What Everyone Needs to Know About Computation

John Barr
Ithaca College
barr@ithaca.edu

Steve Cooper
Purdue University
coopers@acm.org

Mike Goldweber
(XModerator)
Xavier University
mikeyg@cs.xu.edu

Henry Walker
Grinnell College
walker@math.grinnell.edu

Categories and Subject Descriptors
K.3.2 [Computers and Education]: Computer and Information Science Education—Computer Science Education

General Terms
Design, Experimentation

Keywords
Non-major courses, Curriculum, Computational Thinking

1. SUMMARY

For a variety of reasons there has been a renewed interest in non-major level computer science courses. One reason for this is that non-major courses are seen as potentially increasing both interest and enrollment in computer science courses. Another reason is the rising popularity of “computational thinking.” Furthermore, as computation becomes the cornerstone of research methodologies, if not professional practice, in the natural, physical and social sciences, including, as well, a growing influence in the humanities, the demand from external departments/programs for non-major level service courses has began to rise, albeit slowly.

This raises some interesting questions to the CSE community. Given that one offers a single non-major level course, what should be the topics covered as distinct from the technologies used to convey those topics? What learning outcomes are most desired? Finally, the competency, or at least serious introduction of which technological skills, if any, should be considered mandatory (e.g. programming).

In short, what should a non-major/minor student who takes one course in computer science come away with knowing and being able to do? Possibly orthogonal to this is the question: What should any liberally (college) educated person know about computation at this point in time?

This panel will outline four different perspectives on this important question in addition to engaging the audience in a discussion regarding the answers to the questions raised.

2. JOHN BARR

Our experience at Ithaca College is that students entering from high school have, over the last 10 years, become less computer literate. Ten years ago we could assume that students knew what a folder or directory was, how to create, rename, and move files, how to install programs, and even how to change basic operating system settings. Today we find that none of these skills are prevalent. Instead, students are experts at finding papers on the Web, installing apps on their cell phones, and are masters at all sorts of communication and social networking technologies from cell phones to texting to Facebook.

The traditional view of computer literacy sees this trend as an unfortunate development that needs to be reversed through computer literacy/non-major level courses. Students should be forced to learn computation, abstraction (such as folders and files), and the basics of maintaining their operating system. But is this true? What do students really need to know?

The discipline of computer science is dedicated to developing computational tools. One aspect of this goal is to make the tools more accessible and flexible. As the discipline matures, the tools it produces have also become more mature. Compare, for example, the iPhone against all previous smart phones. Certainly the iPhone is no less useful or powerful than previous phones. Yet users no longer need to know the arcania of folders, the simultaneous use of multiple buttons, and stupefying navigation techniques.

The lesson to take from this simple look at cell phone technology is not that users are becoming less adept. Rather the lesson is that users have mastered the technology that exists today, not the technology that existed ten years ago. The gritty details of computation and communication have been abstracted out and users no longer need to master these details to use these tools effectively. Just as some people would argue that computer scientists do not need to be masters of assembly language programming to be successful today, the average user of technology no longer needs to master the details of computation to be an effective user of computation-based technology.

3. MICHAEL GOLDWEBER

It is instructive to consider a proposed course, not just against other proposed courses, but against no course at all. After all, both automotive and telephone technology are as ubiquitous as computation, yet the vast majority can successfully utilize both technologies without any grounding, i.e. non-majors college course, in either physics or mechanical engineering. Taking this argument further one might conclude that other disciplines which require knowledge of computation would be satisfied by either requiring an intro-
dutory major-level CS course or covering what aspects of computation are needed in their own courses.

This position argues that a meaningful non-major level CS course should therefore not be concerned with skill acquisition. Instead it should focus on the discipline’s deep intellectual contributions; which may be described as algorithmic problem solving, harnessing abstraction to manage complexity, and the automation of an algorithmic solution. Of these, I believe that algorithmic problem solving is not only the most important, but also the one where the greatest success can be achieved.

Algorithmic problem solving has a number of standard paradigms; brute force, greedy, divide and conquer, dynamic programming, etc. In addition to these techniques are the important notions of problem reduction, when a new problem can be recast as an instance of a different, previously solved problem, and the recognition and handling of “hard” problems. Whether or not one ever intends to automate a given problem’s solution, being able to approach and eventually solve a given problem using one or more of these techniques is a highly valued skill.

Consider a non-major level course that involves no programming, no discussion of objects, encapsulation or polymorphism, but examines a wide variety of problems drawn from a plethora of problem domains and has students practicing how to develop solutions to these problems. Just as a liberally educated person should be versed in how mathematicians determine truth/solves problems; conjecture and proof, or how natural and social scientists do the same; reproducible experiments informed via the scientific method, they should also be versed in how computer scientists do the same; application of problem reduction and/or the use of standard algorithmic paradigms.

4. HENRY WALKER

Much discussion by panelists here and by others throughout the computing community supports the view that introductory computing courses should involve problem solving. In principle, I too support the idea that problem solving should be central to computer science courses — for non-majors, majors, and anyone else. In courses, I would hope that students would tackle problems and gain experience at finding solutions. However, I wonder how students (or the general citizenry) will state and explain their solutions, and the recognition and handling of “hard” problems. Whether or not one ever intends to automate a given problem’s solution, being able to approach and eventually solve a given problem using one or more of these techniques is a highly valued skill.

Consider a non-major level course that involves no programming, no discussion of objects, encapsulation or polymorphism, but examines a wide variety of problems drawn from a plethora of problem domains and has students practicing how to develop solutions to these problems. Just as a liberally educated person should be versed in how mathematicians determine truth/solves problems; conjecture and proof, or how natural and social scientists do the same; reproducible experiments informed via the scientific method, they should also be versed in how computer scientists do the same; application of problem reduction and/or the use of standard algorithmic paradigms.

5. STEVE COOPER

J. Wing has indicated that she considers computational thinking to be the process of abstraction. Though Wing goes on to focus on the “goodness” of abstractions from an engineering point of view (examining such measures as efficiency, correctness, and software engineering considerations such as simplicity, elegance, usability, modifiability, maintainability, cost, etc.), the bottom line is that any approach towards presenting computational thinking for the masses needs to focus on the process of abstraction. Of course, teaching the process of abstraction is a daunting task, attempting to tackle one of Brooks’ essential difficulties (referred to as software invisibility) inherent in software engineering.

If a non-majors course needs to help students understand the process of abstraction, it seems to me to be necessary to provide the students meaningful contexts from which they are comfortable in exploring abstraction. Certainly my own work with Alice (www.alice.org, www.aliceprogramming.net) provides the context of animation and storytelling from which to explore abstraction. Scratch (scratch.mit.edu) is an alternative. Guzdial’s work with media computation uses the manipulation of sound, images, and video as an alternative context from which to explore abstraction. Roberts’ work with the ACM Java Taskforce provides instructors with a powerful library of graphics and other routines from which to construct their own contexts. Robots are another context, as is the programmable clothing of Lilypad Arduino. And there are many others. It is also not necessary to focus on programming in providing the context. It is certainly possible to explore abstraction through the contexts of networking and/or communication. It is also possible to explore computational thinking and the process of abstraction through domain specific examples/contextes from the geo-sciences, economics, politics, biology, physics, etc. Instructors can demonstrate how computational thinking can be used to provide abstractions for the modeling of behavior.

Guzdial’s media computation is used as an example context by which to explain what I mean by the exploration of abstraction. A picture is an abstraction for a collection of pixels. A pixel is an abstraction for a 3-tuple consisting of x and y coordinates and a color. A color is an abstraction for a 3-tuple, say of red, green, and blue. And the abstractions can continue, by considering how a collection of images can be considered as a video, etc. The bottom line is that a concrete example, meaningful to the student, provides a potential context from which to help students explore the process of abstraction.