largely because he didn't know enough to know that the problem was hopelessly hard... so he just went right ahead and solved it... correctly!) However, you'll want to do your best to have your program work as well as possible.

Good luck, and do well. President Seligman thanks you in advance for saving his big-chunk-of-a-billion dollar plan (and the rumor that he is kicking back a certain percentage of the savings to your course's professor is, most unfortunately, quite untrue).

2.4 Component: Crazy-Literature Critiques

In years when the course has both some students taking it as a nonhonors course and some (in the same room at the same times) taking it as an honors course, I give an extra assignment to the honors students. In particular, I break them into groups, and each group is asked to write a critique of a seriously-intended recent paper, usually from arXiv.org, that claims to resolve a major open issue in computer science, such as P versus NP.

Claiming that the students will succeed in that, and within just a few weeks, may sound like magical thinking. However, I have so far had twenty-four undergraduate students, in the course, attempt to meet the challenge of refuting a paper published at arXiv.org, and none have failed to meet the challenge; those refutations have all appeared on arXiv.org [5–7,9,11,12,20,22], and most are now cited on the field's central resource for tracking the status of attacks on the P versus NP problem, namely, the web page maintained by Prof. Gerhard Woeginger [27].

That is why I titled this component "crazy-literature critiques." The critiqued papers are, as I said, seriously intended. But often they are, well, a bit wild and hard to follow. (As students quickly learn, a crisply, clearly written, incorrect proof is a piece of cake to critique relative to critiquing a confused, unclear, handwaving paper.) And so far, none have been correct.

The students and the literature both benefit from these critiques. The critiques do the literature the service of validating (never so far) or shattering (always so far) claimed resolutions of the field's major problems. The students, in finding what the flaws are in a paper—typically by finding bright-line counterexamples to shatter algorithms claiming to show $\mathrm{P} = \mathrm{NP}$ and by finding incorrect, tacit assumptions that are used in proofs claiming to show $\mathrm{P} \neq \mathrm{NP}$—increase their own critical reading skills, and so when checking their own future papers they will be more demanding critics and better proofreaders.

2.5 Component: Computer Science Gladiators Day and Lightning Research Days

2.5.1 Computer Science Gladiators Day

I often have a Computer Science Gladiators Day (named after the old, and briefly revived, "American Gladiators" television show). On that day, I invite in two of our CS Ph.D. students who as a team compete with the entire class, which itself works as a team of the whole. I assign a challenging problem, usually in the subarea in which the invited graduate students are experts, that has never been seen before by the graduate students or the class members.

The graduate-student team and the team of undergraduates (which is the entire class) spend about two thirds of the class separately tackling the problem. Then the graduate students present their solution and the undergraduates present theirs.

The undergraduates know ahead of time that this counts as a quiz, and I guarantee that if their solution is at least as good as that of the graduate students, then they all get 100%, and if their solution is weaker than that of the graduate students, they will get a grade based on my judgment of the quality of their solution considered in light of the difficulty of the problem.

The way this most typically ends is that the undergraduates as a team do as well as or better than the graduate-student team, and they can (having heard both solutions) clearly themselves see that they did do that well. The Computer Science Gladiators Day is not just an adventure for the students, but also supports the theme mentioned earlier of instilling self-confidence in the students:

The day lets the undergraduate students know in the clearest possible way that they have tremendous talent—talent so strong that they can, when they work cooperatively, fully hold their own against a team of Ph.D. students who are working on a problem in their own area of specialization.

2.5.2 Lightning Research Days

Each term typically has just one Computer Science Gladiators Day, but has several Lightning Research Days. These are quite similar to Gladiators Day, except there are no graduate students. Instead, at the start of the class I present a challenge (usually one with an answer that is known to me but not known to the class, and which is a problem that I think the class will never have seen before), and the students study it for most of the class in their groups, and then either I have each group present its solution, or I have the groups then meet with each other and choose a "best" of the solutions as the one the class as a whole will be going with and will present to me during the final portion of that class session (and that is what the class is graded on if this is being counted as a quiz). (This is a simplified version of a wonderful approach Prof. Edith Hemaspaandra uses in her courses, and that I have incorporated in some of mine. She starts with students working on a problem in groups of size 1, and then following some timetable and pattern, merges them into bigger and bigger groups as the class session proceeds, with the group-size sequence often being the powers of two.)

2.5.3 Computer Science Gladiators Day/Lightning Research Days: Example Problems

Here are a few examples of some of my favorite problems to use on these one-class problem-solving challenges.

Apportionment Algorithms. Motivated by the fact that the US House of Representatives is reapportioned every decade based on census data, I pose to groups the problem of developing an apportionment algorithm. For the apportionment problem, the input is a set of states, the population p_i of each, and the "house" size H (the total size of the legislative body that is being elected), which currently is 435. And the students must assign to each state a natural number (one may not use fractional values) that will be that state's number of representatives, and the sum of those numbers must be exactly the house size H.

I mention to them, to keep things clean, that it—unlike the actual situation in the USA—is legal for their algorithms to assign 0 seats to some states. And I warn them not to overly focus on "tie-breaking" issues, e.g., if there are exactly two states, and they are of the same size, and the house size is odd.

I also mention that the proportional quota of each state j, namely

$$q_j = H\left(\frac{p_j}{\sum_i p_i}\right),$$

may be of interest to them.

I urge them, as they design their algorithms, to think about what properties their algorithms "should" be, or are, satisfying, and what it even may mean for an apportionment algorithm to be "fair."

I let them know who their historical competition is, namely, many of the existing algorithms were designed by the greatest figures in the USA's history. Among the classic algorithms for this problem are Jefferson's Method (the third president), Adams's Method (John Quincy Adams, the sixth president), Hamilton's Method (the first Secretary of the Treasury), and Webster's Method (yes, the dictionary guy, and US senator). The first presidential veto ever cast, namely by George Washington himself, was over the issue of apportionment methods.

The students are thrilled and amazed to hear that the founders of the nation designed and debated the performance of algorithms... and that the students will be trying their own hand at that same challenge. Since they know that the US President is chosen through the Electoral College, and that each state's number of votes in that is the sum of its number of senators (always 2) and its number of representatives in the House, the students realize that this algorithmic challenge is tied not just to the House's apportionment but also to how much influence states have in choosing the president.

The students very often rediscover one of the classic methods, almost always that of Hamilton. Hamilton's Method (we'll here ignore two special rules that apply in the US House, such as having to give each state at least one representative) gives each state, right up front, the floor of its proportional quota, q_i. And then it adds up all the chopped off fractional parts, which of course adds up to some whole number, say K, and it sorts the states by how large their floored-away fractional parts were, and gives the K states that lost the biggest fractional parts each one extra representative. This method, as the students almost always note and convey, "respects quota," i.e., the number of representatives a state is given is always strictly less than one away from its quota. Hamilton's Method in fact was used for a period of time in the USA as the apportionment algorithm for the House of Representatives.

I then explain to the students the other classic methods—all of them are so-called sliding-divisor methods, variously based on floors, ceilings, and arithmetic/geometric/harmonic means. (The USA currently and for many years has used a geometric-mean-based sliding-divisor method, known as the Huntington–Hill Method, to apportion the House, after it was supported in the first third of the 1900s by two blue-ribbon panels of mathematicians [17].) I point out to the students that none of these other methods always respect quota. Yet I explain why Hamilton's Method may not be as wonderful as it at first glance seems: We see that cases exist where if all the state sizes stay the same, and we increase the house size by one, some state can *lose* a seat. This is known in the literature as the Alabama Paradox.

So, within a class, the students have learned that some of the nation's founders were algorithm designers, have tested themselves against those founders and found themselves their equal, and have learned a bit of rather cool history. In this section, I have including almost no literature citations. Let me instead mention, as I do to the students, that there is an utterly wonderful, compellingly readable book on everything mentioned above, namely, the book *Fair Representation: Meeting the Ideal of One Man, One Vote*, by Balinski and Young [4]. The book weaves mathematics, algorithms, and history into a riveting tale, and one that remains important to this day.

Before ending the class, I turn this all a bit on its head, by introducing the notion of power indices, and giving examples showing that, due to that notion, the connection between number of representatives and power is far more subtle than one might think, and thus that trying to match votes closely to proportional quotas may not be the right goal to have (see [13] and the references therein for more background and empirical studies in this direction).

SAT Is in Probabilistic Polynomial Time. Even students who have not yet taken a Hromkovič- [14] or Sipser-type [25] course can typically enjoy this problem. For this challenge, I define what the famous NP-complete satisfiability problem, SAT, is. And I informally define what the complexity class PP is [10,24], namely, the class of (language) problems that have a polynomial-time algorithm (that is allowed unit-cost access to a fair coin) that on each input is right with probability strictly greater than 50%.

The students' challenge is to, during the class session, prove that SAT $\in$ PP, i.e., that there is a probabilistic polynomial-time algorithm for SAT. This is a lovely result of Gill from the 1970s [10] (that has since been much strengthened).

Students are a bit surprised to hear that SAT—and even if they have not seen its definition before they usually know that things that are "NP-complete" are thought hard—is in polynomial time if one gives one's computer access to a coin. But more often than not, they succeed in proving this themselves. We typically end the class session by discussing why this result cannot be used, through repeated sampling and then taking the majority, to get a killer heuristic algorithm for SAT; in particular, they note that in their algorithm, though its success probability is always strictly more than 1/2, the probability can decay toward 1/2 from above exponentially quickly, and so the type of heuristic just mentioned would have to sample so very many times that one might as well solve the problem exactly in deterministic exponential time by a brute-force approach.

One Hundred Prisoners and a Lightbulb. This is an absolutely delightful problem, in that at first it seems outright impossible that a solution exists at all. And it takes an "Aha!" moment for the groups to realize that it can indeed be solved. I won't describe the problem here, but rather will simply give the reference for the work on this by Dehaye, Ford, and Segerman [8], which can currently be accessed easily as a pdf from the web page of its final author. In most years, the students find the algorithm/approach that those authors call "Strategy 2," and the students prove that the algorithm's expected time is quadratic in the number of prisoners.

Since this problem has the flavor of a distributed systems protocol, I often use it for the Computer Science Gladiators Day, inviting systems Ph.D. students in to be the graduate team that challenges the class.

2.6 Component: One-on-One Meetings

During the first half of the term, I schedule one-on-one meetings with each student in the course. The meetings check on how the student is doing, check on whether the groups are functioning well as to sharing the work load and communicating with each other, and answer any questions the student may have. With graduating seniors, I ask about how things are looking as to jobs or graduate-school admissions. With students earlier than that in their time here, I (a) ask whether they are interested in going to graduate school in computer science, and if so, give them my best advice as to how to go about that, and if they are theory people give advice about schools, and if they focused on another area, I make sure they know what faculty member to consult with for expert advice on how schools are in that area; (b) discuss their plans for summer jobs, and (c) discuss their path to completing their CS major.

My guess is that most students think that these meetings are simply a chance to chat with a professor one-on-one about the course and about their career plans. But there is a more important purpose to the meetings. Earlier on, I mentioned the importance of instilling self-confidence in the students, and described some approaches. These meetings are another approach to supporting that. After all, students differ, and addressing the differing worries and talents of each student can't be done just in broadcast mode to the entire class. In contrast, these one-on-one meetings with each student in the class create a human connection between me and the student. The student can in a one-on-one setting get a far better reading of whether his or her impressions of the professor are for real, of what the professor really thinks of the student, and of whether the professor sincerely believes in that student's potential.

2.7 Component: Quizzes

The Gladiator and Lightning Research Days, as described above, often count as quizzes. As to project-related quizzes, most students would work hard even without quizzes. But to help inspire the few others to stay on top of the projects' material so they can pull their weight in their groups, classes sometimes start with 2–15-minute quizzes, sometimes easy ones and sometimes a bit more challenging.

For example, at the very start of class during the first intermediate workshop on the "3-Dimensional Randomized Solitaire Bingo" project above, I give the following two-question quiz.

Question 1 [50 points]. Consider the 2-dimensional version of RITtaire$_N$. For the $N = 4$ case, give a board with 12 filled squares such that there still is no N-in-a-row (of filled squares) on the board.
Question 2 [50 points]. Now, argue that for the 2-dimensional version of RITtaire$_N$, for the $N = 4$ case, every way of filling 13 squares does result in at least one N-in-a-row of filled squares. (Hint: Think of how many squares

remain unfilled. Can you argue something from that?) (Note: From this question and the one above, you have now shown exactly how long the game can go for $N = 4$ in the longest case.)

The quiz is basically helping the students realize what the longest possible length is of a 2-dimensional solitaire bingo game (and in doing so, the question is quietly suggesting, as did the project statement itself, that their papers surely should look at the same issue in 3 dimensions—and almost every group did, not finding a tight bound but in most cases finding relatively close upper and lower bounds for the 3-dimensional case). In the 2-dimensional case that the above quiz was about—i.e., an $N \times N$ board—except for $N = 2$, one can always fill $N^2 - N$ squares without having N-in-a-line on any row, column, or major diagonal. That is optimal: Every way of filling $N^2 - N + 1$ squares yields a bingo (i.e., N filled squares in some row, column, or major diagonal), since with one more square filled, the average number of filled squares per row is strictly greater than $N - 1$, and so at least one row must have N filled squares in it. For $N = 2$, the most filled squares one can have without a bingo is not $2^2 - 2$, but rather is 1. The reason $N = 2$ is pathological has to do with the example that realizes $N^2 - 2$, which is simply to fill every square on the board except to leave empty the entire top-left to bottom-right major diagonal, and then if N is an even number adjust that board by rotating the 4 center-of-the-board cells by 90 degrees. That rotation is to block the other major diagonal, and works fine for all even N except 2, where the rotation would reopen the originally blocked diagonal. Pictorially, here is an 8×8 board with $8^8 - 8$ filled squares yet with no bingo existing:

	●	●	●	●	●	●	●
●		●	●	●	●	●	●
●	●		●	●	●	●	●
●	●	●	●		●	●	●
●	●	●		●	●	●	●
●	●	●	●	●		●	●
●	●	●	●	●	●		●
●	●	●	●	●	●	●	

2.8 Component: Guest Lectures and Readings

The guest lectures and the readings are listed in the same component since they share the same goal: showing students how widely varied are the ways that actual researchers approach research.

As mentioned above, there are usually one or more nonproject days between the project days. Those "open" days are used for the Gladiator and Lightning Research Days, and for guest lectures. Typically, I'll ask one to three people from each of the department's three main areas to give guest talks on their research. When there is time, I'll invite a faculty member from a related department (such as the Department of Electrical and Computer Engineering or the Department of Brain and Cognitive Sciences) to speak, or I'll invite a graduate student to speak. Once or twice, I've invited a very research-successful undergraduate student to speak.

In years when I want to (or due to class-time limitations need to) supplement or partially replace that with readings, I have the students read articles by or about the greatest research minds in computer science's history. My two favorite sources for such readings are the Turing Award Lectures (the early ones are even collected as a book [1]) and the wonderful book *Out of Their Minds: The Lives and Discoveries of 15 Great Computer Scientists* [23].

My goal, in having those talks and assigning those readings, is not to show the students that there is one way to do research. It is to show them that successful researchers fit no mold (aside from usually being passionate about their research). Even within a given area of the field, researchers have different views, approaches, and work habits. My hope is that some of those will resonate with the students, and that each student can enrich his or her own approach to research with those approaches that he or she hears or reads that speak to him or her. Of course, the guest talks also are about introducing the various areas of CS to the students; but to me that isn't the talks' most important function.

My very favorite reading from the above is without any doubt Ken Thompson's three-page Turing Award Lecture, "Reflections on Trusting Trust" [26], which (among other things) basically shows how to push malicious code into such a low level that recompiling programs and or even recompiling the compiler itself doesn't protect one, as the executable that is running when one calls the compiler has itself been compromised in a subtle and self-propagating way.

My favorite reading not of the above two types—Turing Award lectures and chapters from the book *Out of Their Minds*—is the breathtaking article "Cheating Husbands and Other Stories: A Case Study of Knowledge, Action and Communication," by Moses, Dolev, and Halpern [19]. Like the Thompson paper, this article is so clever, brilliant, and surprising that after one gets past feeling awed and inadequate, one squares one's shoulders and tries one's best to do work that displays at least some hint of the sparkling beauty that one finds revealed in those papers.

3 Conclusion

This article has sketched the design of a Knuth-style undergraduate research-immersion course, and provided some examples of projects and problems. The hope is that this article interests some faculty members elsewhere in creating such a course at their own schools, and if so that this article will help them in bringing such a course to life.

The course, far more flexibly and broadly than most course frameworks, has the potential of creating a class that for many faculty members will, after they resculpt it to best match their own style, be located in that most important intersection—the intersection of what they love doing and what they can do exceptionally well. And in this course, there typically is a reinforcement cycle between the professor loving and passionately leading the course, and the students loving it and excelling in it. Sometimes, things all simply come together.

Acknowledgments. My thanks to Hans-Joachim Böckenhauer, Dennis Komm, and Walter Unger, who kindly invited me to participate in this project honoring the inspirational Juraj Hromkovič; to Chris Brown, who first taught my department's undergraduate problem seminar and generously shared with me his excellent materials; to Edith Hemaspaandra and an anonymous reviewer for many helpful suggestions and corrections; to my former department chair, Henry Kautz, who a few years ago suggested I write an article on my version of our undergraduate problem seminar; and to Donald Knuth, whose Autumn 1981 Programming and Problem-Solving Seminar [16] I had the treat of taking, and whose teaching of the course was a model of being in "that most important intersection."

References

1. ACM Turing Award Lectures: The First Twenty Years. ACM Press Anthology Series, Addison-Wesley, Boston (1987)
2. Alexander, L.: The Chronicles of Prydain (The Book of Three, The Black Cauldron, The Castle of Llyr, Taran Wanderer, The High King). Holt, Rinehart, and Winston (1964–1968)
3. Alexander, L.: Taran Wanderer. Holt, Rinehart, and Winston (1967)
4. Balinski, M., Young, H.: Fair Representation: Meeting the Ideal of One Man, One Vote. Yale University Press, New Haven (1982)
5. Cardenas, H., Holtz, C., Janczak, M., Meyers, P., Potrepka, N.: A refutation of the clique-based P = NP proofs of LaPlante and Tamta-Pande-Dhami. Technical report arXiv:1504.06890 [cs.CC]. Computing Research Repository, April 2015
6. Christopher, I., Huo, D., Jacobs, B.: A critique of a polynomial-time SAT solver devised by Sergey Gubin. Technical report arXiv:0804.2699 [cs.CC]. Computing Research Repository, April 2008
7. Clingerman, C., Hemphill, J., Proscia, C.: Analysis and counterexamples regarding Yatsenko's polynomial-time algorithm for solving the traveling salesman problem. Technical report arXiv:0801.0474 [cs.CC]. Computing Research Repository, January 2008
8. Dehaye, P., Ford, D., Segerman, H.: One hundred prisoners and a lightbulb. Math. Intell. **24**(4), 53–61 (2003)
9. Ferraro, F., Hall, G., Wood, A.: Refutation of Aslam's proof that NP = P. Technical report arXiv:0904.3912 [cs.CC]. Computing Research Repository, April 2009
10. Gill, J.: Computational complexity of probabilistic Turing machines. SIAM J. Comput. **6**(4), 675–695 (1977)
11. Hassin, D., Scrivener, A., Zhou, Y.: Critique of J. Kim's, "P is not equal to NP by modus tollens". Technical report arXiv:1404.5352 [cs.CC]. Computing Research Repository, April 2014
12. Hemaspaandra, L., Murray, K., Tang, X.: Barbosa, uniform polynomial time bounds, and promises. Technical report arXiv:1106.1150 [cs.CC]. Computing Research Repository, June 2011

13. Hemaspaandra, L., Rajasethupathy, K., Sethupathy, P., Zimand, M.: Power balance and apportionment algorithms for the United States Congress. ACM J. Exp. Algorithmics **3**(1), 16 (1998). http://doi.acm.org/10.1145/297096.297106
14. Hromkovič, J.: Theoretical Computer Science: Introduction to Automata, Computability, Complexity, Algorithmics, Randomization, Communication, and Cryptography. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-11120-4
15. Klein, D., Murray, K., Weber, S.: Algorithmic programming language identification. Technical report arXiv:1106.4064 [cs.LG]. Computing Research Repository, June 2011
16. Knuth, D., Weening, J.: A programming and problem-solving seminar. Technical report STAN-CS-83-989 (aka STAN-B-83-989). Department of Computer Science, Stanford University, Stanford, December 1983. http://i.stanford.edu/pub/cstr/reports/cs/tr/83/989/CS-TR-83-989.pdf
17. Malkevitch, J.: Apportionment I. FEATURE COLUMN: Monthly Essays on Mathematical Topics (an AMS Web Column), May 2002. http://www.ams.org/publicoutreach/feature-column/fcarc-apportion1. Note: The part giving the history of apportionment in the USA, and mentioning the two blue-ribbon panels of mathematicians, is Section 2. http://www.ams.org/publicoutreach/feature-column/fcarc-apportion2
18. Martin, G.: A Game of Thrones. Bantam Books, New York City (1996)
19. Moses, Y., Dolev, D., Halpern, J.: Cheating husbands and other stories: a case study of knowledge, action and communication. Distrib. Comput. **1**, 167–176 (1986)
20. Richardson, A., Brown, C.: A critique of solving the P/NP problem under intrinsic uncertainty. Technical report arXiv:0904.3927 [cs.CC]. Computing Research Repository, April 2009
21. Rowling, J.: Harry Potter and the Philosopher's Stone. Bloomsbury Publishing, London (1997). American edition published under the title Harry Potter and the Sorcerer's Stone
22. Sabo, K., Schmitt, R., Silverman, M.: Critique of Feinstein's proof that P is not equal to NP. Technical report arXiv:0706.2035. Computing Research Repository, June 2007
23. Shasha, D., Lazere, C. (eds.): Out of their Minds: The Lives and Discoveries of 15 Great Computer Scientists. Springer, New York (1995)
24. Simon, J.: On some central problems in computational complexity. Ph.D. thesis. Cornell University, Ithaca, January 1975. Available as Cornell Department of Computer Science Technical Report TR75-224
25. Sipser, M.: Introduction to the Theory of Computation, 3rd edn. Cengage Learning, Boston (2013)
26. Thompson, K.: Reflections on trusting trust. Commun. ACM **27**(8), 761–763 (1984)
27. Woeginger, G.: The P-versus-NP page. http://www.win.tue.nl/~gwoegi/P-versus-NP.htm