Computational Thinking: Building a Model Curriculum

William A. Booth
Dept. of Computer Science
Baylor University
Waco, TX 76798, USA
bill_booth@baylor.edu

Greg Hamerly
Dept. of Computer Science
Baylor University,
Waco, TX 76798, USA
greg_hamerly@baylor.edu

David Sturgill
Dept. of Computer Science
NC State University
Raleigh, NC 27695, USA
dbsturgi@ncsu.edu

Ivy Hamerly
Dept. of Political Science
Baylor University
Waco, TX 76798, USA
ivy_hamerly@baylor.edu

Todd Buras
Dept. of Philosophy
Baylor University
Waco, TX 76798, USA
todd_buras@baylor.edu

ABSTRACT

As computing technology becomes increasingly ubiquitous, the need to understand how computers solve problems and the need to decipher what types of problems are best solved with computational tools is becoming increasingly relevant throughout the academic and commercial fields. This paper describes a computational thinking curriculum development project. The purpose of this one-semester course is to introduce computational thinking to undergraduate students who are not computer science majors. This course was designed to engage a broad group of students, including those not ordinarily accustomed to using computation as a tool. The course includes skills such as problem abstraction and decomposition, understanding fundamental programming concepts, and appreciating the practical and theoretical limits of computation. With these goals in mind, problems from a diverse set of fields were developed to demonstrate how computational thinking could be applied in a variety of academic and real-world problem domains. These sample problems were then used to build a new course in computational thinking targeted at non-computer science majors. After testing and refining the curriculum, the course was evaluated in two instructional settings to establish its effectiveness. This investigation revealed that formal training in computational thinking decreases computer anxiety while increasing the participants' ability to use computational thinking as a problem solving strategy.

General Terms
Curriculum Development

Keywords
Computational Thinking, Curriculum Development, Pedagogy

1. INTRODUCTION

This paper reports on a computational thinking curriculum recently developed through a National Science Foundation (NSF) Pathway to Revitalized Undergraduate Computer Education (CPATH) grant. The purpose of this one-semester course is to introduce computational thinking to undergraduate students who are not computer science majors. This course is intended to engage a broad range of students, including those not ordinarily accustomed to using computation as a tool. This course includes skills such as problem abstraction and

decomposition, understanding fundamental programming concepts, and appreciating the practical and theoretical limits of computation.

There are several distinctive aspects to our course. First, it is intended for all students rather than just those in science-related disciplines. Second, it is collaborative. Students work in small groups that change periodically. Third, it is problem-based. Students learn computational thinking by solving real problems. Finally, problems given in the course come from disciplines outside of computer science.

Faculty from across the university helped to develop a set of problems to be assigned to students. Because the course focuses on how computational thinking may be used in other disciplines, we expect it will better train non-computer science majors than the existing introductory course for computer science majors. We anticipate that investigating computational problems from a variety of application areas will make the computational topics more interesting and will prepare students to think creatively and broadly about how computational thinking might be relevant and effective on problems in their own discipline.

2. COMPUTATIONAL THINKING

Computational thinking has been around since the very beginning of computer science. It was originally known as algorithmic thinking in the 1950s and 1960s [3]. Computer technology has become nearly ubiquitous in the last decade, but this technology holds little utility if people are not able to use it effectively. Jeannette Wing [14] suggests "Computational thinking is a fundamental skill for everyone, not just for computer scientists." She defines computational thinking as"... solving problems, designing systems, and understanding human behavior, by drawing on the concepts fundamental to computer science."

Not only do students in information-based fields like science and engineering require a basic facility with computational tools, but also this is increasingly necessary for students in non-science fields. Professionals in creative arts use technology for design and production, while those in humanities use and search digitized sources (books, documents, images) for content analysis. Information-based questions often arise in business and social sciences, as well.

A divergence between those who create technology and those who use it has inhibited progress in some fields. Often, those who depend on computing technology are unfamiliar with best practices and even what sorts of computing tasks are possible. Of course, not everyone needs to become an expert in underlying technologies to enjoy their benefits; in many cases, a modest degree of training in computational thinking can help to enhance these benefits.

Computational thinking offers an algorithmic approach to questions, problems and tasks. It allows people to leverage technology, such as computer software, to solve problems that are much larger, more complex, or more tedious than what they would be able or willing to solve by hand. It depends on an understanding of the basics of computation, including abstraction (generalization of objects or procedures), decomposition (breaking a procedure into simpler parts), sequence, independence, decisions, functions, input, and output [12]. It also involves understanding the limits of computation, both practically and theoretically, as well as issues of scale.

Those who can think computationally are at an advantage in effectively applying technology. They are able to understand where computational solutions are possible and how these solutions can be applied to reduce human effort. This understanding can improve quality or enable solutions to new problems. They can be an asset to both their field and at the same time expand the field of computing. Kramer [7] and Hazzan [5] discuss how abstraction is a fundamental aspect of computational thinking. Hazzan suggests that two keys to teaching 'Soft ideas' (i.e. abstractions) are teaching explicit models of abstraction, and further that "... students must be active: they must do and they must reflect on what they are doing." In other words, having students apply their knowledge provides a path for engaging them in learning [7, 5].

Pulimood and Wolz [11] seem to agree, and further propose that collaboration is necessary in upper-division computer science education, when they write "Three essential themes emerge: (1) creative design requires an authentic inquiry-based pedagogy, (2) modern problem solving occurs in a collaborative community, rather than in isolation, and (3) problem solving with computers is increasingly a multidisciplinary endeavor." While this perspective focuses on upper-division computer science, collaboration is also important in introductory courses, when students may have a limited view of the potential of computational thinking. Involving students who are studying a variety of fields will enhance the collaborative nature of a course, and reflect a more realistic vision of computational thinking, as it will be needed outside the classroom.

Buckley [1] suggests that computing education can be motivated by a desire to solve problems of society. In other words, rather than identify a computing problem and a computing solution, start by identifying societal problems, and see how computing can address the issues.

Lu and Fletcher [8] suggest that 'computational thinking language' should be integrated into students' learning during their formative (i.e. K-12) education, long before the concept of programming is introduced. We agree that students should be exposed to computational thinking as early as possible, and that computational concepts can be introduced in a multitude of ways, of which programming is one. Since we work in the university setting, we expect that students who have had a formative education that includes computational thinking will be ready to use programming as one part of a course on the topic. Those students who haven't been previously introduced to computational thinking will still benefit from their facility with everyday technology.

3. RELATED WORK

There have been a number of efforts to introduce computational thinking to students who are not computer science majors. Most of these are primarily focused on teaching students in one of the STEM majors.

Christine Alvarado (Harvey Mud), Rubin Landau (Oregon State), and Jennifer Campbell (U. Toronto) discussed their experiences of offering introductory computer science courses to non-computer science STEM students in a SIGCSE 2008 panel [13]. The first two use a problem-based approach. Another example is Cortina [2] who teaches a course in computer science for non-computer science majors which uses no programming. Instead, they study a broad set of computational topics and use a flowchart simulator (Raptor) to study algorithm

behaviors. Pulimood and Wolz report on a course intended to increase collaboration among a broad spectrum of students to develop a computer game [11]. They believe that collaboration and inquiry-based instruction are important when instructing students about computer science.

Qin [12] found that offering a course on computer science to biology students worked best when students were paired and when they worked on concrete problems. Hambrusch et al. [4] offered a course on computational thinking for science majors that use problems from scientific disciplines to motivate computational topics.

Their work is similar to ours, but our approach is much broader, inviting disciplines both inside and outside of STEM. This curriculum development project was developed by drawing on experts from across the university to help identify and developing computational problems form a diverse collection of academic and professional disciplines.

Perkovic et al. [10] introduced a framework for incorporating computational thinking topics across a broad selection of courses in a liberal arts curriculum. This is similar to our work in that it uses computation with areas not typically associated with computer science. While they suggest pushing computation out to many classes with different subjects, our approach is to bring together many disciplines into one course that focuses on computation. These complementary approaches have different advantages. An advantage of Perkovic's approach is that computation is an integrated concept across the curriculum. One advantage of our approach is simplicity of establishing and offering the course.

4. CURRICULUM DEVELOPMENT

This new course was developed through a NSF CPATH grant, over a one-year period. Collaborators were solicited from departments across Baylor University, the University of Mary Hardin Baylor, and Elon University. A key component of this course was the faculty support from external (non-computer science) departments. Faculty who agreed to collaborate began with a one-hour meeting in which the goals for the new course were explained in detail. The collaborating faculty then spent time on their own thinking about what problems in their respective fields would serve as good topics for a course in computational thinking.

This initial meeting was followed by a series of half-day workshops where the collaborators worked to more fully develop their ideas and submit written problem statements. Finally, the collaborating faculty were invited to give a short lecture to the students in the computational thinking course about their content topic and computational problem. This lecture provided background on the problem from the faculty member's domain. In addition to providing specific details about the computational problem, the class would be solving during the weekly lab.

In the second year of grant support, we offered the course for students. During the semester course, the collaborating faculty gave lectures from a diverse set of academic disciplines including political science, engineering, journalism, film and digital media, philosophy, music, and nursing. For each topic, the students were introduced with a traditional lecture and then worked in small groups to create a computational solution to a problem about the topic during a

subsequent lab. Naturally some problems were more difficult to solve and as a result took several lab meetings to arrive at an acceptable computational solution.

One important skill developed during this course was the ability to analyze a problem to determine what strategy would lead to a satisfactory solution. Some problems were solved using existing computer applications such as Microsoft Excel; other problems required the creation of custom applications, which were written in Python. The students did not directly solve the most challenging problems. When the analysis revealed that the problem was too complicated to be solved by a novice, the students wrote a program specification and had that application developed by an expert. They then used the expert's implementation to solve several instances of the complex computational problem.

5. COURSE OVERVIEW

We first introduced students to general concepts in computational thinking. The first week focused on topics like the significance of computational thinking, skills for problem reading (e.g. identifying input, output, and desired out? comes, etc.), introducing a list of canonical problem types (e.g. optimization, summarizing, generation, etc.), introducing problem-solving skills such as problem decomposition, abstraction, and teaching basic concepts of programming in the language deemed appropriate for the course. In the remainder of the course, students mastered computational tools and techniques by progressing through a graduated sequence of collaborative problem-solving experiences.

Each week the students received one or more problems to solve. Our intent was to offer problems that stimulated students' imaginations and creativity, giving them reason to think individually about the problem outside of class. During class laboratory sessions, students worked with classmates to refine their ideas and implement a solution.

The problems are ordered by topic and by difficulty. This allows us to introduce new computational tools as the semester progresses, such as new functional abstractions (e.g., loops, branching, functions) and data abstractions (e.g., arrays, lists, matrices, trees).

Each week, collaborative student groups are shuffled permitting participants to work with peers from many other disciplines. A group is evaluated based on the solutions it produces. They may be evaluated for correctness, efficiency, closeness of approximation, or other criteria. To track the progress of individual students, we aggregate the performance of the groups they have worked in over the semester.

A key part of learning to think computationally is having students develop their own problems based on their interests and disciplines. Thus, we included an opportunity for students to write their own problem descriptions, including proposed solutions. This practice has been used effectively in some Computer Science courses at our university. It helps students solidify their writing and communication skills and gives them a better sense of what sorts of problems can be computationally solved and how well they might scale to larger problem sizes.

Due to space limitations it is not possible to fully describe all the problems covered in this course. We do, however, share two of the problems that seemed to resonate with the students.

The first problem is "Prisoner's Dilemma", which comes from political science. The second problem is "Probability, Testimony and Belief", which comes from philosophy.

5.1 Prisoner's Dilemma

This subsection describes a topic drawn from the field of political science. After an introduction to the topic, we give a simulation-based problem, which students may use to explore the topic with computation.

5.1.1 Introduction

How can rational, selfish actors cooperate for their common good? This is the essential question at the root of many problems in politics and governance. In one sense, the political realm is a "kill or be killed" environment. There are incentives to take advantage of others in order to avoid becoming a victim. At the same time, there are benefits that come with cooperation. How can rational, selfish actors build enough trust in each other to make cooperation possible?

One-way of thinking about this problem is to use the Prisoner's Dilemma (PD) [9]. You have most likely seen the PD situation play out in police procedural dramas, like CSI or Law and Order. Two people have committed a crime together. They have both been arrested and the detectives are interviewing them in separate interrogation rooms. Each suspect is presented with the same information: If you tell us what happened first, we'll make sure your accomplice gets a heavy sentence and the District Attorney will give you immunity from prosecution. If your accomplice breaks down before you do, you will get the blame for this crime and your accomplice will go free. If you both hold your silence, we can still prosecute you for some minor crimes. The detectives are hoping that both suspects will turn each other in at the same time and they will both get jail time for their crimes.

		Player 2	
		Collude	Cheat
Player 1	Collude	20, 20	0, 30
	Cheat	30, 0	10, 10

Table 1: Payoffs for colluding or cheating in the Prisoner's Dilemma game. Within each cell, the two numbers represent the payoffs for players 1 and 2, respectively (higher payoffs are more desirable). "Cheat" means that a player gives up incriminating information about the other player, while "collude" means withholding this information.

Another way to present this dilemma is given in Table 1. In the table, "cheat" means that a player gives up information about the crime that helps punish the other player, and "collude" means that a player withholds such information. If Player 1 assumes that Player 2 is untrustworthy and prone to cheating, then Player 1 can minimize his/her losses by cheating as well. If Player 1 assumes that Player 2 will be faithful to the deal and hold his/her silence (collude), then Player 1 can't help noticing that his/her payoff would be better if he/she cheats (i.e. blames the accomplice for the crime) and Player 2 does not cheat. Player 2 has the same realization. The end result is that both players are tempted to cheat and usually both go to jail.

Under what circumstances would the accomplices resist the temptation to turn on each other?

Researchers have found that in a one-shot version of this game the equilibrium is that both players cheat. In an iterated game, though, it is possible for the players to collude with each other. By playing this game over and over again, the two players can train each other to cooperate.

In the depictions of this situation on television, the suspect that resists the temptation to cheat often mentions that their accomplice has a network of fellow criminals who could punish them even if the accomplice goes to jail.

The Prisoner's Dilemma analogy has been applied to many situations, but one of the most famous applications is to the study of global nuclear strategy (see Table 2).

		USSR	
		Cooperate	Attack
USA	Cooperate	20, 20	0, 30
	Attack	30,0	10,10

Table 2: A hypothetical payoff table for global nuclear strategy.

Both the USA and the USSR have large arsenals of nuclear weapons: In a standoff, both are tempted to attack their opponent using their nuclear weapons. However, each country knows the other country also has access to nuclear weapons. If the USSR can get away with nuking the USA and be assured that the USA would not be able to counter-strike, then there is the potential for the USSR to win the Cold War using nuclear weapons. If neither country can give a strong enough signal that they would counter-strike, then both countries are tempted to nuke each other. If the countries can convince each other that they would definitely fire off a counter-strike, then the equilibrium outcome is for both countries to refrain from using their nuclear weapons on each other.

5.1.2 Problem Statement

Your task is to code an iterated Prisoner's Dilemma simulator where players will take the following strategies:

- Always cheat
- Always cooperate
- Alternate between cheating and cooperating
- Tit-for-tat: Choose to cooperate in the first round. If the other player cheats, punish them by cheating in the next round. If the other player cooperates, reward them by cooperating in the next round.
- Reverse tit-for-tat: Punish cooperation with cheating. Reward cheating with cooperation.

What strategy works the best in one-shot game? In an iterated game?

5.2 Probabilities, Testimony and Belief

This subsection describes a topic drawn from the field of philosophy. After an introduction to the topic, the students are invited to use computation to illuminate Bayesian reasoning.

5.2.1 Introduction

One major topic in philosophy is the evaluation of our beliefs to determine when and under what conditions a belief has a certain merit, like rationality. The rationality of our beliefs has a lot to do with evidence. Rational beliefs are proportioned to our evidence. Determining the weight of a piece of evidence is a very complicated procedure, typically modeled by Bayesian reasoning. The Bayesian model can yield very surprising results, and there is strong evidence that we are not naturally very good at estimating accurately the weight of our evidence. Thinking about the rationality of our beliefs, at this point, is heavily computational, and often involves very large numbers. Usually, with a little instruction, students can work with the Bayesian model on pen and paper. But it is difficult, and, a bit of a distraction from the main philosophical point. In fact, students may well understand the nature of Bayesian reasoning better by turning to computational tools. The important point is not crunching the numbers, so to speak, but understanding how the numbers are interdependent.

5.2.2 Problem Statement

One good way to illustrate the issues surrounding the weight of evidence is to calculate the tipping point for believing something based on testimony of eyewitnesses (or of experts). No matter how unlikely an event may be (provided that the event is not impossible) there are a number of credible independent witnesses that would convince us that the event had occurred, and a point at which it would be positively irrational not to believe that the event had occurred. The computational problem is to fix that number, and to see how it is a function of the improbability of the event in question, of the reliability of the witnesses (or experts) and of their independence. Using Bayesian reasoning, and assigning plausible estimates (which they should be able to explain), students should be able to compute the number of eyewitness (or experts) testimony needed to convince them to believe something, no matter how improbable. Good examples can be drawn from any number of domains. One might ask, questions like these: How many qualified testifiers would it take to convince me that two students composed precisely the same 500 word essay? How many trusted reports would it take to convince me that a man survived a skydiving accident in which his parachute failed to open?

For this lab, you will create a small program that calculates the probability of an event, using Bayes' theorem, given several parameters that you will adjust. You will answer the question: based on reasonable estimates of prior probabilities, how many eyewitnesses would be necessary to convince a rational person that man survived a sky-diving accident in which his parachute failed to open?

6. EVALUATION PLAN

The curriculum was evaluated in two instructional settings to establish its effectiveness. Initially it was used as the curriculum for a new course, Computational Strategies (CSI 3305) Offered for the first time at Baylor University in the 2011 fall semester. A subsequent half-day

computational thinking seminar was also conducted during the 2011-2012 winter break which presented a subset of the modules taught in the semester long course.

We had hoped to evaluate the impact of the course using a mixed method case study. The quantitative portion of the study planned to use a quasi-experimental design. The comparison group would be students enrolled in a first-semester computer science course for technical majors. Historically, 75% of the 200-plus students enrolled in the course are non-computer-science majors. Students enrolled in our new computational thinking course were to be used as the treatment group and a pairwise comparison was to be made with the control group to control for nuisance variables. Unfortunately, despite our best efforts, enrollment in the new course was significantly lower than we had anticipated. For this reason, only anecdotal findings can be reported from the first offering of CSI 3303.

To provide additional evidence of the effectiveness of this new curriculum, a half-day computational thinking workshop was evaluated using two instruments that were administered to all participants both before and immediately after the workshop. The first instrument, the Computational Thinking Problem Solving Inventory (CTPSI), was designed to measure how often students appropriately select a computational strategy, when designing a solution to a problem. This instrument was developed at Baylor University and field tested during the fall 2011 term. This inventory was given to 51 participants. 22 of the participants had no formal training in computational thinking while the remaining 29 participants had at least two years experience in using computational thinking as a problem solving strategy. A factor analysis of the CTPSI revealed that it loaded on a single factor and the instrument's reliability was established with a Cranach's alpha of 0.771.

The second instrument used to evaluate the impact of the half-day workshop was the Computer Anxiety Rating Scale (CARS) [6]. The CARS is a twenty-item instrument designed to measure anxiety associated with computer tasks. Modifications were made to several of the items to allow them to reflect the modern use of technology. The CARS has high internal consistency (a= .87), good test-retest reliability (r = .70), and good discriminant validity [6].

7. RESULTS

Due to the small number of students participating in the first offering of CSI 3303, only anecdotal evidence can be provided. At the end of the semester each student in the class was able to clearly articulate a definition of computational thinking. Additionally, the students' final projects where of high quality and provided clear evidence that the students were able to apply the computational thinking skills taught in this course. Finally, the end of semester course evaluations, reported in Table 3, indicate the students enjoyed the course and felt the curriculum model was effective in teaching computational thinking.

	Strongly Agree	Agree
Assignments contributed to student's learning.	60%	40%
Students learned a great deal from this course.	60%	40%
Used procedures and methods	80%	20%

conducive to learning.		

Table 3: Student course evaluation scores. These are standard. Questions asked after every undergraduate course at Baylor.

To provide additional evidence of the impact of formal training in computational thinking, a half-day computational thinking workshop was conducted in December of 2011. Fifteen participants were recruited to participate in a half-day computational thinking seminar. The participants were undergraduate students at Baylor University, whose major field of study was not computer science, mathematics, engineering, or a lab science. Of the 15 participants, 14 successfully completed both the pre-test and post-test for the CARS and CTPSI. The results of a one-tailed paired t-test are reported in Table 4. These results show that formal training in computational thinking reduces computer anxiety and increases the participants' ability to use computational thinking in problem solving. Both the CRAS and CTPSI showed statistically significant differences between the pre-test and posttest. The average CRAS score, which measures computer anxiety, decreased from 42.14 to 38.79 which was statistically significant with a $t = 1.940$ and a p-value < 0.037. And the average CTPSI score, which measures the participants computational thinking ability, increased from 15.20 to 18.47. This increase was also statistically significant, with $t = 5.215$ and a p-value < 0.001. These findings demonstrate that formal training in computational thinking decreases computer anxiety while increasing the participants' ability to use computational thinking as a problem solving strategy.

	N	Pre-Test	Post-Test	t	p
CARS	14	42.14	38.79	1.940	0.037
CTPSI	14	15.20	18.47	5.215	0.0001

Table 4: Half-Day Computational Thinking Workshop. Computer Anxiety and Computational Thinking. One Tailed Paired t-test.

8. CONCLUSIONS

The course we have developed advances the state of computational education in several ways. It is specifically focused on helping non-majors develop facility in applying computation to real problems in other disciplines, including their own. The principal investigators believe that these types of students are not well served by introductory computing courses intended for computer science majors, and other courses available at the university are not intended to educate students in computational thinking.

The proposed course attempts to correct this deficiency. Rather than focusing on computational approaches themselves, the course material engages students by focusing on a broad sampling of problems from across the university. While a typical student in a non-technical course would not normally have access to computation as a problem-solving resource, students completing this course have an understanding of the fundamental concepts and experience with available tools sufficient to permit them to apply computation to problems they encounter in their own area of specialization.

If technology is to continue having a positive impact in society, it is essential for the people who interact with technology to understand how to effectively use and create with it. These skills are relevant not only to those for whom technology is a focus but also to those for whom it is

simply a means to an end. The proposed course broadens participation in computing because it is specifically intended to equip non-computer scientists with the ability to think computationally and to use computation in problem solving. This course directly benefits those students completing it by enabling them to apply computation effectively and to understand how it can be applied.

Efforts to develop a "problem library" featuring application areas collected from across the university also help build and maintain connections between faculty in computer science and the other disciplines. This effort has broad support from faculty inside the department and elsewhere in the university. Experience in developing this course and its supporting materials benefit a broader community by providing a curriculum, software tools, and a body of problems that can be used in other settings to educate students in computational thinking.

Future work includes dissemination through teacher training and making this curriculum available on the web. We plan to hold training workshops in this curriculum for K-12 as well as university instructors. We also plan to offer the course again soon with a larger class of students.

9. REFERENCES

[1] M. Buckley. Viewpoint: Computing as social science. *Communications of the ACM*, 52(4), 29-30, 2009.

[2] T. J. Cortina. An introduction to computer science for non-majors using principles of computation. *SIGCSE Bulletin*, 39(1): 218-222, 2007.

[3] P. J. Denning. The profession of IT beyond computational thinking. *Communications Of the ACM*, 52(6):28-30, June 2009.

[4] S. Hambrusch, C. Hoffmann, J. T. Korb, M. Haugan, and A. L. Hosking. A multidisciplinary approach towards computational thinking for science majors. *In SIGCSE '09: Proceedings of the 40th ACM technical symposium on Computer science education*, pages 183-187, New York, NY, USA, 2009. ACM.

[5] O. Hazzan. Reflections on teaching abstraction and other soft ideas. *SIGCSE Bulletin*, 40(2):40-43, 2008.

[6] R. K. Heinssen, C. R. Glass, and L. A. Knight. Assessing computer anxiety: Development and validation of the computer anxiety rating scale. *Computers in Human Behavior*, 3(1):49-59, 1987.

[7] J. Kramer. Is abstraction the key to computing? *Communications of the ACM*, 50(4):36-42, 2007.

[8] J. J. Lu and G. H. Fletcher. Thinking about computational thinking. In Proceedings of the 40th ACM technical symposium' on Computer science education, *SIGCSE '09*, pages 260-264, New York, NY, USA, 2009. ACM.

[9] Kuhn, Steven, "Prisoner's Dilemma", *The Stanford Encyclopedia of Philosophy (Spring 2009 Edition)*, Edward N. Zalta (ed.), URL=<http://plato.stanford.edu/archives/spr2009/entries/prisoner-dilemma/>.

[10] L. Perkovic, A. Settle, S. Hwang, and J. Jones. A framework for computational thinking across the curriculum. In Proceedings of the fifteenth annual conference on Innovation and technology in computer science education, *ITiCSE '10*, pages 123-127, New York, NY, USA, 2010. ACM.

[11] S.M. Pulimood and U. Wolz. Problem solving in community: a necessary shift in CS pedagogy. *In SIGCSE '08: Proceedings of the 39th SIGCSE technical symposium on Computer science education*, pages 210-214, New York, NY, USA, 2008. ACM.

[12] H. Qin. Teaching computational thinking through bioinformatics to biology students. *In SIGCSE '09: Proceedings of the 40th ACM technical symposium on Computer science education*, pages 188-191, New York, NY, USA, 2009. ACM.

[13] G. Wilson, C. Alvarado, J. Campbell, R. Landau, and R. Sedgwick. CS-1 for scientists. *SIGCSE Bulletin*, 40(1): 36-37, 2008.

[14] J. M. Wing. Computational thinking. *Communications of the ACM*, 49(3): 33-35, 2006.