CRA-E White Paper

Creating Environments
for
Computational Researcher Education
August 9, 2010

Committee Members

Andy van Dam, Brown (chair)
Jim Foley, Georgia Tech
John Guttag, MIT
Pat Hanrahan, Stanford
Chris Johnson, University of Utah
Randy Katz, UC Berkeley
Henry Kelly, DOE (formerly FAS)
Peter Lee, DARPA and CMU
David Shaw, D.E. Shaw Research

Acknowledgements

Thanks to the CRA-E committee members, who provided lively, provocative, and generative ideas during the four meetings.

Thanks to Jeannette Wing, Lynn Andrea Stein, and Ed Fox for detailed and relevant reviews that helped refine and shape the raw material of the report.

Thanks to CRA's Andrew Bernat for support, patience, and timely suggestions.

Thanks to Rosemary Michelle Simpson, who pulled it all together and uncovered the patterns lurking in the mass of material.

COMPUTING RESEARCH ASSOCIATION
1828 L Street, NW Suite 800,
Washington, DC 20036-4632
Email: info@cra.org
Tel: 202-234-2111
Fax: 202-667-1066
URL: http://www.cra.org

Table of Contents

Executive Summary

The CRA-E committee was formed in the spring of 2008 and had its first meeting at the CRA Snowbird 2008 conference, with this as its mission statement:

> "Our charter is to explore the issues of undergraduate education in computing and computational thinking for those who will do research in disciplines from the sciences to the humanities.
>
> As technology and teaching methodologies continue to evolve, how should programs in computer science, computational science, and information science co-evolve? Can we communicate a core set of ideas, principles, and methodologies that is domain-independent?"

Over the following year and half the committee continued to meet and investigate the issues involved; this report records the recommendations that have resulted from that process.

Issues

The 20th century was characterized by exponentially rapid technological and societal changes which produced a heightened need across the population for educational flexibility and lifelong learning. Though this observation has often been repeated, like many clichés it remains true today and the need to adapt to the accelerating rate of change has become more apparent and urgent.

The explosive growth of computationally-oriented subjects in particular has deeply affected higher education – not only in computer science departments, but virtually everywhere else, as demonstrated in the rise of such diverse fields as computational-linguistics, -finance, and -history, digital arts, and even -philosophy. At the turn of the century many computer science departments simultaneously experienced three trends: *first*, a demand to include more and more topics and courses in the curriculum, *second*, decreasing enrollments (now reversed at many schools) and migration of some of the top students to other fields, such as molecular biology, that seemed newer and were perceived as more attractive, and *third*, external and internal pressures for greater flexibility in the selection of course sequences in parallel with a questioning of 'what every student graduating with a computer science degree should know'.

At the same time, as a consequence of growing up in a digital culture, today's students have very different skill sets than did students of the 20th century. They are adept at tinkering with online facilities to obtain - and shape - what they want, skilled at rapidly switching contexts and tasks (a controversial skill sometimes referred to as multitasking), and are experienced in the kind of ad-hoc collaboration exemplified by social networking. While many of their mentors acknowledge the value of these new capabilities, they often decry the loss of other abilities, such as the ability to read deeply and critically/analytically, and to write coherently and analytically, and point out the difference between social networking and genuine goal-directed teamwork skills.

Mindful of such changes, CRA asked its education committee, CRA-E, to address an open-ended question of concern not just to college educators but also employers, funding agencies, and policy makers: how best to educate students for a future as computationally-oriented researchers in all fields. Note: this question is related to but deliberately more focused on the needs of future researchers than a general consideration of computational thinking at all levels or the redesign of the computer science curriculum.

Goal of the CRA-E Committee White Paper

The overall goal of this white paper is to provide guidance that will help institutions create an undergraduate environment that supports the acquisition and internalization of the computationally-oriented researcher mindset. To achieve this, we identified two sub-goals: *first*, to identify the issues facing faculty charged with educating computationally-oriented researchers in the first part of the 21st century, and *second*, to make recommendations that address those issues and that are both relevant and implementable within the current institutional context.

Mechanisms for Implementing the Goal

Our intention is that colleges and universities adapt the recommendations of this report as appropriate to the needs of their own student population and institutional structures. To this end, the CRA-E committee suggests three major mechanisms for meeting the goal of fostering the creation of a computational researcher mindset:

- Develop *flexible curricular structures* that can more easily reflect and adapt to change. These curricular structures range from a "lean core" and specialized tracks to fully integrated joint majors that reflect and encourage deep collaboration and synergy among disciplines.
- Provide a *"research-oriented" environment* in the undergraduate program. Among other things such an environment would include: apprenticeships/internal internships, collaborations with researchers, projects requiring research skills, and independent study.
- Support the *assimilation and putting into practice of enduring cognitive skills and core concepts* over four years and different contexts through the deepening process of building mastery.

Scope of the CRA-E White Paper

Within the rubric of the CRA-E overall goal -- provide institutional guidance for developing undergraduates with a computationally-oriented researcher mindset -- the white paper addresses overall directions rather than comprehensive details; it is not a curriculum design. We tried not to duplicate work being done by related efforts such as the ACM/IEEE Computer Science Curricula 2001 report/2008 update, or any effort having to do with K-12 education such as ACM's Model Curriculum for K-12 Computer Science and NSF's CS 10,000 Project, whose goal is the revision of the AP computer science course. Instead we viewed our work as complementary to these efforts, providing a specific focus that still underscores and reinforces the overall goal of computationally-oriented education in the 21st century.

While we have leveraged the work of existing curriculum restructuring efforts and experiments already under way at a wide variety of schools, we have chosen a small group of schools for a more comprehensive focus. The common denominator of these efforts combines (1) a rigorous re-evaluation of what computationally-oriented students need to know with (2) the design of flexible specialization mechanisms that facilitate modification in response to changing needs, and (3) an emphasis on grounding the material in physically-situated 'artifacts'. Furthermore, we advocate relaxing the notion that every student takes the same sequence of core courses. The core may, in fact, be embedded in the specialization so that the content will vary.

Given CRA's focus on research, we have constrained our attention to issues concerned with how to best educate future computationally-oriented researchers. Within this framework, computer science departments take on a new and critical role as identifiers and promulgators of the core set of cognitive skills and computational concepts that inform various computational-X programs, independent of their domains. Thus, we have focused on computer science departments in Recommendations 2 through 6, while extracting their essential components in the appendices to aid other departments wishing to develop domain-specific computationally-oriented courses.

Unfortunately, we were not able to undertake the very important subject of pedagogy and the appropriate use of Internet- and other technology-based facilities as a component of undergraduate educational design; we hope that future CRA-E committees will undertake that investigation. However, [CSTB 2010] "Report of a Workshop on The Scope and Nature of Computational Thinking" contains a number of pedagogical discussions and references. In addition, the index of this report includes an entry "pedagogy, related references" with pointers to pedagogy-related references in the bibliography. The two sets of references complement each other - the CSTB report references focus on current and past practices, while the CRA-E report references focus on pedagogical theory and background.

Strategy of the CRA-E White Paper

Over the course of its four meetings, from Snowbird, 7/2008 to New York City, 8/2009, the committee identified a common set of concerns and approaches that we have encoded into six recommendations. While the recommendations both affect and reflect each other, we have arranged them into three thematic groups:

- *Introduce* students to computational thinking by foundational courses that address student interests within the fundamental range of computational thinking concepts and skills (*Recommendation 1),*
- *Refactor* computer science curricula that provide a flexible and adaptable range of options for computationally-oriented directions in any domain (*Recommendations 2, 3, and 4), and*
- *Identify* cognitive, mastery, and research skills that should pervade the entire curriculum, from introductory courses through the advanced courses taken by seniors heading to graduate school (*Recommendations 2, 5, and 6).*

This paper attempts to lay a foundation for addressing this common set of concerns across educational institutions of all sizes and types that prepare students for graduate school and potential research careers. In our descriptions of the recommendations we identify some of the enduring concepts and principles, as well as cognitive and researcher skills that interweave all the courses.

Recommendations

Note - Appendix A - Recommendations Summary contains a detailed list of all recommendations

Computationally-Oriented Foundations

1. **Introductory Courses - addressing a broad range of student interests**

   Address student interests while at the same time ensuring that these courses address a significant subset of the fundamental range of concepts and skills that comprise computational thinking *[See the local appendix at the end of this Introduction: Computational Thinking - Summary of Views].*

   Use these courses to instill a set of cognitive skills such as learning how to create, validate, and establish relationships among abstractions from data and information on hand, a key skill in effective modeling, simulation, and validation. This skill in working with abstractions, in turn, undergirds both the scientific method and computational thinking, and should be a part of every computationally-oriented course. The differences among such courses help to reinforce the underlying skills as students meet the same concepts in different contexts.

   Other examples of cognitive skills include: working with the tradeoffs involved with different representations; moving, where appropriate, from a declarative understanding of a problem to an imperative understanding of that problem; reducing computationally intractable problems to related tractable problems; and building, simulating, and validating computational models that shed light on important questions.

Refactoring Computer Science Curricula

2. **Core/Foundation for All Computer Science Graduates** - lean core with focus on enduring concepts, techniques, and skills

   A relatively lean core emphasizes foundational concepts and skills while allowing students more time to explore areas in depth, both by taking courses and by engaging in undergraduate research. Additionally, a lean core makes it easier for students with multidisciplinary interests to pursue a joint major *[See Recommendation 4 - Specialization: Integrated Joint Majors]* while still sharing a common experience with computer science majors.

3. **Specialization: Tracks, Threads, and Vectors** - flexible approaches to gaining understanding and skills

   Define sets of meaningful specializations to permit students to pursue their interests in a context

that guides their development while providing strong motivation. Ensure that these 'tracks' are specialized enough that a course sequence can lead to a student attaining some reasonable depth in the area but broad enough that someone in a company or graduate school will be able to fit it into their institutional context.

4. **Specialization: Integrated Joint Majors** - deep collaboration among disciplines

   Coherent, integrated multidisciplinary, inter-departmental joint majors provide a balanced approach that addresses the differences in intellectual culture, concepts, and strategies between different fields by establishing the common ground between them. Use these integrated joint majors to provide a creative synthesis beyond that which can be provided by a computer science department alone, one that blends the cultures and mindsets of multiple departments and synergistically establishes new techniques for problem solving.

Establishing Mastery across the Curricula

5. **Design Under Constraints and the Gaining of Mastery** - deepening the skill set

   Provide students the ability to attain mastery by gaining experience in learning new technologies and techniques, building and analyzing artifacts, and learning to understand design as an iterative process that involves evaluating tradeoffs, analyzing system performance, and testing at each step. Create design and development experiences that tap into the actual interests of the students within a structure that both rewards effort and requires debugging/dealing with the uncertainties and approximations of real-world non-determinacy.

6. **Attracting, Selecting, and Preparing Students for Research Careers** - developing computationally-oriented researchers

   Skillfully introduce research problems and their intellectual excitement in all courses, thus helping to entice potential research students by disabusing them of the notion that our field has become routinized. Successful courses that attract and excite students present new concepts within the context of the ongoing research of the R&D community.

   Combine explicit research skill training with an apprenticeship approach to acculturate future researchers to their communities of practice. Provide systematic guidance in the practices of computationally-oriented research from freshman year through graduation combined with the support provided by close relationships with graduate students, research groups, and professors.

Tools for Using this Report

We have provided several tools to help individual departments and institutions develop approaches that reflect their culture, goals and resources.

1) Each recommendation section contains background issues, examples, and solution approaches, with the resulting recommendations summarized at the end of the section.
2) The complete recommendation set is provided as a single document in Appendix A to give a sense of the set as a whole.
3) The references (Appendix B) contain both a bibliography and the complete set of all the URLs that are mentioned in the report.
4) The index supports both search and browsing modes through extensive See Also trails.

Introduction

The CRA-E committee was formed in the spring of 2008 and had its first meeting at the CRA Snowbird 2008 conference, with this as its mission statement:

> "Our charter is to explore the issues of undergraduate education in computing and computational thinking for those who will do research in disciplines from the sciences to the humanities.
>
> As technology and teaching methodologies continue to evolve, how should programs in computer science, computational science, and information science co-evolve? Can we communicate a core set of ideas, principles, and methodologies that is domain-independent?"

Over the following year and half the committee continued to meet and investigate the issues involved; this report records the recommendations that have resulted from that process.

Issues

The 20th century was characterized by exponentially rapid technological and societal changes which produced a heightened need across the population for educational flexibility and lifelong learning. Though this observation has often been repeated, like many clichés it remains true today and the need to adapt to the accelerating rate of change has become more apparent and urgent.

The explosive growth of computationally-oriented subjects in particular has deeply affected higher education – not only in computer science departments, but virtually everywhere else, as demonstrated in the rise of such diverse fields as computational-linguistics, -finance, and -history and even -philosophy, not to mention digital arts and entertainment technology. At the turn of the century many computer science departments simultaneously experienced three trends: *first*, a demand to include more and more topics and courses in the curriculum, *second*, decreasing enrollments (now reversed at many schools) and migration of some of the top students to other fields, such as molecular biology, that seemed newer and were perceived as more attractive, and *third*, external and internal pressures for greater flexibility in the selection of course sequences in parallel with a questioning of 'what every student graduating with a computer science degree should know'[1].

At the same time, as a consequence of growing up in a digital culture, today's students have very different skill sets than did students of the 20th century. They are adept at tinkering with online facilities to obtain - and shape - what they want, skilled at rapidly switching contexts and tasks (a controversial skill sometimes referred to as multitasking), and are experienced in the kind of ad-hoc collaboration exemplified by social networking[2]. While many of their mentors acknowledge the value of these new capabilities, they often decry the loss of other abilities, such as the ability to read deeply and critically/analytically[3], and to write coherently and analytically[4], and point out the significant difference between social networking and genuine goal-directed teamwork skills.

Mindful of such changes, CRA asked its education committee, CRA-E, to address an open-ended question of concern not just to college educators but also employers, funding agencies, and policy makers: how

---

[1] While some computer science departments viewed it in part as a threat because newer offerings in other departments might successfully compete for resources and students, others saw these trends as a great opportunity. Indeed, the question can be generalized as 'What should *every college graduate*, no matter what his or her major is, know about computational thinking and computational methods'. Certainly, an argument can be made for a form of universal literacy that goes well beyond a 'computing-lite' literacy course.

[2] [Palfrey & Gasser 2008]. Born Digital: Understanding the First Generation of Digital Natives.

[Boyd 2008] Taken Out of Context: American Teen Sociality in Networked Publics.

[3] [Adler & Van Doren 1972]. How to Read a Book.

[4] [Palfrey & Gasser 2008].

best to educate students for a future as computationally-oriented researchers in all fields.[5] Note: this question is related to but deliberately more focused on the needs of future researchers than a general consideration of computational thinking at all levels or the redesign of the computer science curriculum.

Goal of the CRA-E Committee

The overall goal of this white paper is to provide guidance that will help institutions create an undergraduate environment that supports the acquisition and internalization of the computationally-oriented researcher mindset. To achieve this, we identified two sub-goals: *first*, to identify the issues facing faculty charged with educating computationally-oriented researchers in the first part of the 21st century, and *second*, to make recommendations that address those issues and that are both relevant and implementable within the current institutional context.

Mechanisms for Implementing the Goal

Our intention is that colleges and universities adapt the recommendations of this report as appropriate to the needs of their own student population and institutional structures. To this end, the CRA-E committee suggests three major mechanisms for meeting the goal of fostering the creation of a computational researcher mindset:

- Develop *flexible curricular structures* that can more easily reflect and adapt to change. These curricular structures range from a "lean core"[6] and specialized tracks to fully integrated joint majors that reflect and encourage deep collaboration and synergy among disciplines.
- Provide a "research-oriented" *environment* in the undergraduate program. Among other things such an environment would include: apprenticeships/internal internships, collaborations with researchers, projects requiring research skills, and independent study.[7]
- Support the assimilation and putting into practice of enduring cognitive skills and core *concepts* over four years and different contexts through the deepening process of building mastery.

Scope of the CRA-E White Paper

Within the rubric of the CRA-E overall goal -- provide institutional guidance for developing undergraduates with a computationally-oriented researcher mindset -- the white paper addresses general directions rather than comprehensive details; it is not a curriculum design. We tried not to duplicate work being done by related efforts such as the ACM/IEEE Computer Science Curricula 2001 report/2008 update[8], or any effort having to do with K-12 education such as ACM's Model Curriculum for

---

[5] As [Lewis et al. 2010] point out, educators should also look closely at the attitudes students bring with them at the beginning of their undergraduate careers and how those attitudes mesh with the educational goals departments wish to inculcate.

[6] What do we mean by "lean core"? We use the term 'lean core' here to mean a set of foundational courses that embody the content we expect 'every student who graduates with some kind of CS degree to know'. Given that we advocate broadening the notion of CS to include not just traditional CS but also computational science and computational-X for almost any field X, we don't think of the core as a set, let alone a sequence of courses that are identical for all students. For example, computational biology students and students studying digital story telling (e.g., electronic game design and digital media) should be exposed to the same set of core concepts and techniques but may see them in quite different contexts/courses starting with the introductory courses. This range of possibilities could mean that every student still takes the same courses in a foundation-style sequence before specializing, or - increasingly more common - it could mean a suite of contextually-distinct courses from the very beginning.

[7] CRA-W (CRA Committee on the Status of Women in Computing Research) provides distributed mentoring and research programs (http://www.cra-w.org/projects). Another point to note here is that while the focus of 4-year colleges is teaching, not research, they provide a rich background for future researchers. Thus, it is highly appropriate for them to include computationally-oriented cognitive and research skills, as discussed in this paper, in their curricula.

[8] [ACM/IEEE 2008]ACM/IEEE Computer Science Curricula 2001 report/2008 update. Andrew McGettrick, Chair of the ACM Education Board and Education Council, reports in a personal communication that "The ACM Education

K-12 Computer Science[9] and NSF's CS 10,000 Project[10], whose goal is the revision of the AP computer science course. Instead we viewed our work as complementary to these efforts, providing a specific focus that still underscores and reinforces the overall goal of computationally-oriented education in the 21st century.

While we have leveraged the work of existing curriculum restructuring efforts and experiments already under way at a wide variety of schools, we have chosen a small group of schools for a more comprehensive focus.[11] The common denominator of these efforts combines (1) a rigorous re-evaluation of what computationally-oriented students need to know with (2) the design of flexible specialization mechanisms that facilitate modification in response to changing needs, and (3) an emphasis on grounding the material in physically-situated 'artifacts'[12]. Furthermore, we advocate relaxing the notion that every student takes the same sequence of core courses. The core may, in fact, be embedded in the specialization[13] so that the content will vary.

Given CRA's focus on research, we have constrained our attention to issues concerned with how to best educate future computationally-oriented researchers. Within this framework, computer science departments take on a new and critical role as identifiers and promulgators of the core set of cognitive skills and computational concepts that inform various "computational-X" programs, independent of their domains. Thus, we have focused on computer science departments in Recommendations 2 through 6, while extracting their essential components in the appendices at the end of this introduction and of Recommendation 2 to aid other departments wishing to develop domain-specific computationally-oriented courses.

Unfortunately, we were not able to undertake the very important subject of pedagogy and the appropriate use of Internet- and other technology - based facilities as a component of undergraduate educational design; we hope that future CRA-E committees will undertake that investigation. However, [CSTB 2010] "Report of a Workshop on The Scope and Nature of Computational Thinking" contains a number of pedagogical discussions and references. In addition, the index of this report includes an entry "pedagogy, related references" with pointers to pedagogy-related references in the bibliography. The two sets of references complement each other - the CSTB report references focus on current and past practice, while the CRA-E report references focus on pedagogical theory and background.

Strategy of the CRA-E White Paper

Over the course of its four meetings, from Snowbird, 7/2008 to New York City, 8/2009, the committee identified a common set of concerns and approaches that we have encoded into six recommendations.

Board and the IEEE Computer Society's Educational Activities Board are embarking on a project to produce the next version of their Computer Science curricular guidelines. Currently this is being referred to as CS 2012. Preliminary discussions on the matter took place at a meeting of the ACM Education Council (which the British Computer Society representatives attended) in Berkeley on 21st and 22nd June 2010."

[9] [ACM CSTA 2006] ACM's Model Curriculum for K-12 Computer Science

[10] NSF's CS 10,000 Project: Jan Cuny. "Computational Thinking will Require Transforming High School Computer Science: CS / 10,000 Project", http://un-gaid.ning.com/profiles/blogs/computational-thinking-will; Jan Cuny. "Transforming High School Computing: A Compelling Need, A National Effort", http://opas.ous.edu/Committees/Resources/Publications/NSF_AP_CS_10000ExecSumm_Ed.pdf.

[11] See Recommendations 2-5 for details.

[12] We want to argue for balance, that understanding concepts and developing cognitive skills must be joined with the hands-on experience of building artifacts. Not everything is virtualizable to software or mathematics; you do get confronted with physicality, the constraints of real devices. Furthermore, we are physical beings, and 'tinkering', to use John Seely Brown's phrase (http://www.johnseelybrown.com/- Tinkering as a Mode of Knowledge Production video), helps our brains to better build connections between concepts and the real world. While virtual labs have great practical and sometimes irreplaceable value, involving the human proprioceptic system in physical labs, with real devices provides invaluable experience with real physical constraints.

[13] The USC games major is one example - [Zyda 2009] "Computer Science in the Conceptual Age"

While the recommendations both affect and reflect each other, we have arranged them into three thematic groups:

- *Introduce* students to computational thinking by foundational courses that address student interests within the fundamental range of computational thinking concepts and skills (*Recommendation 1),* [14]
- *Refactor* computer science curricula that provide a flexible and adaptable range of options for computationally-oriented directions in any domain (*Recommendations 2*[15]*, 3, and 4), and*
- *Identify* cognitive, mastery, and research skills that should pervade the entire curriculum, from introductory courses through the advanced courses taken by seniors heading to graduate school (*Recommendations 2, 5, and 6).*

This paper attempts to lay a foundation for addressing this common set of concerns across educational institutions of all sizes and types that prepare students for graduate school and potential research careers. In our descriptions of the recommendations we identify some of the enduring concepts and principles, as well as cognitive and researcher skills that interweave all the courses.

Tools for Using this Report

We have provided several tools to help individual departments and institutions develop approaches that reflect their culture, goals and resources.

1) Each recommendation section contains background issues, examples, and solution approaches, with the resulting recommendatons summarized at the end of the section.
2) The complete recommendation set is provided as a single document in Appendix A to give a sense of the set as a whole.
3) The references (Appendix B) contain both a bibliography and the complete set of all the URLs that are mentioned in the report.
4) The index supports both search and browsing modes through extensive See Also trails.

Recommendations

Computationally-Oriented Foundations

1. **Introductory Courses** - addressing a broad range of student interests

   Address student interests while at the same time ensuring that these courses address a significant subset of the fundamental range of concepts and skills that comprise computational thinking *[See the local appendix at the end of this Introduction: Computational Thinking: Summary of Views].*

   Use these courses to instill a set of cognitive skills such as learning how to create, validate, and establish relationships among abstractions from data and information on hand, a key skill in effective modeling, simulation, and validation. This skill in working with abstractions, in turn, undergirds both the scientific method and computational thinking, and should be a part of every computationally-oriented course. The differences among such courses help to reinforce the underlying skills as students meet the same concepts in different contexts.

   Other examples of cognitive skills include: working with the tradeoffs involved with different representations; moving, where appropriate, from a declarative understanding of a problem to an imperative understanding of that problem; reducing computationally intractable problems to related tractable problems; and building, simulating, and validating computational models[16] that

[14] See "Computational Thinking - Summary of Views" at the end of this Introduction.

[15] See "Approach to an Integrated Map of Lean Core Cognitive Skills, Concepts, and Techniques" at the end of Recommendation 2: Core/Foundation for All Computer Science Graduates

[16] When we use the terms "model" and "modeling" in this paper we mean symbolic computational models, not numeric models, which are sets of differential equations.

shed light on important questions.

Refactoring Computer Science Curricula

2. **Core/Foundation for All Computer Science Graduates** - lean core with focus on enduring concepts, techniques, and skills

   A relatively lean core emphasizes foundational concepts and skills while allowing students more time to explore areas in depth, both by taking courses and by engaging in undergraduate research[17]. Additionally, a lean core makes it easier for students with multidisciplinary interests to pursue a joint major *[See Recommendation 4 - Specialization: Integrated Joint Majors]* while still sharing a common experience with computer science majors.

3. **Specialization: Tracks, Threads, and Vectors** - flexible approaches to gaining understanding and skills

   Define sets of meaningful specializations to permit students to pursue their interests in a context that guides their development while providing strong motivation. Ensure that these 'tracks' are specialized enough that a course sequence can lead to a student attaining some reasonable depth in the area but broad enough that someone in a company or graduate school will be able to fit it into their institutional context.

4. **Specialization: Integrated Joint Majors** - deep collaboration among disciplines

   Coherent, integrated multidisciplinary, inter-departmental joint majors provide a balanced approach that addresses the differences in culture, concepts, and strategies between different fields by establishing the common ground between them. Use these integrated joint majors to provide a creative synthesis beyond that which can be provided by a computer science department alone, one that blends the cultures and mindsets of multiple departments and synergistically establishes new techniques for problem solving.

Establishing Mastery across the Curricula

5. **Design Under Constraints and the Gaining of Mastery** - deepening the skill set

   Provide students the ability to attain mastery by gaining experience in learning new technologies and techniques, building and analyzing artifacts, and learning to understand design as an iterative process that involves evaluating tradeoffs, analyzing system performance, and testing at each step. Create design and development experiences that tap into the actual interests of the students within a structure that both rewards effort and requires debugging/dealing with the uncertainties and approximations of real-world non-determinacy.

6. **Attracting, Selecting, and Preparing Students for Research Careers** - developing computationally-oriented researchers

   Skillfully introduce research problems and their intellectual excitement in all courses, thus helping to entice potential research students by disabusing them of the notion that our field has become routinized. Successful courses that attract and excite students present new concepts within the context of the ongoing research of the R&D community.

   Combine explicit research skill training with an apprenticeship approach to acculturate future researchers to their communities of practice. Provide systematic guidance in the practices of computationally-oriented research from freshman year[18] through graduation combined with the support provided by close relationships with graduate students, research groups, and professors.

Structure of each recommendation section

---

[17] See an early precedent for this approach in [Gibbs & Tucker 1986]. "A Model Curriculum for a Liberal Arts Degree in Computer Science".

[18] There is a chicken and egg situation here - students may not know that they are potentially interested in a research career until they've had a chance to do some research. Therefore we advocate institutional support for programs that make it easy and natural for a student to experiment with doing research rather than a blanket expectation that every undergraduate should have a research experience.

Each of the recommendation sections has a common structure:

- Problem/issues
- Goals
- Solution mechanisms and strategies
- Recommendations

Computational Thinking - Summary of Views

Introduction

Computational thinking is a rich and evolving set of cognitive skills, concepts, and techniques that resists exact definition. It includes algorithmic thinking, but is more than that, parallel thinking, but is more than that, problem solving, but is more than that, data, state, behavior, interaction, and design, but is more than that. Indeed, the very difficulty of capturing its essential nature provides a key argument for the need to address it, however imperfectly, incompletely, and provisionally.

In this overview we give brief quotes from Jeannette Wing, Charles Isbell and Lynn Andrea Stein, Peter Denning, the NRC Workshop on The Scope and Nature of Computational Thinking report, and the CMU Center for Computational Thinking.

Jeannette Wing

"The essence of computational thinking is *abstraction*. In computing, we abstract notions beyond the physical dimensions of time and space. Our abstractions are extremely general because they are symbolic, where numeric abstractions are just a special case.

In two ways, our abstractions tend to be richer and more complex than those in the mathematical and physical sciences. First, our abstractions do not necessarily enjoy the clean, elegant or easily definable algebraic properties of mathematical abstractions, such as real numbers or sets, of the physical world. For example, a stack of elements is a common abstract data type used in computing. We would not think 'to add' two stacks as we would two integers. An algorithm is an abstraction of a step-by-step procedure for taking input and producing some desired output. What does it mean 'to interleave' two algorithms, perhaps for efficient parallel processing? A programming language is an abstraction of a set of strings each of which when interpreted effects some computation. What does it mean 'to combine' two programming languages? These kinds of combinators are themselves abstractions that take careful thought, perhaps an entire research agenda, to define. Second, because our abstractions are ultimately implemented to work within the constraints of the physical world, we have to worry about edge cases and failure cases. What happens when the disk is full or the server is not responding? What happens when a program encounters at run-time an error that should have been caught at compile time? How do we get a robot to move down a hallway without bumping into people?

...

And so the nuts and bolts in computational thinking are defining abstractions, working with multiple layers of abstraction and understanding the relationships among the different layers. Abstractions are the 'mental' tools of computing.

The power of our 'mental' tools is amplified by the power of our 'metal' tools. Computing is the automation of our abstractions. We operate by mechanizing our abstractions, abstraction layers and their relationships. Mechanization is possible due to our precise and exacting notations and models. Automation implies the need for some kind of computer to interpret the abstractions. The most obvious kind of computer is a machine, i.e. a physical device with processing, storage and communication capabilities. Yes, a computer could be a machine, but more subtly it could be a human. Humans process information; humans compute. In other words, computational thinking does not require a machine. Moreover, when we consider the combination of a human and a machine as a computer, we can exploit the combined processing power of a human with that of a machine. For example, humans are still better than machines at parsing and interpreting images; on the other

hand, machines are much better at executing certain kinds of instructions far more quickly than humans and processing datasets far larger than a human can handle."[19]

Charles Isbell and Lynn Andrea Stein

" As a discipline, computing brings together models, languages, and machines to represent and generate processes. The heart of computing is not the particular artifacts around which our curricula often revolve. Instead, this key idea—that models, languages, and machines are equivalent—is the fundamental core of computing. Further, this idea admits a broad set of practices and specialties, including computer science, information science, human-centered computing, software engineering, and many others, as well as what we will call more generally contextualized computing.

From this position, we also argue that the curricula of existing courses should be revisited to inculcate the computationalist mindset—specifically, core competencies in modeling, scales and limits, simulation, abstraction, automation, and interpretation of data.

For core computationalists for whom the historical computing curriculum centers on understanding or using the machine, we propose that courses also include a focus on models and languages— the intellectual frameworks of computationalist thinking. For contextualized computationalists, curricula grounded in principles of computationalist thinking tailored to domain-specific needs has the potential to be transformative, not only by encouraging innovation within a domain but also by creating entirely new disciplines."[20]

Peter Denning

"...computing is not just algorithms and data structures; it is transformations of representations. The interactions are the real action of a field. Their complexities and uncertainties demand constant experimentation and validation in science and engineering. They make things messy and unpredictable. They are sources of innovation."[21]

"Today the term [computational thinking] has been expanded to include thinking with many levels of abstractions, use of mathematics to develop algorithms, and examining how well a solution scales across different sizes of problems. There is something even more fundamental than an algorithm: the representation... Representations convey information. A computation is an evolving representation and an algorithm is a representation of a method to control the evolution... computational thinking is not a principle; it is a practice. A practice is a way of doing things at which we can develop various levels of skill."[22]

Great Principles of Computing: [23]

* Computation
* Communication
* Coordination
* Recollection
* Automation
* Evaluation
* Design

NRC Workshop on The Scope and Nature of Computational Thinking[24]

[19] [Wing 2008] "Computational thinking and thinking about computing"

[20] [Isbell, Stein, et al. 2009] "(Re)Defining Computing Curricula by (Re)Defining Computing"

[21] [Denning 2009a] "Computing: The Fourth Great Domain of Science"

[22] [Denning 2009b] "Beyond Computational Thinking"

[23] [Denning 2007] Great Principles of Computing website - http://cs.gmu.edu/cne/pjd/GP/GP-site/welcome.html

[24] [NRC 2010] "Report of a Workshop on The Scope and Nature of Computational Thinking"

Introduction

"...computational thinking is a fundamental analytical skill that everyone, not just computer scientists, can use to help solve problems, design systems, and understand human behavior. As such, they believe that computational thinking is comparable to the mathematical, linguistic, and logical reasoning that is taught to all children. This view mirrors the growing recognition that computational thinking (and not just computation) has begun to influence and shape thinking in many disciplines—Earth sciences, biology, and statistics, for example. Moreover , computational thinking is likely to benefit not only other scientists but also everyone else—bankers, stockbrokers, lawyers, car mechanics, sales people, health care professionals, artists, and so on.

To explore these notions in greater depth, the Computer and Information Science and Engineering Directorate of the National Science Foundation asked the National Research Council to conduct two workshops to explore the nature of computational thinking and its cognitive and educational implications. This report summarizes the first workshop, which focused on the scope and nature of computational thinking and on articulating what "computational thinking for everyone" might mean...

Under NRC guidelines for conducting workshops and developing report summaries, workshop activities do not seek consensus and workshop summaries (such as the present volume) cannot be said to represent “an NRC view” on the subject at hand. This workshop report reveals the plethora of perspectives on computational thinking..."

The range of perspectives is reflected in the themes used to organize the major section of the report, "What is Computational Thinking":

Computational Thinking as a Range of Concepts, Applications, Tools, and Skill Sets

Computational Thinking as Language and the Importance of Programming

Computational Thinking as the Automation of Abstractions

Computational Thinking as a Cognitive Tool

Computational Thinking in Contexts Without Programming a Computer

CMU Center for Computational Thinking

"Computational thinking is a way of solving problems, designing systems, and understanding human behavior that draws on concepts fundamental to computer science. To flourish in today's world, computational thinking has to be a fundamental part of the way people think and understand the world. Computational thinking means creating and making use of different levels of **abstraction**, to understand and solve problems more effectively. Computational thinking means thinking **algorithmically** and with the ability to apply mathematical concepts such as **induction** to develop more efficient, fair, and secure solutions. Computational thinking means understanding the consequences of **scale**, not only for reasons of efficiency but also for economic and social reasons."[25]

[25] http://www.cs.cmu.edu/~CompThink/ - accessed, 3 May 2010.

Computationally-Oriented Foundations

Recommendation 1: Introductory Courses - *addressing a broad range of student interests*

Problem

Computational thinking and methods have become essential tools for those wishing to pursue research careers in engineering, the physical, life, and social sciences, mathematics, art and design, and the humanities. Consequently, institutions should encourage all undergraduates contemplating research careers in these areas to learn about computational thinking early in their education. Indeed, some schools with a CSTEM (Computing, Science, Technology, Engineering, and Mathematics) focus, e.g., Georgia Tech and Harvey Mudd[26], now mandate that all their students take an 'introduction to computational thinking' course.

To succeed, departments must address students with different mindsets[27] as well as different interests. For example, there are students who think they dislike problem solving and dread the notion of programming but love exploring, finding patterns, making things, and other such activities that are also important to computationally-oriented research. The introductory courses, therefore, must address the following issues:

- How do we address a diverse and talented undergraduate population in these introductory courses?
- What cognitive skills, concepts, and techniques should be included in these courses?
- How do we leverage the personal interests and abilities of students?
- How do we integrate student experiences with introductory courses into a meaningful research-oriented undergraduate experience?

Goals

Computational thinking comprises a rich and evolving set of cognitive skills[28], concepts[29], and techniques[30],that draw from multiple disciplines whose development spans thousands of years. Yet, one approach[31] treats them as a logical extension to those fundamental skills children learn[32] that are accessible to all ages and mindsets[33] and relevant to all disciplines. If successfully embedded in the full pre-K through undergraduate range, computational thinking could become a common language that

---

[26] The name of the course is not significant; Harvey Mudd, for example, calls its foundations course "Introduction to Computer Science".

[27] See [Gardner 1983] "Frames of Mind", [Nisbett 2003] "The Geography of Thought", and [Nisbett 2009] "Intelligence and How to Get It", for discussions of the nature and impacts of innate and cultural mindsets on thinking, learning, and behavior.

[28] A *cognitive skill* is a mental skill required by computational subjects, e.g., identifying patterns in an information space, which may or may not be thought of as data. Another example is representing that pattern by a symbol or set of symbols and relationships.

[29] A *concept* is a named abstraction that has a definition, e.g., recursion; interaction; concurrency.

[30] A *technique* is a goal-directed set of operations and strategies, e.g., modeling, simulation, machine learning.

[31] [ACM CSTA 2006] "A Model Curriculum for K-12 Computer Science"

[32] See [Bransford et al. 1999] "How People Learn, [Donovan & Bransford 2005] How Students Learn"

[33] In [Bruner 1960] "The Process of Education", Jerome Bruner asserted that "We begin with the hypothesis that any subject can be taught effectively in some intellectually honest form to any child at any stage of development.", a statement that underscores the intention of the "pre-K to Grey" initiative reflected in the CSTA report cited above.

helps bridge the divide between the sciences and the humanities described by C.P. Snow in his 1959 lecture "The Two Cultures and the Scientific Revolution".[34]

Some of the ways in which these foundations can be introduced include the following:

- Offer a set of appropriate introductory courses that have no pre-requisite beyond high school mathematics. These courses should be designed to be attractive to students potentially interested in particular areas of study that are not identified as computer science per se, such as game design[35], computational science[36], digital art[37], or computational philosophy[38].
- Develop a core set of concepts and techniques for these courses (discussed under Solutions below) that enables students attracted to computationally-oriented approaches through one of the introductory courses to decide subsequently to deepen their understanding through additional courses in computer science, continuing on to pursue computationally-oriented research in their area.
- Ensure that some of the material learned in the introductory course is used in a variety of undergraduate courses throughout the university. Thus, encourage all departments to design courses that incorporate computational techniques in meaningful ways.
- Provide CS-oriented introductory courses for those students whose main interests are in computer science, whether theory or practice.[39]

Solutions - mechanisms for implementing the goals.

Things to know

At the end of one semester students should have learned:

Converting Patterns of Data to Knowledge

- Methods for exploring an interesting domain to understand what constitutes 'data' in that domain, and then extract and represent that data.
- Methods for deriving and validating information from data, including simple statistical and visualization tools.
- Methods for analyzing the data to determine what the fundamental issues are for modeling, simulation, and validation. Case studies with primary documents, similar to HBS (Harvard Business School) case studies could be an interesting approach.

Representing Relationships as Models and Programs

- A systematic approach to designing, writing, and debugging several hundred line programs. This should include an understanding of the reasons why programming is a way to manipulate patterns as well as a tool for problem solving and modeling, and how it compares with - and augments - other strategies such as the scientific method, mathematics, and classic humanities analytic strategies.
- The process of moving from an ambiguous problem statement to a computational formulation of a method for decomposing and solving the problem as a set or hierarchy of subproblems that solve/implement them.

---

[34] [Snow 1959] "The Two Cultures and the Scientific Revolution". Ironically, as some humanists have been decrying the loss of a common culture engendered to a large extent by the digital revolution, especially the Web, a new common culture has been arising from the computational culture embraced by 'digital natives'.

[35] USC Survey of Digital Games & Their Technologies http://www.cs.usc.edu/admissions/undergrad/bsgames.htm)

[36] University of Utah: Introduction to scientific computing http://www.eng.utah.edu/~cs3200/

[37] Stony Brook University: ARS 225 Introductory Digital Art http://www.art.sunysb.edu/undergrad.html

[38] Augsburg College: Major in Computational Philosophy http://www.augsburg.edu/cs/degree_requirements.html

[39] Examples include Brown's CS017/018, which is theory-oriented, and CS015/016, which is practice-oriented.

- The meaning and use of algorithms, including the importance of and strategies for scaling.

Exploring and Validating Hypotheses and Models

- Methods for using simulations to shed light on problems that are ambiguous or that don't have an obvious (closed form) solution.
- Validation strategies that analyze the results of a simulation against the initial hypotheses and data

Approaches to course design

Computer science and engineering

Many students are drawn to computer science for its theoretical, mathematical, and engineering aspects. These students are well-served by introductory courses that directly address these interests; the table below gives several examples, such as Brown University's CS015/016, CS017/018, and CS053, and CMU's CS15-105.

Contextualization

While domain-based approaches to teaching introductory computationally-oriented courses are not new[40], the motivation for doing so has grown much stronger with the drive to provide 'computational thinking for everyone'. In 2003, Georgia Tech "...adopted an approach that we call contextualized computing education. We chose to teach computing in terms of practical domains (a "context") that students recognize as important. The context permeates the course, from examples in lecture, to homework assignments, and even to the textbooks specially written for the courses. We decided to teach multiple courses, to match majors to relevant contexts."[41]

Much of the learning in a computational methodology course comes through using a programming language[42] to describe, investigate, and work with topics of interest. For example, humanities and social sciences students could learn how to represent a pattern they want to explore, such as analyzing voting bloc behavior or patterns of language use in speeches or documents, how to encode a strategy for doing that exploration, and how to validate the results against their expectations[43].

---

[40] At the CRA-E meeting in Snowbird in 2008 Peter Lee described his work with the Pennsylvania Governor's School for the Sciences (PGSS), a summer program for bright rising high school seniors. [Unfortunately PGSS was cancelled for lack of funds in 2009] As he described it, in order to get kids involved with computer science areas such as compilers or genetic algorithms, you have to "trick kids into doing this", e.g., engage their real interests in projects that they are motivated to do but which require them to work with the concepts you want to convey. As Peter put it: "So, putting those things together in service of, for example, analyzing or generating music in a certain style, ends up attracting a dozen of the 90, whereas only two had signed up for computer science. The majority of them are girls. This has been consistent over the last handful of years."

Another early example from the 1990s was a CUNY course that used the Internet as it's basis: [Gurwitz 1998] "The Internet as a Motivating Theme in a Math/Computer Core Course for Nonmajors".

[41] [Guzdial 2009] "Teaching Computing to Everyone". The term "contextualized computing" was first introduced by Mark Guzdial in [Rich et al. 2005] "A CS1 Course Designed to Address Interests of Women" and [Yarosh & Guzdial 2008] "Narrating Data Structures: The Role of Context in CS2".

[42] Experience suggests that it may be tricky to teach such a course when there is a significant disparity in programming experience among the students. Students with little or no programming experience can be intimidated by observing the more experienced students. This inhibits them from actively participating in class and, worse yet, can lead them to confuse a lack of experience for a lack of aptitude. Some students with programming experience could place out of the requirement to take an introductory course. However, some students manage to acquire some programming experience without learning much about computational thinking, problem solving, or how to produce a well-structured design. These students might benefit from alternate approaches including a faster-paced version of the class, a project-oriented approach that requires analysis of the results, or concept-oriented placement exams that require understanding, not just hacking.

[43] See Brown's CS931 - Introduction to Computation in the Sciences and Humanities (http://www.cs.brown.edu/courses/csci0931/)

One version of the course could provide projects from a variety of domains[44]. This would help the students appreciate the universality of the basic ideas covered in the course. However, it might also be productive to provide various versions of the course that tailor the programming projects to fit the interests of different groups of students; this might be a good way to pique student interest and help them appreciate the immediate relevance and utility of taking a computational approach to working with data. Such an approach provides design suites relevant to a particular engineering, science, humanities, or social science discipline. What all such courses should have in common is the notion of building a computation that provides a model of, and insight into, an interesting problem or area. For example, students interested in biology[45] might be asked to construct a simulation that sheds light on the interaction between antibiotics, microphages, and bacteria. Students interested in economics might be asked to formulate a portfolio management strategy as an optimization problem. Other domains include computer science itself as theory and as experimental science.[46]

Recommendations 2 through 5 provide many domain-specific examples that could provide materials for an introductory foundations course.[47]

Example Introductory Courses Chart

The chart below includes courses from a wide variety of institutions, ranging from small private liberal arts colleges to large public and private research universities. Six Brown University introductory computationally-oriented courses have been included to give a sense of the range that a suite of introductory courses might have.

Course	Course Domain
Brown - CS020 - Concepts and Challenges of Computer Science	Computer science history and trends - literacy course for non-computer science majors that introduces programming and analytic skills through HTML, PHP and Python assignments. No prerequisites. http://www.cs.brown.edu/courses/csci0020/
Brown - CS015-CS016 - Introduction to Object-Oriented Programming and Computer Science and Introduction to Algorithms and Data Structures	Traditional project-oriented, OO programming in Java with graphics (CS015) and Java-based introduction to algorithms and data structures (CS016) http://www.cs.brown.edu/courses/csci0150/ http://www.cs.brown.edu/courses/csci0160/
Brown - CS017-CS018 - CS: An Integrated Introduction	Multi-lingual two-semester introduction to the functional and imperative programming paradigms that includes Scheme, ML, and Java. http://www.cs.brown.edu/courses/csci0170/ http://www.cs.brown.edu/courses/csci0180/
Brown - CS019 - Programming with Data Structures and Algorithms	One-semester multi-lingual introduction to computer science for students with a strong prior computer science background. http://cs.brown.edu/courses/cs019/2009/
Brown - CS040 - Intro for Engineers and Scientists	Computational problem-solving techniques for scientists and engineers using MATLAB and C. Currently it may not be used as part of a CS major. http://www.cs.brown.edu/courses/csci0040/

[44] See Harvey Mudd's CS for Scientists (http://www.hmc.edu/newsandevents/Grants%20Fall09.html)

[45] An early - [NRC 1987] - report by the National Research Council Committee on Computer-Assisted Modeling. "Computer-Assisted Modeling: Contributions of Computational Approaches to Elucidating Macromolecular Structure and Function" describes the process and advantages of computational modeling for macromolecular biology.

[46] Note that contextualization applies to cognitive skills as well: Susan Engel in her insightful paper about good college teaching strategies and techniques [Engel 2009] describes "the relationship between the tools of a discipline and the problems that discipline could solve".

[47] For example, see CMU's Bachelor of Computer Science and the Arts program and Stanford's Symbolic Systems program, which are referenced in the Recommendation 3 examples charts, as sources of domain-specific examples that could be mined for domain-specific introductory courses

Recommendation 1: Introductory Courses

Brown - CS053 - Linear Algebra	"A computational approach to linear algebra that provides students interested in computer science an introduction to vectors and matrices and their use in modeling and data analysis. The course balances programming, algorithms, and proof techniques." This course has no prerequisites and satisfies requirements for linear algebra in the BSCS degree. http://www.cs.brown.edu/courses/csci0053/
Brown - CS931 (Intro to Computation in the Social Sciences and Humanities)	Humanities and social science problem design for non-CS majors. Topics covered include data gathering, data analysis, web-based interfaces, security, and scripting. http://www.cs.brown.edu/courses/csci0931/
CMU - CS15-105	Principles of Computation - "15-105 is an introduction to the principles that form the foundation of computer science for students with no prior background in computing. This course is suitable for students with a non-technical background who wish to study the key principles of computer science rather than just computer programming" http://www.cs.cmu.edu/~tcortina/15-105sp09/courseinfo.html
Georgia Tech - CS1315/CS1316 - Introduction to Media Computation and Representing Structure and behavior	Contextualized approach to introducing computing using a theme of manipulating media http://coweb.cc.gatech.edu/cs1315
Harvard - CS50	Computer Science Introduction - "Introduction to the intellectual enterprises of computer science and the art of programming. This course teaches students how to think algorithmically and solve problems efficiently. Topics include abstraction, encapsulation, data structures, databases, memory management, software development, virtualization, and websites. Languages include C, PHP, and JavaScript plus SQL, CSS, and XHTML. Problem sets inspired by real-world domains of biology, cryptography, finance, forensics, and gaming. Designed for concentrators and non-concentrators alike, with or without prior programming experience." http://www.cs50.net/
Harvey Mudd - CS for Scientists	Computational approaches to science applications. http://www.hmc.edu/newsandevents/Grants%20Fall09.html
MIT - 6.00 - Introduction to Computer Science and Programming	Introduction to computer science and programming for students with little or no programming experience. Students learn how to program and how to use computational techniques to solve problems. Topics include algorithms, simulation techniques, and use of software libraries. Assignments are done using the Python programming language. http://web.mit.edu/6.00/www/info.shtml
MIT 6.01- Introduction to EECS 1	Mobile robots focus on computational engineering. Assume a previous knowledge of programming. http://mit.edu/6.01/mercurial/fall09/www/index.html
Princeton - Computational Universe	Computational Universe - Computers have brought the world to our fingertips. We will try to understand at a basic level the science -- old and new -- underlying this new Computational Universe. Our quest takes us on a broad sweep of scientific knowledge and related technologies: propositional logic of the ancient Greeks (microprocessors); quantum mechanics (silicon chips); network and system phenomena (internet and search engines); computational intractability (secure encryption); and efficient algorithms (genomic sequencing). Ultimately, this study makes us look anew at ourselves -- our genome; language; music; "knowledge"; and, above all, the mystery of our intelligence." http://www.cs.princeton.edu/courses/archive/spring08/cos116/
Purdue - SECANT	SECANT: Science Education in Computational Thinking "SECANT is a community building project funded through the NSF CPATH (Pathways to Revitalized Undergraduate Computing Education)

	program[1]). The goal of SECANT is to bring together scientists who recognize that computer science has become indispensable to scientific inquiry and is set to permeate science in a manner that is transformative, changing computing from a service discipline for the sciences into a fundamental paradigm for science in general. The effort complements Purdue's recently adopted College of Science curriculum, which includes a requirement that all science students take at least one course in computing and at least one course giving a multi-disciplinary experience. "http://secant.cs.purdue.edu/
Towson - Honors 223	Honors 223 - Honors Special Seminar Topic: Everyday Computational Thinking "This course provides an introduction to "computational thinking," defined as the methods, models and other mental tools related to the understanding, utilization and design of computational processes as executed by humans or computers. Computational thinking manifests itself in the everyday tasks of problem solving, the everyday interactions with systems, and the everyday task of information processing. This course is one of five new "Computational Thinking" courses to be offered through the Honors College at Towson University in the 2009-2010 academic year. The development of these courses was supported by National Science Foundation CPATH grant # 0829661." http://triton.towson.edu/users/dierbach/Pages/Courses/Honors%20ECT/HONR223.htm
Union College/Lafayette College	Campus Wide Computation Initiative: A New Model for Computing Education. "Union and Lafayette Colleges have received a five-year $1.15M National Science Foundation grant for our proposal *Campus Wide Computation Initiative: A New Model for Computing Education*. The motivation of the CPATH solicitation was to revitalize undergraduate computing education. Rather than focus specifically on computer science enrollments, the goal of our project is to broaden the pool of students who are prepared to integrate computation into their fields of study. We believe that computation can play a role in disciplines across our campuses, and have designed the grant activities for the broadest possible participation." http://cs.union.edu/~barrv/Grants/computational-science.html
University of Washington BENE**FIT**	BENE**FIT (Fluency with Information Technology)** course. "**Fluency with Information Technology (FIT)** is the knowledge to explore, interact with, and live in a society that has become more and more dependent not just on technology in general, but on information technology (IT) in particular. As technology advances, so too must its users adapt in order to harness the skills necessary to employ IT to their advantage. By becoming a FIT individual, you are better able to apply today's information technology effectively in your personal and professional life, and to adapt it to personally relevant goals. Gaining technical skills, or technical literacy, is only the first BENE**FIT** of your journey towards fluency. BENE**FIT** goes beyond literacy to provide the additional concepts and capabilities that will allow you to ride the wave of ever-changing technology." Based on [CSTB 1999] "Being Fluent with Information Technology" http://courses.washington.edu/benefit/FIT100/
University of Utah	Access Program and Engineering Scholars Program, University of Utah http://www.science.utah.edu/access.html http://www.coe.utah.edu/current-undergrad/esp.php These are not computational in orientation but do provide excellent examples of introduction and inspiration in science and engineering.

Recommendations

- **Introduce students to computational approaches** through foundation courses across the spectrum of student interests, instilling a set of cognitive skills such as those described earlier in the Introduction - "... learning how to create, validate, and establish relationships among abstractions from data and information on hand, a key skill in effective modeling, simulation, and validation.... working with the tradeoffs involved with different representations; moving, where appropriate, from a declarative understanding of a problem to an imperative understanding of that problem; reducing computationally intractable problems to related tractable problems; and building, simulating, and validating computational models[48] that shed light on important questions."
- **Emphasize the creation of appropriate and usable sets of representations and relationships** among different levels of abstractions; a deep understanding of how to represent information is one of the most difficult cognitive skills students need to learn. The practice of presenting accessible but important research papers as part of introductory courses not only introduces students to actual research work but also starts to build an understanding of how to compare different representations, analyze unstated assumptions, and build a common representation structure across a set of related projects.
- **Establish collaborative efforts** involving a computer science department, which could assume primary responsibility for the courses, and the departments whose prospective students are expected to take one of the courses. Additionally, the computer science department should help other departments in developing follow-on courses that take advantage of computational thinking taught in the first course.
- **Begin to shape a collaborative student culture** that will mature into effective professional teamwork skills, as described in Recommendations Two through Six.
- **Encourage students to begin building a digital portfolio**, including journal entries, possibly online [dealing appropriately with privacy issues], that carries through the core ideas and can be added to and be available for research ideas, building mastery, understanding one's own perspective, use in applying to graduate school. See Recommendation Six for a more complete description of this idea.

[48] When we use the terms "model" and "modeling" in this paper we mean symbolic computational models, not numeric models, which are sets of differential equations.

Refactoring Computer Science Curricula

As described in the Introduction, the explosion of computationally-oriented content into essentially all academic areas has generated a rethinking of what constitutes computer science and how to address the various forms of "computational-X". Furthermore, rapidly changing situations drive the need for a flexible environment that facilitates adaptation to constant change, including the changing interests and mindsets of students. Finally, to attract and prepare top students for research careers, institutions must provide relevant course offerings, and opportunities to work in areas and with researchers who are on the leading edge of current research.

As computer science departments have embraced a wider and wider range of subjects, they are finding it necessary to rethink the foundation set of knowledge and skills. The approach we advocate, and one which is based in part on what multiple leading computer science departments are doing is to refactor curricula into a lean core set of foundational concepts and skills that all students learn, followed by an expanding set of specialized tracks among which students can choose. These tracks allow students to develop a depth of understanding in one or more sub-fields of interest.

The core+tracks strategy, of course, is not new. Computer science as well as even more mature subject areas, ranging from sciences such as chemistry and physics to humanities such as history and English literature have long addressed the wide, expanding, and evolving scope of their domains by identifying foundational subjects that all undergraduate majors in the field must take followed by upper-level courses in a specialty.

What is new is the attempt to address the expanding breadth of the subjects being incorporated into the computational framework as well as the way the methodologies of computational thinking, mathematics, and science are impacting all academic domains. Thus, the goal of refactoring is to structure curricula to accommodate change in content, context, and students by supporting flexibility and the gradual acquisition of skills over the full range of undergraduate courses. In particular, for maxiumum flexibility we endorse a trend of trimming down the core and moving trimmed material to the specialization tracks.

Refactoring is not a trivial process and needs to be undertaken with a clear sense of the foundational abstractions, skills, and topics of computer science in general, as well as the properties and constraints of departments and their relationship to other groups and fields. Each department must identify the specific courses that constitute a core for it, articulate why these are, and then identify how the specialization sequences are to be structured, how they relate to the core, how they interact with each other, and how they satisfy the requirements that graduates could expect from potential graduate schools.

A synergistic approach to refactoring addresses major interdisciplinary areas through the establishment of integrated joint majors. Integrated joint majors work best with domains where there is a strong balance between two or more major subject areas. Thus, this section has three recommendation areas:

2. **Core/Foundation for All Computer Science Graduates** - lean core with focus on enduring cognitive skills, concepts, and techniques
3. **Specialization: Tracks, Threads, and Vectors** - flexible approaches to gaining understanding and skills
4. **Specialization: Integrated Joint Majors** - deep collaboration among disciplines

Recommendation 2: Core/Foundation for All Computer Science Graduates - *lean core with focus on enduring cognitive skills, concepts, and techniques*

Problem

The computational core plays an important role in defining a common intellectual foundation for the field of computer science, and is critical to providing students with the flexibility to enter new sub-fields as they develop or as the student's interests change. Defining this common intellectual foundation is an ongoing task, changing as the relationship of computer science within itself and to the rest of the world changes.

Thus, the problem that must be addressed is what cognitive skills, concepts, and techniques the core should consist of, how this should be decided, how it should be embodied in specific courses and programs, and - perhaps most important - what the curricular change mechanisms[49] are that can and should evolve as the inevitable changes occur in the years to come.

Goals

Current attempts to re-envision and reposition computer science have resulted in broad initiatives ranging from Denning's Great Principles[50] (an approach that attempts to identify the principles undergirding and interlinking the various subject areas of computer science) to the large ACM/IEEE curriculum task forces[51]. CRA-E, advocates a "lean core+specialization" approach that identifies and emphasizes the foundational core while allowing students more time to explore topics and domains in depth, both by taking courses and by engaging in undergraduate research[52]. A lean core makes it easier for students with multidisciplinary interests to pursue a joint major *[See Recommendation 4 - Specialization: Integrated Joint Majors]* while still sharing a common experience with computer science majors. In addition, the explicit identification of the lean core components *[See Content Area Details below in this recommendation]* enables a wide range of institutions to identify resources, establish a strong basic computer science foundation, and help their graduates pursue computationally-oriented research careers.

The observations motivating this proposal, and many of the ideas embodied in it, have been derived from examining recent curricular changes at a number of leading-edge institutions, both large and small, including CMU[53], Cornell[54], Georgia Tech[55], MIT[56], Harvey Mudd[57], Olin[58], and Stanford[59]. While these programs differ from each other in many respects, there is considerable commonality:

- Each of these departments has reduced the number of core requirements by eliminating the need to take courses, that were once thought central to computer science, e.g., compilers or theory of

[49] See Recommendations 3 and 4 for examples of curricular change mechanisms.

[50] [Denning 2007] "Great Principles of Computing website"

[51] [ACM 2010] "ACM Curriculum Reports"

[52] See [Gibbs & Tucker 1986] referred to in Recommendation 1 for an early approach, and the Olin College small footprint approach for a current example: [Downey & Stein 2006] "Designing a small-footprint curriculum in computer science"

[53] http://www.csd.cs.cmu.edu/education/bscs/currreq.html

[54] http://www.cs.cornell.edu/ugrad/CSMajorTransition08-09.htm

[55] http://www.cc.gatech.edu/future/undergraduates/bscs/threads

[56] http://www.eecs.mit.edu/ug/newcurriculum/SBCS_6-3.html

[57] http://www.hmc.edu/academicsclinicresearch1.html

[58] http://www.olin.edu/academics/olin_history/cdmb_report.html

[59] [Sahami et al. 2010] "Expanding the frontiers of computer science: designing a curriculum to reflect a diverse field"

computing. Students interested in learning about these areas will find courses dealing with them in a specialization *[See Recommendation 3 - Specialization: Tracks, Threads, and Vectors].*

- Some fields that were typically represented by one or two core courses have evolved into large and active research areas, for example computer graphics gave rise to computer games and digital media, deserving of their own tracks.
- Additionally, some areas that have not historically been thought of as central to computer science, such as probability and statistics, have entered a number of core curricula and play a prominent role in many of the tracks as the need for dealing with uncertainty, massive data sets, and simulation has become more prominent. Thus, we recommend some coverage of probability and statistics in the core.

Solutions - Mechanisms for Implementing the Goals

The components of the lean core comprise *cognitive skills*, *concepts, and techniques* embedded within specific *content areas*. We first give an overview of each component, with an expanded section on the content area details. Then, we provide representative lists of cognitive skills, one approach to lean core concepts, and lastly a small set of techniques. Our set builds upon our experience as well as being influenced in part by the curricular restructuring at CMU, Cornell, Georgia Tech, Harvey Mudd, MIT, and Stanford, and the thinking of Peter Denning, Jeannette Wing, Charles Isbell, and Lynn Andrea Stein and is meant to be representative and illustrative, not definitive.

To repeat and extend the definitions from Recommendation One:

Cognitive skills: A cognitive skill is a mental skill required to understand and practice computational subjects. Cognitive skills include, among others, traditional problem solving and mathematical skills, such as pattern recognition, facility with different levels of abstraction and representation, and inductive reasoning as well as the critical thinking, analysis, and synthesis skills gained from work in the humanities and social sciences.

Concepts: A concept is a named abstraction that has a definition, such as recursion and concurrency. Concepts are generators in the sense that they underlie and give meaning to computer/computational science. They lead to specific techniques that instantiate the concept. They include: algorithmic thinking and problem analysis, levels of abstractions, representation, approximation, and dealing with errors, constraints on computation and computational complexity, data structures and algorithms, transformation and patterns, communication and coordination, optimization, flow of control, and human-computer interaction.

Techniques: A technique is a goal-directed set of strategies and operations, such as modeling, simulation, and machine learning. Techniques are strategies and operations that manipulate data, apply across domains, and are grounded in the real physical world. They include such areas as: mathematical modeling, numeric simulation, programming languages, massive data set exploration[60], scientific method, and proof techniques.

Content areas/domains: In this description we describe our recommendations for the lean core content that will provide the foundation for learning the essential and enduring cognitive skills, concepts, and techniques needed by future computationally-oriented researchers.

Content Area Details

The core, as we envision it, has two parts. The first part, which we call "the shared core", is not specific to computer science, and describes a set of knowledge and skills that should be acquired by every

[60] See [Hey et al. 2009] "The Fourth Paradigm: Data-Intensive Scientific Discovery": "Increasingly, scientific breakthroughs will be powered by advanced computing capabilities that help researchers manipulate and explore massive datasets. The speed at which any given scientific discipline advances will depend on how well its researchers collaborate with one another, and with technologists, in areas of eScience such as databases, workflow management, visualization, and cloud computing technologies."

undergraduate contemplating a computationally-oriented research career. It will have much in common with what students in other CSTEM disciplines study as core.

The second part, which we call "the computer science core", discusses additional knowledge and skills that should be acquired by those contemplating a research career in any branch of computer science and gives the rationale behind those choices - how they provide a necessary and sufficient foundation for current, and as yet unforeseen, specialized tracks.

[See Recommendations 1, 5, and 6 for more details about the lean core component skills and how they could develop throughout the undergraduate curriculum.]

Computational Thinking/Methodology Introduction: In some implementations of these recommendations the introductory course described in Recommendation One will be part of the computer science core, while in others it may well be grounded in a domain that lies in the shared core and would build a bridge between fields. For example, a computationally-oriented linear algebra course[61] could serve as an introductory course for mathematically-inclined students.

A different approach to the introductory course might provide exposure to methods for using computation to understand specific domains ranging from the sciences to the humanities. While in some versions the student may do a substantial amount of programming, programming skill is not the goal; the primary intellectual focus will be on using computation to gain insights into specific domains.

Shared Core: Research is often multidisciplinary. Undergraduates are, therefore, well advised to acquire a broad multidisciplinary education. However, acquiring any particular set of knowledge or skills is less important than exposure to the different modes of thought associated with different branches of mathematics, science, and the humanities, in addition to an introduction to computational thinking.

Consequently, the particular topics recommended here should be viewed as illustrative rather than prescriptive.

- *Mathematics:* calculus and analytic geometry provide exposure both to thinking in the continuous domain and to geometric thinking. Linear algebra provides exposure to modeling problems as systems of equations and to the representational roles and relationships of vectors and matrices. Probability and statistics provide exposure to mathematical reasoning in the presence of uncertainty. To provide both computational and mathematical undergirding, the syllabus should include applications, algorithms, and proofs.
- *Sciences*: physical sciences, such as physics or chemistry, and life sciences, such as biology or neuroscience, provide experience with the various uses of the scientific method - experimental, theoretical, and naturalist. The sciences provide training in modeling, simulation, and validation, including the issues of what constitutes a viable hypothesis, how to choose what to include in a model, what the assumptions are, validation of results, and what the implications are.
- *Humanities*: provide training in critical analysis and synthesis in areas where the information may be ambiguous, subjective, and controversial. Some classical critical reading, writing, and thinking skills include building logically correct arguments, detecting logical fallacies, identifying implicit assumptions, and managing complex and contradictory arguments and information.

Computer Science Core: The four areas described below can be implemented in a variety of ways, depending on the particular faculty background and research interests. For example, the advanced programming area could be implemented in a design studio format or as a software engineering practicum offered in the context of an industrial partnership.

[61] See Brown University's CS053.

The goal is to give undergraduates a foundational set of computationally-oriented concepts and skills, along with a taste of what research in different areas of computer science might be like. They can then choose to learn more about specific areas by completing their degree in specialized tracks. The foundational concepts would include the following:

- *Mathematical and logical foundations for computer science:* Different kinds of computer science research build upon different mathematical foundations. For the most part, these are the same mathematical foundations used by other scientists and engineers. There are, however, a few areas that are not likely to be touched upon in courses of the kind discussed in under the Shared Core. This area provides a grounding of some of these topics, including formal logic[62] and proof techniques, induction, and sets/functions/relations. The topics covered should provide the mathematical tools necessary to study more advanced topics such as program analysis, algorithm and protocol design, cryptography, etc.
- *Algorithms:* All research in computer science requires an understanding of algorithmic thinking and techniques. Some of this will have been introduced in the introductory courses, but this area should go considerably deeper. Students should acquire a good understanding of orders of growth, basic optimization techniques, the relationship of data representations to performance on contemporary computing systems, and the uses of randomness.
- *Advanced programming:* While all students will have learned to write small programs in their introductory courses *[See Recommendation 1 - Introductory Courses]*, they will not have learned much about the engineering that goes into producing high quality software. This area helps them learn to design, implement, and test moderately-sized (several thousand lines) programs of the sort produced in academic research groups. The implementation should include a team project and provide the students with experience of incorporating/modifying existing building blocks, designing interactive user interfaces, exploiting concurrency and dealing with probabilistic issues of the sort dealt with in machine learning problems. Note that a design studio approach is well suited to this area.
- *The computation stack:* This area provides a vertically integrated look at how a modern computation system is built. It covers architectures for multi-core chips, computer organization, operating systems, and distributed systems. Many of these topics are related to the physical constraints that govern the efficiency of computing hardware. Unlike traditional versions of this material there should be considerable emphasis on parallelism, communication, and scalability. A topic of particular relevance is data-intensive, cloud computing that is done on huge clusters where software rather than hardware is used to achieve reliability.

Approach to an Integrated Map of Lean Core Cognitive Skills, Concepts, and Techniques

The interactive NSDL AAAS Science Literacy Maps[63] cited below provide an excellent model for a tool that could help guide both students and organizations in learning about and using core computational

[62] In [Vardi 2009] Moshe Vardi, current editor-in-chief of ACM Communications, states: "Logic has been called 'the calculus of computer science'. The argument is that logic plays a fundamental role in computer science, similar to that played by calculus in the physical sciences and traditional engineering disciplines. Indeed, logic plays an important role in areas of Computer Science as disparate as architecture (logic gates), software engineering (specification and verification), programming languages (semantics, logic programming), databases (relational algebra and SQL), artificial intelligence (automatic theorem proving), algorithms (complexity and expressiveness), and theory of computation (general notions of computability)."

[63] See the NSDL Strand Maps (http://strandmaps.nsdl.org/cms1-2/docs/index.jsp) - "NSDL Science Literacy Maps are a tool for teachers and students to find resources that relate to specific science and math concepts. The maps illustrate connections between concepts as well as how concepts build upon one another across grade levels. Clicking on a concept within the maps will show NSDL resources relevant to the concept, as well as information about related AAAS Project 2061 Benchmarks and National Science Education Standards."

"The Strand Map Service (SMS) (http://strandmaps.nsdl.org/cms1-2/docs/index.jsp) provides an interactive graphical interface that helps K-12 teachers and students understand the relationships between science concepts. The

cognitive skills, concepts, and techniques. We encourage research into methods for creating and exploring an integrated, interactive map of how the cognitive skills, concepts, and techniques interact with each other in a lean core. To initiate that endeavor we present here a set of representative lists as starting seeds.

Cognitive Skills

Abstractions - creating and validating

Algorithmic thinking - representing information, working with constraints and automating the process

Analysis - examining the components and structure of concepts, data, and research results

Approximations - estimating from data observations and representing in algorithmic form

Assumptions - identifying and validating

Automation - representing processes in terms of repeated operations such as iteration and recursion

Comparing and contrasting - identifying the way in which two or more things are similar and different. The basis for creating abstractions.

Critical reading and writing - close attention to the semantics of terms, unstated assumptions, and relationships with other work. Related work sections in research papers and reporting on papers in seminars provide training in this skill

Debugging - detecting pattern anomalies, using isolation strategies

Decomposing - complex entities into simpler ones

Designing - integrating user, performance, simplicity, and reliability concerns

Evaluating results in terms of assumptions and goals

Exploring - observing and identifying patterns for possible classification

Hypotheses - pattern recognition and assumption use in forming

Integrating disparate data and concepts

Interaction - identifying and representing different roles and their interrelationships; developing communication mechanisms among the different roles

Logical analysis of representation relationships

Parallel thinking - identifying sub-components that don't share dependencies

Patterns - recognition and classification

Planning - setting goals, developing strategies, and outlining tasks and schedules to accomplish the goal

Problem solving - working with time and space constraints, decomposing complex problems,

Rapid prototyping - integrating representation relationships, implementing, and evaluating the outcomes

Reasoning under uncertainty - reasoning and making decisions based on incomplete and/or uncertain data and models

Representing abstractions and their relationships through notations and language

Scaling - understanding time/space/and power constraints

Searching - focused exploration

Symbols and notations - representing and manipulating information and relationships

---

interactive maps are available through a public Web 2.0 JavaScript API and a REST API that lets developers embed the maps in Web sites and display educational resources and other content in the maps."

Synthesis - combining components of concepts, data, or research into a new construction

Tinkering - manipulating portions of existing entities

Tradeoffs - working with

Translating qualitative insights into computative representations

Concepts and Principles

Note: As many as possible of these core concepts/principles and their simplest expressions as techniques should be covered at some level in introductory courses - the spiral approach[64]. As stated above, this summary, which is not intended to be definitive but only as a starter kit - each institution/department will have its own collection. Our list builds upon our own experience combined with contributions from the curricular restructuring at CMU, Cornell, Georgia Tech, Harvey Mudd, MIT, and Stanford, and the thinking of Jeannette Wing, Lynn Andrea Stein, and Peter Denning.

Algorithmic thinking and problem analysis

- Problem decomposition
  - o divide and conquer
  - o levels of abstractions
- Reasoning
  - o correctness, logics, invariants, verification, debugging
- Time and space constraints

Abstractions (levels of)

- *What* to model
  - o salients, constraints, pitfalls in assumptions and in approximations
- *How* to model it
  - o what type
  - o multidisciplinary models
- *How* to implement the model
  - o solve analytically
  - o simulate
    - - kinds of simulation
  - o visualize the results

Representation, approximation, and dealing with errors

- Data
  - o types of data to be represented
  - o representation techniques and formats, and their limitations
- Behavior
  - See the next section - Techniques - and, in this section - Concepts and Principles:
    - - Processing techniques and their limitations
    - - Flow of control

Processing techniques and their limitations

- Linearization
- Kinds of simulation
- Granularity in spatio-temporal sampling

[64] See [Bruner 1960] "The Process of Education"

Computational models and constraints on computation

Models of computations...

- o automata and grammars
- o computation graphs
- o dataflow and Petri Nets
- o ATN

Scaling - constraints and tradeoffs in time, space, power, ...

Fault tolerance, reliability

Complexity, intractability, undecidability

Data structures and algorithms

(the usual and growing collections)

Graphs and networks

- o physical
- o virtual
- o social
- o hypertext

Transformation and Patterns

Transformation

- o mapping between representations
- o examples: rule-based systems...

Patterns

- o defining -> searching vs. discovering/recognizing
- o examples: machine learning...

Language models

Information, Knowledge, and Machine Learning

Information

- o data models
- o query languages
- o issues - data integrity, ...

Knowledge

- o representaton
- o logical reasoning and cognition
- o examples - natural language processing, ...

Machine learning

- o supervised and unsupervised learning
- o examples - data mining, robotics, ...

Communication and coordination

Abstraction levels and protocols

Centralized/distributed

Models such as

- o synchronous/asynchronous
- o broadcast/P2P
- o client-server

- o shared memory/message-passing
- o blackboard architecture
- o cloud

Error handling

- o concurrency control problems [Denning] and deadlock

Flow of control

Sequential

Conditional

Iteration

Recursion

Parallelism

- o co-routines
- o threads and processes
- o multi-processing
- o multi-core
- o distributed

Non-deterministic computation

The human element

Why the human element matters

What aspects to consider

- o perception
- o cognition
- o interaction
- o social dynamics
- o ...

Techniques

Abstraction mechanisms
Combinatorics
Distributed processing
Exploration of data-intensive subjects
Machine learning
Modeling
Numerical Methods
Programming
Proof techniques
Scientific method
Simulation
Symbol manipulation
System design

Recommendations

Lean core with focus on the minimum essential cognitive skills, concepts, and techniques.

- Having a relatively lean core emphasizes **foundational cognitive skills and concepts** while allowing students more time to explore areas in depth, both by taking courses and by engaging in undergraduate research.
- The deep issues of **mastery and skills** faced by the core have strong connections to the issues discussed in *Recommendation 5 - Design under Constraints and the Gaining of Mastery*.
- A lean core makes it easier for students with **multidisciplinary interests** to pursue a joint major *[See Recommendation 4 - Specialization: Integrated Joint Majors],* while still sharing a common experience with computer science majors.
- The explicit identification of the lean core components makes it easier for a wide range of institutions to **identify resources**, establish a strong basic computer science foundation, and help their graduates pursue computationally-oriented research careers.

Recommendation 3: Specialization - Tracks, Threads, and Vectors - *flexible approaches to gaining understanding and skills*

Problem

Commitment to a lean core of fundamental concept and content areas opens the question of how to guide students beyond traditional computer science tracks such as systems, languages, AI, and theory. The discipline of computing is now so broad that we can't expect students to be conversant with key ideas in each of the sub-disciplines available to them.

Furthermore, these sub-disciplines themselves are subject to change, evolving rapidly as new areas become important and older areas fade or change their focus. In addition, students are attracted to domain-focused content that appeals to their talents, mindsets, and interests. Finally, students benefit more when they are helped to attain real depth in one or more areas of specialization as opposed to simply gathering a scattershot collection of electives.

Goal

Define sets of meaningful specializations that represent important aspects of computing and that permit students to pursue their interests in a context that guides their development while providing significant motivation to persist. Such contextualized computing tracks should be specialized enough that a course sequence can lead to a student mastering the fundamentals of an area *[See Recommendation 5 - Design under Constraints and the Gaining of Mastery]*, yet broad enough that prospective employers and graduate schools will be able to fit it into their reference frame

During the 4-5 year process of achieving this mastery students develop cognitive skills that persist as they reencounter those skills in new domains and challenges. For example, they must develop accurate abstractions from data in all computationally-oriented courses, whether in a digital media track or a computational biology track.

Solutions - Mechanisms for Implementing the Goal

As Furst, Isbell, and Guzdial point out in their SIGCSE 2007 paper on the design of Georgia Tech's Threads program[65], specialized 'tracks' (Threads in Georgia Tech's case) are extremely context-sensitive to the particular school; domain as well as institution type and size and context are critical. Tracks can be rooted in introductory courses and use the mastery and design courses described in *Recommendation 5 - Design under Constraints and the Gaining of Mastery*. Related specialization approaches include vectors and minors. They represent a computer science-focused approach to specialization, while double majors and traditional joint majors include department-centered courses from other departments as well as computer science. The most consolidated approach, integrated joint majors which is described in *Recommendation 4 - Specialization: Integrated Joint Majors*, represent a blended-culture approach of computer science and another domain.

Tracks

Tracks provide a set of related courses intended to provide additional depth in one or more areas. At Stanford, gateway courses provide overviews so that students can sample a track before making a decision about whether or not to continue in that area. Stanford[66] also provides an "unspecialized track for students who take 4 or 5 gateway courses without finding a single area they like best".

[65] [Furst et al. 2007] "Threads™: How to restructure a computer science curriculum for a flat world".

[66] http://csmajor.stanford.edu/Tracks.shtml

Threads

Georgia Tech's approach to specialization is a highly structured and intertwined set of contexts that *[Furst et al 2007]* describe as follows: "Threads form a cohesive, coordinated set of contexts for understanding computing. The union of all threads covers the breadth 'computer science'. The union of any two threads is sufficient to cover a computer science degree."

Vectors

Cornell's specialization format is called vectors. It describes the reasoning behind it as follows: "The term 'vector' is meant to be evocative. Vectors have a direction (intellectual coherence) and a magnitude (coursework requirements), and need not be even close to orthogonal, but rather can have high inner product (overlap). We thus did not take a "top-down" approach of trying to divide computer science up into a relatively few distinct sub-fields, but rather, a "bottom-up" approach, where we can create new vectors on the fly as the field evolves."

Double Majors/Traditional Joint Majors

Double majors range from traditional combinations such as Harvey Mudd's [67]Computer Science and Mathematics to programs like Brown's [68]Computer Science and Economics, CMU's [69]BCSA (Bachelor of Computer Science and the Arts), and Georgia Tech's [70]Computational Media.

Minors

Minors fit into the classic major/minor undergraduate path and provide a way for non-computer science majors to incorporate computer science elements with minimal administrative load, as well as a way for computer science majors to incorporate their domain of interest, such as history or music[71]. Harvey Mudd has an interesting approach in that students who have a non-HMC (Harvey Mudd College) major at one of the other Claremont colleges (it must be a major that is not offered at HMC) can take a computer science minor at HMC.

Specialization Program Examples Chart

School	Program Type	Description
CMU	Double Major	**Bachelor of Computer Science and Arts (BCSA) Degree Program:** Sponsored by the School of Computer Science (SCS) and the College of Fine Arts (CFA) at Carnegie Mellon. "The BCSA curriculum has three main components: general core requirements, fine arts concentration requirements, and computer sciences concentration requirements. Each student's course of study is structured so they can complete this rigorous program in four years." http://www.cmu.edu/interdisciplinary/programs/bcsaprogram.html
Cornell	Vectors	Computer science majors are required to complete one vector from among the following current set: Renaissance (Basis), Network Science, Artificial Intelligence, Programming Languages, Computational Science and Engineering, Security and Trustworthy Systems, Data-Intensive Computing, Software Engineering / Code Warrior, Graphics, Systems, Human-Language Technologies, Theory http://www.cs.cornell.edu/ugrad/vectors.htm http://www.cs.cornell.edu/degreeprogs/ugrad/CSMajor/Vectors/index.htm http://www.cs.cornell.edu/ugrad/CSMajorTransition08-09.htm
Georgia Tech	Threads	There are eight threads: Modeling & Simulation, Devices, Theory, Information Internetworks, Intelligence, Media, People, and Platforms. CS majors are

[67] http://www.cs.hmc.edu/program/csmath-major

[68] http://www.cs.brown.edu/ugrad/concentrations/cs-econ_scb-reqs.html

[69] http://www.cmu.edu/interdisciplinary/programs/bcsaprogram.html

[70] http://lcc.gatech.edu/compumedia/

[71] CMU requires computer science majors to minor in something else.

		required to complete any two threads for their degree, which provides 28 possible combinations. Minors in Computer Science select courses from among the Threads, with a requirement that at least two of the courses be from the same thread. http://www.cc.gatech.edu/future/undergraduates/bscs/threads/ http://www.cc.gatech.edu/future/undergraduates/csminor
	Traditional Joint Major	"The Bachelor of Science in Computational Media (BSCM) was developed in recognition of computing's significant role in communication and expression, and is a joint offering between the College of Computing and the School of Literature Communication and Culture within the Ivan Allen College of Liberal Arts." http://www.cc.gatech.edu/future/undergraduates/bscm http://lcc.gatech.edu/compumedia/
Harvey Mudd	Traditional Joint Major	This small (~700 students) CSTEM-oriented liberal arts college handles specialization in several creative ways - utilizing collaboration among departments of the school, cooperation with area colleges and joint work with industry and research institutions. Students may elect a computer science and mathematics joint major, a standard computer science major, or a computer science minor with a major taken at another area college.
	Minor	"The HMC CS minor is designed for students who want to pursue other interests but who would also like to build a background in computer science. It is available only to students with an off-campus major." *http://www.cs.hmc.edu/program/cs-minor* *http://www.hmc.edu/academicsclinicresearch/majors/offcampusmajor.html*
MIT	Theme areas	"While breadth is important, it is also crucial for students to attain mastery in some area. This gives satisfaction and a sense of achievement, as well as the confidence and ability to go on to master new areas. In this curriculum, we will ask undergraduates to choose two specialization areas to study in depth, and to build a curriculum of foundational subjects that support study in those areas." *http://www.eecs.mit.edu/ug/newcurriculum/ugcur-newsletter06.html* *http://www.eecs.mit.edu/ug/newcurriculum/index.html*
Olin	Specialization and Realization	"... the Olin curriculum consists of three phases: *foundation* , which emphasizes mastering and applying technical fundamentals in substantial engineering projects; *specialization* , in which students develop and apply in-depth knowledge in their chosen fields; and *realization* , in which students bring their education to bear on problems approaching professional practice. In all three phases of the curriculum, students are engaged in interdisciplinary engineering projects that require them to put theory into practice, to put engineering in context, and to develop teaming and management skills." *http://www.olin.edu/academics/curriculum.aspx* *http://www.olin.edu/academics/olin_history/cdmb_report.html*
Stanford	Tracks	Stanford's track system provides a broad set of options for exploring different interests, with a great deal of flexibility ranging from an unspecialized track for those students who don't find a single area they like best, to eight approved tracks, to individually-designed tracks. *http://cs.stanford.edu/degrees/undergrad/Tracks.shtml*
	Traditional Joint Major	"The Symbolic Systems Program (SSP) focuses on computers and minds: artificial and natural systems that use symbols to represent information. Symbolic Systems' affiliated faculty come from several departments at Stanford University, including Computer Science, Linguistics, Philosophy, Psychology, Communication, and Education." *http://symsys.stanford.edu/*
Virginia Tech	Tracks	"There are five advisory tracks organized around a particular theme or sub-topic in computer science. : Human Computer Interaction; Knowledge, Information and Data; Media/Creative Computing, Scientific Computing; and Systems and Networking. Completing a track is not a requirement for graduation, but it allows a student to focus their undergraduate studies in an area of particular interest or prepares them for a particular career or graduate school option."

		http://www.cs.vt.edu/undergraduate/tracks

Mastery and Research

A specialization based on a set of courses can help students develop a sense of mastery, but may not instill a sense of excitement for a subject or a drive to pursue further studies in graduate school. We are strong supporters of undergraduate research experiences – working with faculty and graduate students in a shared endeavor – as well as of summer internships in research or development activities. It is appropriate, indeed desirable, for some academic credit to be awarded for the experience, particularly when participants write a report reflecting on the experience or help in writing a research paper. In *Recommendation 6 - Attracting, Selecting, and Preparing Students for Research Careers*, we describe in more detail the process of identifying and developing good research candidates.

Recommendations

Specialized computing, through domain-centered tracks:

- We encourage schools to develop a **broad series of specializations**. The specifications reflected in the undergraduate course offerings of a given department will, of necessity, be based on department faculty interests and capabilities and the availability of courses in other relevant disciplines.
- The concepts that should guide the specializations include considerations of the ways in which the components of a particular sequence **build the skills and mastery** needed post-graduation, and how graduates will be viewed by graduate schools and potential employers.
- There is also a question of **scale** – departments with small faculties and student enrollments are clearly not able to offer as many tracks as larger departments. Concern with "break-even" course sizes are a necessary pragmatic. So, the number of tracks, and their depth, will vary quite a bit; small, resource-limited departments may not be able to do this at all, although as described in the chart above, Harvey Mudd, with an enrollment of only 700, provides a counter-example of the creative use of resources to create specialized approaches.

Recommendation 4: Specialization - Integrated Joint Majors - *deep collaboration among disciplines*

Problem

In the not so distant past, many departments viewed each other as engaged in a zero-sum game with regard to which department "owned" which course; does machine learning "belong" in applied math or in computer science? Such questions reflect past attitudes of silo-based knowledge, applications, and departmental fiefdoms. In recent years, the spread of computational approaches into virtually every disciplinehas brought an intrinsically multidisciplinary element to the field of computer science which is not yet fully reflected at the curricular level.[72] In an emerging field whose foundations involve both computational and non-computational subject matter, the practical benefits of training in both areas is clear. In addition to this practical imperative, however, there is a distinct advantage for a future researcher to have training in how to interact with different academic cultures, skills, and mindsets.

One approach to the realization of such multidisciplinary training is the introduction of what we refer to here as "integrated joint majors." Our use of the qualifier "integrated" is meant to distinguish such programs from "double majors" (and from "joint" programs of a similar character) that require students to take coursework in two departments, but without any unified structural and intellectual framework that ties such studies together. By way of contrast, an integrated joint major would be one that involved substantial collaboration between departments to (a) decide what knowledge and skills are to be imparted, (b) design the curriculum and requirements, (c) evaluate how well the program is working, and (d) make improvements adaptively. Such a program might involve the introduction of at least some new multidisciplinary courses, and faculty advisors might be associated with the program itself, and not just with one of its constituent departments.

Many barriers exist, however, to the implementation and successful operation of truly integrated multidisciplinary joint majors. Below we identify certain potential pitfalls and "failure modes" of such programs, along with some possible strategies for their mitigation.

Potential Pitfalls and Mitigation Strategies

The joint major provides a solid foundation in only one of the two areas.

Example:

Not enough methodological knowledge in one area and not enough application knowledge in the other.

Mitigation strategy:

Look at both areas to determine the central core of methodological and application knowledge, then focus on that core, eliminating less essential areas as necessary.

The joint major is not feasible or productive at the undergraduate level

Example:

A major in which meaningful work requires too many

prerequisites, or prerequisites that are too challenging for most of the targeted students

Mitigation strategies:

- Offer the joint major only as part of a five-year combined B.S./M.S. program
- Increase the time spent on prerequisites and decrease exposure to applications. (This may come

[72] Computational programs and minors at CMU provide a foretaste of the institutional impact of computational approaches: computational and applied mathematics, computational biology, computational chemistry, computational design, computational economics/computational finance, computational linguistics, computational mechanics, computational neuroscience, computational physics, and computational and statistical learning.

at the cost of a less compelling program, however, and may be less appropriate for a student pursuing a terminal degree or graduate/professional work in another area.)

The joint major is not well understood outside of the offering institution

Issues:

- Will graduates of the joint program in question find it harder to find a job, gain admission into a strong graduate department, or switch to another field later on?
- Information asymmetry: Potential employers and graduate departments will know less about the program than the student and undergraduate institution.
- Prejudice in favor of known majors: Graduate schools and future employers may assume that the student simply took whatever courses he or she wanted, or may have the perception that joint majors may be easier.

Mitigation strategies:

- Promote external visibility and understanding of the joint major program. (While this approach may be more challenging for smaller, lower-profile institutions, a well-designed website may help to level the playing field.)
- Where possible, encourage coordination across different universities offering similar joint majors with respect to the naming and requirements of these programs.

The costs of establishing the joint major proves prohibitive

Issues:

The institutional costs of setting up and maintaining the joint major may be too high in time, money, and/or human resources relative to other priorities that the institution (and/or its faculty) has to meet. Furthermore, the heavyweight process of establishing a joint major may not be flexible enough to adapt to the rapidly changing nature of technology, the global economy, and student interests.

Mitigation strategies:

Explore more

lightweight approaches within specific departments while encouraging close cross-departmental ties with the related departments. Cross-departmental seminars, team-based research, and joint research facilities all provide mechanisms for increasing collaboration and cross-departmental culture exchanges without the high cost of a full joint major. A further advantage is the flexibility and adaptability these mechanisms provide.

Goals

A fully integrated joint major should:

- Address the needs of students whose interests are in computational approaches to another discipline such as biology, finance, journalism, or music. This differs from the specialized tracks described in Section 3 in that the focus is more on the application domain, and courses will more strongly reflect that balance.
- Enable students to find a job or gain admission to graduate school in both computational fields and in other fields where computation is and/or will be important. This goal impacts careers in research areas of academia, industry, and government (e.g., National Laboratories), as well as in application areas in corporations, startups, not-for-profits, and public service and policy.
- Provide a durable foundation of knowledge and skills that is likely to remain relevant and useful 10 to 20 years from now.

Solutions - Mechanisms for Implementing the Goals

Joint majors usually evolve from earlier collaborative ventures and shared research interests. The earliest instantiations of the idea typically involved identifying a set of existing courses from two or more departments and identifying sequences through them that met appropriate multidisciplinary educational objectives. More recently there has been a growing, and we think healthy, interest in developing new multidisciplinary courses around which joint majors can be built.

A deep understanding of both cultures individually and of their nature when merged into a blended culture is key to the effective development of joint majors. Explicitly identifying the components of the computer science core *[See Recommendation 2 - Core/Foundation for All Computer Science Graduates]* establishes a solid foundation for this process, ensures that the computational dimension is not diluted, and helps the domain culture to effectively incorporate the computational dimensions.

Within the spectrum of cross-disciplinary efforts, three stages of development can be identified. Appendix X includes prototypes for each of the stages described below. Here we include one example: computational biology.

Existing – There are well established multidisciplinary areas, e.g., computer engineering and scientific computing. In these areas there are existing groups who have already designed joint majors.

Emerging - Several institutions have implemented examples of these, but they are not as well understood and accepted, so there will need to be more consensus-building among the faculty and more curriculum design efforts to fit the new program within the context of the specific institution. Examples include digital arts and media, and computational finance.

Opportunities - Possibilities for such programs are being explored but have just begun to be implemented. The implications and value of such programs need more research, so there needs to be even more design and consensus-building as well as the addition of new faculty and new infrastructure. Examples include law and humanities areas such as history and philosophy.

Prototype: Computational biology

Departmental involvement:

Computer science

Biology

Possibly one or more faculty members from, for example, chemistry, physics, applied mathematics and/or statistics who have related research interests

Distinctive subject matter (required and/or elective):

Bioinformatics and systems biology

- Exposure to simple applications of statistical analysis and data mining
- Use of large databases containing genomic, proteomic, and other biological data

Biomolecular modeling and simulation

- Focus on molecular structures and dynamics
- Use of iterative numerical algorithms
- Applications of basic computational physics (mechanics, electrostatics, etc.)

Visualization of biological data

- Graphical representation of statistical data and network models
- Abstraction, rendering, and animation of three-dimensional molecular structures

Recommendations

Recommendaton 4: Specialization - Integrated Joint Majors

Deep collaboration among disciplines, exemplified by integrated joint majors:

- Coherent, integrated multidisciplinary, interdepartmental joint majors provide a **balanced approach** that addresses the differences in culture, concepts, and strategies between different fields by establishing the common ground between them.
- To undertake the considerable resource cost of joint major development, we recommend encouraging—indeed, incentivizing—the additional faculty effort required to design and implement new integrated courses and curricula. **Incentives** could include summer salary, release time, the designation of a dedicated 'curriculum czar', and the supervision of students with appropriate prior background and motivation as research and teaching assistants to help with design and implementation
- **Initial exploration** of collaborative possibilities can include multi-disciplinary curriculum committees, individual experimental courses, support for multi-disciplinary GISPs (Group Independent Study Projects), plus Internet-based collaboration using existing tools such as wikis.

Building Mastery Throughout the Curriculum

Overview

The recommendations in *Section II - Refactoring the Computer Science Curricula* concerned the curricula restructuring necessary to support the 21st century environment of rapid change. This section, Section III - *Building Mastery Throughout the Curriculum* focuses on the strategies needed to develop mastery: depth in the understanding of cognitive skills, concepts, and techniques that will support researchers through graduate school and throughout their careers.

As we describe the process in *Recommendation 5 - Design under Constraints and the Gaining of Mastery*, mastery involves: "...(1) learning how to do things in better, simpler, and more aesthetically pleasing ways, (2) developing "wizardry" via learning increasingly sophisticated tricks of the trade, and (3) computing with a purpose, by understanding an application and its requirements, and the impact it can have on people and their lives, that motivates the technology they are learning."[73]

Some of the mastery components include creating abstractions from data and establishing relationships among them, building, simulating, and validating models, working with the tradeoffs involved with different representations, and dealing with different levels of detail. These components are needed by all students as preparation for whatever they do in life, whether they find themselves becoming computational practitioners, researchers, or following some entirely different path.

In addition, researchers need specific preparation and skills that should be addressed from the introductory courses right on through the entire period of their undergraduate career. The pursuit of mastery begins in the foundational material and skills of introductory courses (described in *Recommendation 1 - Introductory Courses*), influences the decisions to pursue a research career, and guides the undergraduate preparation needed by future researchers.

Thus, the goal of mastery support is to create a coherent environment that begins with introductory courses that capitalize on domain enthusiasms to introduce students to important cognitive skills and computational content, and then deepens undergraduate cognitive skills in a variety of contexts over the entire four years of the undergraduate program.

Note that attracting and educating future researchers requires an apprenticeship framework that formerly had characterized just the graduate-level programs. This means access to collaboration with professors and graduate students from the beginning rather than positioning the student as passive recipient of knowledge. The student must be seen as active agent, not passive consumer. This section has two recommendation areas:

5. **Design under Constraints and the Gaining of Mastery** - deepening the skill set
6. **Attracting, Selecting, and Preparing Students for Research Careers** - developing computationally-oriented researchers

[73] See Recommendation 5 - Design under Constraints and the Gaining of Mastery.

Recommendation 5: Design under Constraints and the Gaining of Mastery - *deepening the skill set*

Problem

As students proceed through the curriculum they need to deepen their understanding and mastery of cognitive skills and concepts, and to extend their set of skills beyond those they were introduced to in their introductory courses. In addition, during this period they need to be provided with opportunities to focus their intentions on career decisions: do they wish to become computationally-oriented researchers at some point in their careers *[See Recommendation 6- Attracting, Selecting, and Preparing Students for Research Careers]* or do they want to become practitioners?

This paper is primarily concerned with the choices involved in educating researchers, but the gaining of mastery and learning to design under constraints is necessary for both researchers and practitioners. Indeed, it is critical for all human endeavors. Note also that while this section talks in terms of computer science, the comments and recommendations apply to all computationally-oriented fields.

Students frequently fail to understand the broad applicability of the design skills and implementation abilities they have acquired because of the compartmentalized way in which computer science is currently taught. Gaining mastery and learning how to design systems under real-world constraints is critical for their success. Problems with the current computer science curriculum include:

- Knowledge is too compartmentalized and students lack opportunities for synthesis and integration across the sub-disciplines of computing. Also, existing courses give students inadequate exposure to human-centric issues, particularly in terms of how and for what purpose information technology systems are used.
- There is a gap in time and order of presentation between the introduction of a concept and its application in context. This yields a sense of "technology for technology's sake" that can give students a misleading view of the field early in their education and perhaps dissuade them from becoming computer scientists.
- There is inadequate experience with many aspects of real-world system design, including open-ended design under constraint, social context of design teams, systems-level thinking, complex systems design, and nondeterministic system behavior.

Accreditation bodies, such as ABET (Accreditation Board for Engineering and Technology)[74], demand that curricula include integrative experiences, and most departments have such courses. The problem is that these so-called "capstone design" courses are encountered by students finishing their programs, not earlier when such courses can have a positive influence on the trajectory of a student's studies.

Goals

Our goal is to better motivate students to enter and remain in the field of computer science and go into research by experiencing the excitement and satisfaction that comes from gaining mastery of a field. We define *mastery* as (1) learning how to do things in better, simpler, and more aesthetically pleasing ways, (2) developing "wizardry" via learning increasingly sophisticated tricks of the trade, and (3) computing with a purpose, which involves understanding an application as well as its requirements, and the impact it can have on people and their lives to motivate learning.

To better achieve this motivation, the teaching of the field requires:

- **Gaining experience**: learning new technologies and techniques in a context that considers how they will be applied, in what way, and for what purpose. This includes identifying assumptions and understanding the implications of model limitations.

[74] http://www.abet.org/

- **Building real artifacts**: integrating knowledge gained by constructing real systems whose performance and effectiveness can be measured. This includes understanding the pitfalls of validation methods.
- **Designing as an iterative process**: building, evaluating, —and throwing away—initial prototypes, to better understand the space of possible designs, their implementations, and the tradeoffs in the solution, performance, and cost design spaces.
- **Experiencing projects that tap into enthusiasms**: engaging in projects that provide the student with opportunities to develop creativity, have societal impact, and leverage strong extra-curricular interests.
- **Debugging/**dealing with real-world issues: building systems of a size and scale that confronts students with the challenges of making distributed systems actually work. This comes in the later part of the undergraduate curriculum.

It should also be clear that these skills and experiences are as important for the development of outstanding graduate researchers as for developing excellent undergraduates who intend to become system designers for industry.

Mechanisms for Implementing the Goals

In this subsection, we suggest a set of educational experiences that motivate students to incrementally develop increasing mastery of the field, through substantial design experiences that allow them to integrate knowledge through the construction of real computer-based artifacts.

Integrating Design Early in the Curriculum[75]

"Design" is an intrinsic component of many computer science courses. However, the design experience is often constrained and overly specified in such a way that it limits room for creativity by the student in formulating the artifact to be designed or the solution approach.

Furthermore, the design project is often computer science discipline-specific, such as a compiler, an operating system, or a hardware control system, rather than a complete, albeit modest, system. Course design projects rarely involve large teams even in the university context, sometimes work on their own, although groups of two students are frequently the norm. Thus, students rarely gain experience in working in teams, an important practical element of an engineering career.

Examples of Early Design Experience

MIT 6.01 - Introduction to EECS I	First course in electrical engineering and computer science, at MIT, which combines computing and sensing/actuating in a "first" course in computer science and electrical engineering, crossing these disciplines in order to construct a physical artifact, in this case a mobile robot. http://mit.edu/6.01/mercurial/spring10/www/index.html
Berkeley EE 120 - Signals and Systems	Sophomore/junior course in signal processing and system-level thinking at Berkeley. This course presents an introduction to signal processing in the context of a robot design project. http://www-inst.eecs.berkeley.edu/~ee120/fa08/

These appear to be successful integrative courses because they combine theory with hands-on reduction to practice in the form of a real-world and exciting project.

[75] [NRC 2009], the new NRC report on Engineering Education for K-12 makes the point that engineering is about design, e.g. creativity.

Capstone Design Experiences

A more open-ended design experience, incorporating a project of substantially larger size and complexity, has traditionally been reserved for the end of the undergraduate program. These are commonly known as capstone design courses.

Often they are in specific subdisciplines of the field, such as software engineering or embedded systems, although the projects can be selected from an extensive range of applications.

Software engineering capstone courses typically cover the following critical aspects of building software-based computing systems. They differ from advanced programming courses in the scope of their integrated focus that draws upon the undergraduate experience:

- Designing and implementing a software system to "a customer specification," sometimes iteratively determined.
- Development and specification of service-level objectives.
- Evaluation of systems to demonstrate that implementation meets its specification.
- Working in larger-scale teams, sometimes with a project leader (perhaps an MBA or School of Information student).
- Experience in formal presentations.

Examples of Software Engineering Capstone Courses

Berkeley CS169 - Software Engineering	Unusual in the computer science curriculum in that students are involved in a large-team project. They experience what it takes to collaborate with people with different skills and approaches to software development. The project is substantially open-ended: the students select the topics of the projects and almost all aspects of development (programming language, libraries, build environment, etc.) http://www-inst.eecs.berkeley.edu/~cs169/sp10/doku.php?id=info
Harvey Mudd - CS and Engineering Clinic[76]	A year-long upperclass program in which teams of 4-5 students, faculty advisors, and industrial liaisons work with company-specified requirements to propose, create, test, and demonstrate solutions. Some "Clinic" projects are multidisciplinary and are jointly run by several HMC departments. http://www.eng.hmc.edu/EngWebsite/index.php?page=Clinic.php

Embedded systems courses offer an alternative experience in capstone design, but the fundamental design and project management processes remain the same. The integrative project focuses on combining processing with sensing of the real world. This kind of course requires students to consider real-world constraints such as limited volume, payload, electrical power, processing power and time. Oral and written reports justify design choices. Students are evaluated in part by the by quantitative performance and robustness of their designs.

Such courses typically involve the following:

- An integrative design experience that spans hardware, software, sensing, actuating, and control.
- Some kind of robotic application, but at a higher level of complexity than MIT 6.01 or Berkeley EE 120.
- Real-world constraint-based design, including being limited to a bag of parts, confronting size + power limitations, etc.

Example of Embedded Systems Capstone Courses

Berkeley EE 192 - Mechatronics Design Laboratory	A typical class project is to design racing robots that can follow an embedded wire over a curving and self-crossing

[76] [Harvey Mudd 2007] "Engineering Department. Engineering clinic handbook"

	path at speeds greater than 3 meters per second. Each team starts with a radio-controlled car and a predesigned CPU/FPGA board and undertakes the design of sensors, electronics, and control algorithms to develop a winning optimal strategy. National Semiconductor Corporation sponsors a contest that pitted teams from different schools against each other. This adds the real-world element of competition into the experience. http://www-inst.eecs.berkeley.edu/~ee192/sp10/

"Computing for Good" Capstone Courses: "Computing for Good" capstone project courses, sometimes called "service learning" courses, explicitly focus on societal impact of technology. We note that such hardware-software co-design courses may be difficult to duplicate in computer science departments that lack computer engineering programs. Nevertheless, creating a course with a focused design project, with well-specified performance goals and constructed from constrained building blocks, is also possible in software engineering courses and other software-intensive courses.

Example of "Computing for Good" Capstone Courses

Georgia Tech's C4G optional course for upper division undergraduates	C4G centers on the concept of applying computing to social causes and improving quality of life. It draws on the self-focused and altruistic sides of students by presenting computer science as a cutting-edge discipline that empowers them to solve problems of personal interest as well as problems important to society at large. http://www.cc.gatech.edu/about/advancing/c4g
Purdue's EPIC (Engineering Programs in Community Service) program	EPICS is a unique program in which teams of undergraduates are designing, building, and deploying real systems to solve engineering-based problems for local community service and education organizations. EPICS was founded at Purdue University in Fall 1995. https://engineering.purdue.edu/EPICS/About

Design Studio Courses

These kinds of courses combine the open-ended design experience of the software engineering courses with the real-world constraints of the embedded systems courses. While the design theme of the course is determined early - one example being designing systems to assist physically-challenged individuals - the instructor is actively engaged with individual student teams to define feasible projects.

Often these courses are structured to span quarters or semesters to allow substantial time for project formulation, followed by sufficient time for project implementation. Artifacts created during one sequence may be reused by later sequences, increasing in scale and complexity on a year-by-year basis. They typically develop the following:

- Skills in reverse engineering, construction and use of reusable building blocks, reading and reusing code and libraries, elements that persist and are enhanced from semester to semester.
- An understanding of design as a continuous learning process, i.e., design thinking: define, research, ideate (brainstorm on design approaches), prototype (explore the space to refine the design), choose, implement, and learn.
- Student mastery in the development of a design portfolio of implementation projects with a graded assessment that indicates the level of proficiency in skills obtained. This is now required for ABET accreditation.

Examples of Design Studio Courses

Berkeley CS194 - Internet of Everyday Things	The theme of the course is to tear everyday appliances apart and put communication and programmability into their core, allowing them to become software controlled, interact with other devices, integrated with powerful servers, and able to draw from or provide information to the web. With these new capabilities, the students investigate what interesting and unusual things they can do with these integrated, scriptable, and now networked devices. The course spans elements of hardware design, embedded systems software, networking, server integration, web services, and use-case and business analysis. http://www-inst.eecs.berkeley.edu/~cs194-5/sp08/
Berkeley CS 98/198 - GamesCrafters	A unique element of this course, which can be repeated many times, is that it is open to students at levels from beginners to experts, allowing the more junior students to learn "the tricks of the trade" from more experienced students, the latter also developing their own teaching and mentoring skills. http://www.eecs.berkeley.edu/Courses/Data/485.html

Common Themes and Methods

The common themes and methods that emerge in these courses are:

- Integrate - even in the first course in computing - an individual or small team **design experience** that makes concrete the foundational concepts being introduced in the course.
- Build on **student-accumulated knowledge and experience** to permit greater choice in the formulation of individual projects, even if they remain constrained in theme (e.g., a game, a race car, a networked appliance) or choice of implementation approach (e.g., software-only, embedded system, etc.).
- Introduce **human-centric design experiences** through projects that involve human-directed inputs based on system communicated outputs. Encourage the development of communications skills through frequent design presentations, project write-ups, and engagement within the project team and with external "customers."
- Introduce **constrained design** by imposing user requirements and limitation to the implementation approach on the projects. Evaluate projects not simply on functionality (e.g., meeting functional requirements), but also on performance, cost/performance, and elegance (e.g., fewest components, lines of code, etc.).
- Use **cooperation** (e.g., sharing tools and libraries) and **competition** (e.g., bragging rights) where appropriate to undertake larger design or to motivate students.
- Re-conceptualize curriculum around the **project focus** to introduce materials "just in time" to be useful in project design and implementation.
- Develop the capacity for professional **teamwork.**

Recommendations

Deepen the skill set and the pursuit of depth and mastery through attention to design under constraints across the full undergraduate spectrum. Attaining mastery entails gaining experience in learning new technologies and techniques, building real artifacts, and learning to understand design as an iterative process. Design and development experiences should tap into the actual interests of the students within a structure that both rewards effort and requires debugging/dealing with real-world nondeterminacy.

Specific recommendations include the following:

- Provide **integrative design experiences** earlier in the curriculum, including the first course, and throughout the curriculum, building on the student's increasing skills.

- Incorporate **skill descriptions** in addition to course topics in all courses. Articulate how the elements of mastery, wizardry and purpose form part of the course outcomes.
- Integrate **success stories of project integration across the curriculum** into individual courses, to better leverage instructor time and resources. Make course developments widely available on the web, so that others may use, adapt, and extend them for their own courses and for the community at large.

6. Attracting, Selecting, and Preparing Students for Research Careers - *developing computationally-oriented researchers*

Problem

Computing researchers know that computing is an exciting area that that touches our lives directly on a daily basis, greatly impacts all scientific and other quantitative disciplines, and offers rich opportunities for multidisciplinary research in other fields such as medicine, business, and the arts and humanities. Computing research advances in the last two decades have changed the way we work, play, learn, and communicate.

In this section we discuss four questions that need to be addressed as we consider attracting and preparing students for computationally-oriented research careers in the 21st century.

- How do we **identify** those students best suited to research careers, or how do they self-identify?
- How do we **attract** these students?
- What **skills** do they need above and beyond those we've described earlier in this report?
- How should we **prepare** them for this goal, starting with the introductory courses and moving through the curriculum?

Goal

To identify and attract the best candidates for computationally-oriented research careers, and to prepare them to succeed in getting into graduate school, as the essential gateway into such a career.

Solutions - Mechanisms for implementing this goal

How do we identify those students best suited to research careers?

- We need to look for students who manifest curiosity, motivation, persistence, tolerance for frustration, courage, and a desire to explore new areas. Perhaps most importantly, future researchers should stand out as idea-generators because otherwise they can't make strong contributions to advancing research frontiers.

How do we attract them to computationally-based research careers?

- Attracting students to research careers starts in the introductory courses with research-oriented professors who are accessible to undergraduates. Attraction is personal and is enhanced by the excitement of professors who infuse lectures with research questions[77] and ideas and assignments with simplified versions of research problems, etc.
- Students need opportunities to learn what research is and how to do it; without those opportunities it is a totally abstract concept. The mechanisms for providing these include - among others - in-class descriptions of the research process relative to the topic under consideration, professors who are willing to work with a class on areas with which they are still grappling, departmental and instructor webpages about what the current research work is and what qualities are looked for in undergraduate collaborators[78].
- Further, students need to be made aware that advances in computing were responsible for many

[77] "In my first year as an undergraduate back in Brazil, one of my professors said, 'There's this thing called the Web. It's growing rapidly and it's raising lots of interesting problems.' So I started reading and studying about it," Fonseca said, "and my interest grew along with it. That was 1997." in [Nickel 2009]"Rodrigo Fonseca Assistant Professor of Computer Science"

[78] See webpage of Prof. John Hughes, Brown University for an example: http://www.cs.brown.edu/~jfh/working/working.htm

of the most important life-changing innovations of the past 30 years and that there is tremendous opportunity for future computing research advances, in for example, creating the future of communication, empowering developing countries, and revolutionizing predictive, preventative, and personalized medicine.

What cognitive skills do they need above and beyond those we've described earlier in this report?

- Ability to read different authors with different perspectives, assumptions, vocabularies, etc. and do an 'apples to apples' comparison. The 'related work' sections of research papers reflect this skill. This is a higher form of abstraction-creation and is a form of pattern-finding synthesis skill.
- Ability to identify hidden assumptions, in oneself and in other researchers.
- Ability to balance vision (seeing a new or 'big' picture) with great detail/rigor skill. This involves being able to handle multiple levels of detail within an (un)common framework.
- Ability to be constructively dissatisfied with 'what is', to challenge, to ask 'what if...', to come up with new ideas in response to failure.

How should we prepare them for this goal?

- The fundamental approach is to provide an apprenticeship environment from introductory courses through graduation that helps students assimilate the computationally-oriented researcher mindset. The long-term exposure is important because the attitudes and cognitive skills must be learned over time in many contexts.

Introductory courses:

- Possible approaches include working with grad students in labs, references to research-level problems in class, in assignments, and in labs; and providing examples of the process through articles, such as the interview with David Shaw in the October CACM[79] and interviews with Turing Award winners such as Barbara Liskov[80].

Mastery courses, all levels:

- Students at all levels, from freshmen through seniors, can work in research labs with graduate students and their professors, making contributions, and then participate in writing and reviewing proposals, research papers, and research sponsor visits, attending symposia and conferences, perhaps including presentations of poster sessions, and even papers.
- One interesting example of using a senior level course as a researcher-training course is Brown's CS237 - Interdisciplinary Scientific Visualization[81]. In this class students work in small multidisciplinary groups, to identify scientific problems, propose solutions involving computational modeling and visualization[82], design and implement the solutions, apply them to the problems, and evaluate their success. As part of the process they investigate related work, write papers and proposals, and demo their projects. Some of these papers have been accepted to major conferences, and some of the work has gone on to be the basis for more significant projects in the area.
- Purdue's Doctoral Program illustrates another approach to training future researchers: first-

[79] [Shaw 2009] "A conversation with David E. Shaw"

[80] [Frenkel 2009] "Liskov's creative joy"

[81] David Laidlaw - http://www.cs.brown.edu/courses/csci2370.html

[82] Visualization techniques grow increasingly critical as massive data sets become the norm. Brian Cantwell Smith in "On the Origin of Objects", p. 19 points out that "It has even been speculated that the entire field of non-linear dynamics, popularly called 'chaos theory', could not have happened without the development of such [graphics] displays. No one would have 'seen' the patterns in textual lists of numbers." All of Edward Tufte's books are valuable resources, as are the facilities of MATLAB and Mathematica.

year graduate students must take one research methods course and two research practicum courses, followed by a series of written and oral qualifying exams.[83]

Integration into a researcher-oriented environment

We believe that students will find it valuable to capture their insights and ideas from the very beginning of their undergraduate career, including records of their projects, journals of their process and future plans, notes from class discussions, as well as materials obtained during the class. In the past, spiral notebooks typically served as repositories of the experience of a particular class, but accessing this handwritten material was difficult, incomplete, and less than satisfactory as an ongoing portfolio. Research environments frequently require researchers to maintain dated and page-numbered bound lab notebooks as legal documentation for potential patent applications and while some advanced courses provide training in their use, they also fall short of a complete and extensible digital record. Encouraging students to form the idea of a digital portfolio from the beginning of their first introductory classes could not only provide a persistent and useful tool over time, enabling them to evolve project notions from year to year, it could also become the basis for an emergent sense of themselves as researchers and, ultimately, provide a record for future papers and graduate school work.

Recommendations

Attract students to research careers through the introduction of research approaches and skills across the full undergraduate spectrum:

- Combine explicit **research skill training** with an **apprenticeship approach** to acculturate future researchers to their community of practice. This means systematic guidance in the practices of computationally-oriented research from freshman year through graduation combined with the support provided by close relationships with graduate students, research groups, and professors.
- Attract and prepare the best qualified students by exposing them to and engaging them in **exciting computing research** – the earlier, the better.
- **Engage undergraduates in research** via various means: focused study groups that include graduate students, student-initiated GISPs (Group Independent Study Projects), seminars that undergraduates are encouraged to attend, undergraduate research assistantships, summer programs and internships, university-sponsored internships, and special scholarship programs, such the University of Utah Access Program[84] and Engineering Scholars Program[85].
- Facilitate the creation and use of a **persistent digital portfolio** from the beginning of their first introductory classes, continuing through all of their ongoing courses, to provide both an idea resource base and a record for future papers and graduate school work
- Emulate the model for project-oriented courses provided by the **domain-specific introduction to graduate research methods** (Brown's CS237 - Interdisciplinary Scientific Visualization), described in the "How should we prepare them for this goal?" section above.
- Provide **small classes** including a large percentage of qualified students to enable significant teacher interaction for the students.

Finally, several CRA reports that address the issues of recruiting and retaining graduate students have suggestions [86] that are relevant to attracting and preparing undergraduates for research careers.

---

[83] http://www.cs.purdue.edu/academic_programs/graduate/curriculum/doctoral.sxhtml#Research

[84] http://www.science.utah.edu/access.html

[85] http://www.coe.utah.edu/current-undergrad/esp.php

[86]]**[CRA 2006b] Graduate Recruitment & Retention in CSE** - http://people.virginia.edu/~jlc6j/gradrr/

This research studies the effectiveness of computer science departments' efforts to recruit and retain women graduate students.

[CRA 2007] CRA Taulbee Survey - http://www.cra.org/resources/taulbee/

The Taulbee Survey is the principal source of information on the enrollment, production, and employment of Ph.D.s in computer science and computer engineering (CS & CE) and in providing salary and demographic data for faculty in CS & CE in North America.

[CRA 2006a] Recruiting and Retaining Women Graduate Students in Computer Science and Engineering - http://www.cra.org/uploads/documents/resources/workforce_history_reports/gradrr07.pdf

This NSF-funded study was initiated to test the validity of an earlier report, "Recruitment and Retention of Women Graduate Students in Computer Science and Engineering" (Cuny and Aspray, 2001). It summarizes and expands on the results of a 2006 workshop and outlines research-based practices likely to promote gender balance in graduate computing programs.

[CRA 2000a] Recruitment and Retention of Women Graduate Students in Computer Science and Engineering - http://www.cra.org/uploads/documents/resources/workforce_history_reports/rrwomen.pdf

The report, written by Jan Cuny (U. of Oregon) and William Aspray (CRA), is the result of a workshop that was held in June, 2000. Workshop participants included long-time members of the CSE academic and research communities, social scientists engaged in relevant research, and directors of successful retention efforts. The report's goal is to provide departments with practical advice on recruitment and retention in the form of a set of specific recommendations.

[CRA 2000b] Recruitment and Retention of Underrepresented Minority Graduate Students in Computer Science - http://www.cra.org/uploads/documents/resources/workforce_history_reports/rrminorities.pdf

Report of a committee convened by the Coalition to Diversify Computing (CDC). The report offers 25 practical suggestions for departments to consider. Each contains a general discussion followed by a recommended course of action. Examples of successful and promising new programs are given, along with contact information for those who want to explore further.

Appendices

In addition to the Recommendations Summary (Appendix A) and the References (Appendix B), Appendices C and D include a set of essays and outlines by individual authors as well as summary descriptions from course websites. Each exemplar or prototype essay or outline includes the name and affiliation of the author.

Recommendations Summary

Computationally-Oriented Foundations

1. Introductory Courses - addressing a broad range of student interests

Address student interests while at the same time ensuring that these courses address a significant subset of the fundamental range of concepts and skills that comprise computational thinking [See the local appendix at the end of this Introduction: Computational Thinking - Summary of Views].

Use these courses to instill a set of cognitive skills such as learning how to create, validate, and establish relationships among abstractions from data and information on hand, a key skill in effective modeling, simulation, and validation. This skill in working with abstractions, in turn, undergirds both the scientific method and computational thinking, and should be a part of every computationally-oriented course. The differences among such courses help to reinforce the underlying skills as students meet the same concepts in different contexts.

Other examples of cognitive skills include: working with the tradeoffs involved with different representations; moving, where appropriate, from a declarative understanding of a problem to an imperative understanding of that problem; reducing computationally intractable problems to related tractable problems; and building, simulating, and validating computational models that shed light on important questions.

Specific recommendations

- **Introduce students to computational approaches** through foundation courses across the spectrum of student interests, instilling a set of cognitive skills such as those described earlier in the Introduction - "... learning how to create, validate, and establish relationships among abstractions from data and information on hand, a key skill in effective modeling, simulation, and validation.... working with the tradeoffs involved with different representations; moving, where appropriate, from a declarative understanding of a problem to an imperative understanding of that problem; reducing computationally intractable problems to related tractable problems; and building, simulating, and validating computational models[87] that shed light on important questions."
- **Emphasize the creation of appropriate and useable sets of representations and relationships** among different levels; a deep understanding of how to represent information is one of the most difficult cognitive skills students need to learn. The practice of presenting accessible but important research papers as part of introductory courses not only introduces students to actual research work but also starts to build an understanding of how to compare different representations, analyze unstated assumptions, and build a common representation structure across a set of related projects.
- **Establish collaborative efforts** involving a computer science department, which could assume primary responsibility for the courses, and the departments whose prospective students are expected to take one of the courses. Additionally, the computer science department should help other departments in developing follow-on courses that take advantage of computational thinking taught in the first course.
- **Begin to shape a collaborative student culture** that will mature into effective professional teamwork skills, as described in Recommendations Two through Six.
- **Encourage students to begin building a digital portfolio**, including journal entries,

[87] When we use the terms "model" and "modeling" in this paper we mean symbolic computational models, not numeric models, which are sets of differential equations.

possibly online [dealing appropriately with privacy issues], that carries through the core ideas and can be added to and be available for research ideas, building mastery, understanding one's own perspective, use in applying to graduate school. See Recommendation Six for a more complete description of this idea.

Refactoring Computer Science Curricula

2. Core/Foundation for All Computer Science Graduates - lean core with focus on enduring concepts, techniques, and skills

A relatively lean core emphasizes foundational concepts and skills while allowing students more time to explore areas in depth, both by taking courses and by engaging in undergraduate research. Additionally, a lean core makes it easier for students with multidisciplinary interests to pursue a joint major *[See Recommendation 4 - Specialization: Integrated Joint Majors]* while still sharing a common experience with computer science majors.

Specific recommendations

Lean core with focus on the minimum essential cognitive skills, concepts, and techniques.

- Having a relatively lean core emphasizes **foundational cognitive skills and concepts** while allowing students more time to explore areas in depth, both by taking courses and by engaging in undergraduate research.
- The deep issues of **mastery and skills** faced by the core have strong connections to the issues discussed in *Recommendation 5 - Design under Constraints and the Gaining of Mastery*.
- A lean core makes it easier for students with **multidisciplinary interests** to pursue a joint major *[See Recommendation 4 - Specialization: Integrated Joint Majors],* while still sharing a common experience with computer science majors.
- The explicit identification of the lean core components makes it easier for a wide range of institutions to **identify resources**, establish a strong basic computer science foundation, and help their graduates pursue computationally-oriented research careers.

3. Specialization: Tracks, Threads, and Vectors - flexible approaches to gaining understanding and skills

Define sets of meaningful specializations to permit students to pursue their interests in a context that guides their development while providing strong motivation. Ensure that these 'tracks' are specialized enough that a course sequence can lead to a student attaining some reasonable depth in the area but broad enough that someone in a company or graduate school will be able to fit it into their institutional context.

Specific recommendations

Specialized computing, through domain-centered tracks:

- We encourage schools to develop a **broad series of specializations**. The specifications reflected in the undergraduate course offerings of a given department will, of necessity, be based on department faculty interests and capabilities and the availability of courses in other relevant disciplines.
- The concepts that should guide the specializations include considerations of the ways in which the components of a particular sequence **build the skills and mastery** needed post-graduation, and how graduates will be viewed by graduate schools and potential employers.
- There is also a question of **scale** – departments with small faculties and student enrollments are clearly not able to offer as many tracks as larger departments. Concern with "break-even" course sizes are a necessary pragmatic. So, the number of tracks, and their depth, will vary quite a bit; small, resource-limited departments may not be able to do this at all, although as described in

the chart above Harvey Mudd, with an enrollment of 700, provides a counter-example of the creative use of resources to create specialized approaches.

4. Specialization: Integrated Joint Majors - deep collaboration among disciplines

Coherent, integrated multidisciplinary, inter-departmental joint majors provide a balanced approach that addresses the differences in intellectual culture, concepts, and strategies between different fields by establishing the common ground between them. Use these integrated joint majors to provide a creative synthesis beyond that which can be provided by a computer science department alone, one that blends the cultures and mindsets of multiple departments and synergistically establishes new techniques for problem solving.

Specific recommendations

Deep collaboration among disciplines, exemplified by integrated joint majors:

- Coherent, integrated multidisciplinary, interdepartmental joint majors provide a **balanced approach** that addresses the differences in culture, concepts, and strategies between different fields by establishing the common ground between them.
- To undertake the considerable resource cost of joint major development, we recommend encouraging—indeed, incentivizing—the additional faculty effort required to design and implement new integrated courses and curricula. **Incentives** could include summer salary, release time, the designation of a dedicated 'curriculum czar', and the recruitment of students as research and teaching assistants to help with design and implementation.
- **Initial exploration** of collaborative possibilities can include multi-disciplinary curriculum committees, individual experimental courses, support for multi-disciplinary GISPs (Group Independent Study Projects), plus Internet-based collaboration using existing tools such as wikis.

Establishing Mastery across the Curricula

5. Design Under Constraints and the Gaining of Mastery - deepening the skill set

Provide students the ability to attain mastery by gaining experience in learning new technologies and techniques, building and analyzing artifacts, and learning to understand design as an iterative process that involves evaluating tradeoffs, analyzing system performance, and testing at each step. Create design and development experiences that tap into the actual interests of the students within a structure that both rewards effort and requires debugging/dealing with the uncertainties and approximations of real-world non-determinacy.

Specific recommendations

- Provide **integrative design experiences** earlier in the curriculum, including the first course, and throughout the curriculum, building on the student's increasing skills.
- Incorporate **skill descriptions** in addition to course topics in all courses. Articulate how the elements of mastery, wizardry and purpose form part of the course outcomes.
- Integrate success stories of **project integration across the curriculum** into individual courses, to better leverage instructor time and resources. Make course developments widely available on the web, so that others may use, adapt, and extend them for their own courses and for the community at large.

6. Attracting, Selecting, and Preparing Students for Research Careers - developing computationally-oriented researchers

Skillfully introduce research problems and their intellectual excitement in all courses, thus helping to entice potential research students by disabusing them of the notion that our field has become routinized. Successful courses that attract and excite students present new concepts within the context of the ongoing research of the R&D community.

Combine explicit research skill training with an apprenticeship approach to acculturate future researchers to their communities of practice. Provide systematic guidance in the practices of computationally-oriented research from freshman year through graduation combined with the support provided by close relationships with graduate students, research groups, and professors.

Specific recommendations

Attract students to research careers through the introduction of research approaches and skills across the full undergraduate spectrum:

- Combine explicit **research skill training** with an **apprenticeship approach** to acculturate future researchers to their community of practice. This means systematic guidance in the practices of computationally-oriented research from freshman year through graduation combined with the support provided by close relationships with graduate students, research groups, and professors.
- Attract and prepare the best qualified students by exposing them to and engaging them in **exciting computing research** – the earlier, the better.
- **Engage undergraduates in research** via various means: focused study groups that include graduate students, student-initiated GISPs (Group-Independent Study Programs), seminars that undergraduates are encouraged to attend, undergraduate research assistantships, summer programs and internships, university-sponsored internships, special scholarship programs, such the University of Utah Access Program[88] and Engineering Scholars Program[89].
- Facilitate the creation and use of a **persistent digital portfolio** from the beginning of their first introductory classes, continuing through all of their ongoing courses, to provide both an idea resource base and a record for future papers and graduate school work
- Emulate the model for project-oriented courses provided by the **domain-specific introduction to graduate research methods** (Brown's CS237 - Interdisciplinary Scientific Visualization), described in the "How should we prepare them for this goal?" section above.
- Provide **small classes** including a large percentage of qualified students to enable significant teacher interaction for the students.

[88] http://www.science.utah.edu/access.html

[89] http://www.coe.utah.edu/current-undergrad/esp.php

References

Bibliography

[ACM CSTA 2006] ACM CSTA (Computer Science Teachers Association). *A Model Curriculum for K-12 Computer Science: Final Report of the ACM K-12 Task Force Curriculum Committee, Second edition*. http://www.csta.acm.org/Curriculum/sub/CurrFiles/K-12ModelCurr2ndEd.pdf

[ACM 2010] ACM Curriculum Reports. http://www.acm.org/education/curricula-recommendations

[ACM/IEEE 2005] ACM/IEEE Computing Curricula 2005: The Overview Report. http://www.acm.org/education/education/curric_vols/CC2005-March06Final.pdf

[ACM/IEEE 2008] ACM/IEEE Computer Science Curricula 2001 Report/2008 Update. http://www.acm.org education/curricula/ComputerScience2008.pdf

[Adler & Van Doren 1972] Mortimer J. Adler and Charles Van Doren. *How to Read a Book: The Art of Getting a Liberal Education, Revised Edition*. Touchstone, 1972.

[AHC 2005] Association for History and Computing. *Humanities, Computers and Cultural Heritage: Proceedings of the XVI International Conference of the Association for History and Computing*. Royal Netherlands Academy of Arts and Sciences, Amsterdam. http://www.knaw.nl/publicaties/pdf/20051064.pdf

[Allen et al. 2010] Vicki Allan, Valerie Barr, Dennis Brylow, and Susanne Hambrusch. "Computational thinking in high school courses" in *Proceedings of ACM SIGCSE '10*.

[Barr et al. 2010] Valerie Barr, Chun Wai Liew, and Rich Salter. "Building bridges to other departments: three strategies" in *Proceedings of ACM SIGCSE '10*.

[Boonstra et al. 2004] Onno Boonstra, Leen Breure, and Peter Doorn. *Past, Present and Future of Historical Information Science*. Radboud University of Nijmegen. http://www.niwi.knaw.nl/nl/geschiedenis/onderzoek/onderzoeksprojecten/past_present_future_of_historical_information_science/draft_report/toonplaatje

[Boyd 2008] Danah Boyd. *Taken Out of Context: American Teen Sociality in Networked Publics*. PhD Dissertation. University of California-Berkeley, School of Information. http://www.danah.org/papers/TakenOutOfContext.pdf

[Bransford et al. 1999] John D. Bransford, Ann L. Brown, and Rodney R. Cocking, editors. *How People Learn: Brain, Mind, Experience, and School*. National Academy Press.

[Breck et al. 2008] Eric Breck, David Easley, K-Y Daisy Fan, Jon Kleinberg, Lillian Lee, Jennifer Wofford, and Ramin Zabih. "A New Start: Innovative Introductory {AI}-Centered Courses at Cornell" in *Proceedings of 2008 AAAI Spring Symposium on Using AI to Motivate Greater Participation in Computer Science*. http://www.cs.cornell.edu/home/kleinber/aaaiss08-courses.pdf

[Brown 2008] John Seely Brown. *Tinkering as a Mode of Knowledge Production*. Carnegie Foundation for the Advancement of Teaching video, October 2008. http://www.johnseelybrown.com/

[Bruner 1960] Jerome Bruner. *The Process of Education*. Cambridge, MA: Harvard University Press.

[Catmull 2008] Ed Catmull. "How Pixar Fosters Collective Creativity" in *Harvard Business Review*, September 2008. http://corporatelearning.hbsp.org/corporate/assets/content/Pixararticle.pdf

[CBMS 2001] Conference Board of the Mathematical Sciences. *The Mathematical Education of Teachers: Issues in Mathematics Education, Volume 11*.

[Cornell 2010] *Cornell University curriculum change*. http://www.cs.cornell.edu/ugrad/CSMajorTransition08-09.htm; http://www.cs.cornell.edu/ugrad/vectors.htm.

[COSEPUP 1997] Committee on Science, Engineering, and Public Policy. *Adviser, Teacher, Role Model, Friend: On Being a Mentor to Students in Science and Engineering*. National Academy Press - http://www.nap.edu/openbook.php?record_id=5789.

[Countryman 1992] Joan Countryman. *Writing to Learn Mathematics: Strategies that Work*. Heinemann.

[CRA 2000a] Jan Cuny and William Aspray. *CRA Workshop: Recruitment and Retention of Women Graduate Students in Computer Science and Engineering*. http://www.cra.org/uploads/documents/resources/workforce_history_reports/rrwomen.pdf.

[CRA 2000b] William Aspray and Andrew Bernat. *CRA Workshop - Recruitment and Retention of Underrepresented Minority Graduate Students in Computer Science, March 4-5, 2000*. http://www.cra.org/uploads/documents/resources/workforce_history_reports/rrminorities.pdf.

[CRA 2006a] Holly Lord and J. McGrath Cohoon. *Recruiting and Retaining Women Graduate Students in Computer Science and Engineering*. http://www.cra.org/uploads/documents/resources/workforce_history_reports/gradrr07.pdf.

[CRA 2006b] *Graduate Recruitment & Retention in CSE*. http://people.virginia.edu/~jlc6j/gradrr/.

[CRA 2007] *CRA Taulbee Survey*. http://www.cra.org/resources/taulbee/.

[CSTB 1999] CSTB Committee on Information Technology Literacy. *Being Fluent with Information Technology*. National Academy Press.

[Cuny 20009a] Jan Cuny. *Computational Thinking will Require Transforming High School Computer Science: CS / 10,000 Project*. http://un-gaid.ning.com/profiles/blogs/computational-thinking-will.

[Cuny 2009b] Jan Cuny. *Transforming High School Computing: A Compelling Need, A National Effort*. http://opas.ous.edu/Committees/Resources/Publications/NSF_AP_CS_10000ExecSumm_Ed.pdf.

[Damasio 2003] Antonio Damasio. *Looking for Spinoza: Joy, Sorrow, and the Feeling Brain*. Harcourt, Inc.

[Darwin 1839] Charles Darwin. *The Voyage of the Beagle*. http://darwin-online.org.uk/EditorialIntroductions/Freeman_JournalofResearches.html

[Davidson & Goldberg 2009] Cathy N. Davidson and David Theo Goldberg. *The Future of Learning Institutions in a Digital Age*. The John D. and Catherine T. MacArthur Foundation Reports on Digital Media and Learning, MIT Press. http://mitpress.mit.edu/catalog/item/default.asp?ttype=2&tid=11841

[Denning 2007] Peter Denning. *Great Principles of Computing website*. http://cs.gmu.edu/cne/pjd/GP/GP-site/welcome.html.

[Denning 2009a] Peter Denning. "Computing: The Fourth Great Domain of Science", in *Communications of the ACM (CACM),* 52: 9, pp.27-29.

[Denning 2009b] Peter Denning. "Beyond Computational Thinking", in *Communications of the ACM (CACM),* 52: 6, pp. 28-30.

[Devlin 1998] Keith Devlin. *The Language of Mathematics: Making the Invisible Visible*. W. H Freeman & Co.

[Dodds et al. 2008] Zachary Dodds, Ran Libeskind-Hadas, Christine Alvarado, and Geoff Kuenning. "Evaluating a Breadth-first CS 1 for Scientists" in *Proceedings of ACM SIGCSE '08.*

[Donovan & Bransford 2005] M. Suzanne Donovan and John D. Bransford, editors. *How Students Learn: Mathematics in the Classroom*. National Academic Press.

[Downey & Stein 2006] Allen B. Downey and Lynn Andrea Stein. "Designing a small-footprint curriculum in computer science" in *Proceedings of ASEE/IEEE Frontiers in Education Conference*, October 2006.

[Engel 2009] Susan Engel. "What is Good College Teaching" in *Educause Forum Futures 2009 - Forum for the Future of Higher Education*. net.educause.edu/ir/library/pdf/ff0914s.pdf

[Forte & Guzdial 2005] Andrea Forte and Mark Guzdial. "Motivation and Non-Majors in CS1: Identifying Discrete Audiences for Introductory Computer Science" in *IEEE Transactions on Education*. 48: 2, pp. 248-253.

[Frenkel 2009] Karen A. Frenkel. "Liskov's creative joy" in *Communications of the ACM (CACM)* 52: 7, pp. 20-22.

[Friedman 2005] Thomas L. Friedman. *The World Is Flat: A Brief History of the Twenty-first Century*. Farrar, Straus and Giroux.

[Furst & DeMillo 2006] Merrick L. Furst and Richard DeMillo. *Creating Symphonic-Thinking Computer Science Graduates for an Increasingly Competitive Global Environment*. Technical Report. http://www.cc.gatech.edu/sites/default/files/Threads%20Whitepaper.pdf

[Furst et al. 2007] Merrick L. Furst, Charles L. Isbell, and Mark Guzdial. "Threads™: How to restructure a computer science curriculum for a flat world" in *Proceedings of ACM SIGCSE 2007*, pp 420-424.

[Gardner 1983] Howard Gardner. *Frames of Mind: The Theory of Multiple Intelligences*. Basic Books.

[Gibbs & Tucker 1986] Norman E. Gibbs and Allen B. Tucker. "A Model Curriculum for a Liberal Arts Degree in Computer Science" in *Communications of the ACM (CACM),* 29: 3, pp. 202-210.

[Goldin et al. 2006] Dina Goldin, Scott A. Smolka, and Peter Wegner, editors. *Interactive Computation: The New Paradigm*. Springer.

[Grossman & Minow 2001] Lawrence K. Grossman and Newton N. Minow. *A Digital Gift to the Nation: Fulfilling the Promise of the Digital and Internet Age*. Century Foundation Press.

[Gurwitz 1998] Chaya Gurwitz. "The Internet as a Motivating Theme in a Math/Computer Core Course for Nonmajors" in *Proceedings of ACM SIGCSE 1998*.

[Guzdial & Forte 2005] Mark Guzdial and Andrea Forte. "Design Process for a Non-majors Computing Course" in *Proceedings of ACM SIGCSE 2005*.

[Guzdial 2009] Mark Guzdial. "Teaching Computing to Everyone" in *Communications of the ACM (CACM),* 52: 5, p. 31.

[Hagel et al. 2010] John Hagel III, John Seely Brown, and Lang Davison. *The Power of Pull: How Small Moves, Smartly Made, Can Set Big Things in Motion*. Basic Books.

[Hartmanis & Lin 1992] Juris Hartmanis and Herbert Lin, editors. *Computing the Future: A Broader Agenda for Computer Science and Engineering*. National Academies Press, http://www.nap.edu/catalog.php?record_id=1982 .

[Harvey Mudd 2007] Engineering Department. *Engineering Clinic Handbook*. http://www.eng.hmc.edu/EngWebsite/Clinic/07-08ClinicHandbook.pdf

[Hey et al. 2009] Tony Hey, Stewart Tansley, and Kristin Tolle, editors. *The Fourth Paradigm: Data-Intensive Scientific Discovery*. Microsoft Research (October 16, 2009).

[Hodges 2000] Andrew Hodges. *Alan Turing: The Enigma*. Walker & Company.

[Hyde 2007] Lewis Hyde. The Gift: Creativity and the Artist in the Modern World, 25th Anniversary Edition. Vintage Books.

[Isbell, Stein, et al. 2009] Charles L. Isbell and Lynn Andrea Stein, plus Robb Cutler, Jeffrey Forbes, Linda Fraser, John Impagliazzo, Viera Proulx, Steve Russ, Richard Thomas, and Yan Xu. "(Re)Defining Computing Curricula by (Re)Defining Computing" in *ACM SIGCSE Bulletin*, 41: 4, pp. 195-207.

[Klein 1990] Julie Thompson Klein. *Interdisciplinarity: History, Theory, & Practice*. Wayne State University Press.

[Lewis et al. 2010] Clayton Lewis, Michele H. Jackson, and William M. Waite. "Student and faculty attitudes and beliefs about computer science" in *Communications of the ACM (CACM),* 53: 5, pp.78-85.

[Ma 1999] Liping Ma. Knowing and Teaching Elementary Mathematics: Teachers' Understanding of Fundamental Mathematics in China and the United States (Studies in Mathematical Thinking and Learning Series). Routledge.

[Markoff 2010] John Markoff. "In a Video Game, Tackling the Complexities of Protein Folding" in *NY Times,* August 4, 2010, http://www.nytimes.com/2010/08/05/science/05protein.html.

[Martin 2009] Roger L. Martin. *The Oposable Mind: Winning Through Integrative Thinking*. Harvard Business Press. http://www.rotman.utoronto.ca/integrativethinking/definition.htm

[Mitchell et al. 2003] William J. Mitchell, Alan S. Inouye, and Marjory S. Blumenthal, editors. *Beyond Productivity: Information Technology, Innovation, and Creativity*. National Academies Press.

[Morrison et al. 1982] Philip Morrison, Phyllis Morrison, and the Office of Charles and Ray Eames. *Powers of Ten: A Book About the Relative Size of Things in the Universe and the Effect of Adding Another Zero*. W.H. Freeman & Company, http://www.powersof10.com/.

[Nickel 2009] Mark Nickel. "Rodrigo Fonseca Assistant Professor of Computer Science" in *Today at Brown,* September 9, 2009, http://today.brown.edu/faculty/2009/fonseca.

[Nisbett 2003] Richard E. Nisbett. *The Geography of Thought: How Asians and Westerners Think Differently...and Why*. Free Press.

[Nisbett 2009] Richard E. Nisbett. *Intelligence and How to Get It: Why Schools and Cultures Count*. W.W.Norton & Company.

[NRC 1987] National Research Council Committee on Computer-Assisted Modeling. *Computer-Assisted Modeling: Contributions of Computational Approaches to Elucidating Macromolecular Structure and Function*. National Academy Press.

[NRC 2003] National Research Council. *BIO 2010: Transforming Undergraduate Education for Future Research Biologists*. National Academies Press.

[NRC 2004] National Research Council. *Computer Science: Reflections on the Field; Reflections from the Field*. National Academies Press.

[NRC 2009] National Research Council and National Academy of Engineering, Linda Katehi, Greg Pearson, and Michael Feder, Editors. *Engineering in K-12 Education: Understanding the Status and Improving the Prospects*. National Academies Press.

[NRC 2010] National Research Council Committee for the Workshops on Computational Thinking. *Report of a Workshop on The Scope and Nature of Computational Thinking, National Academies Press*. http://www.nap.edu/catalog/12840.html.

[Olin 2002] Olin College Curriculum Decision Making Board. *Once Upon a College*. http://www.olin.edu/academics/olin_history/cdmb_report.html.

[Palfrey & Gasser 2008] John Palfrey and Urs Gasser. *Born Digital: Understanding the First Generation of Digital Natives*. Basic Books.

[PCAST 1997] Presidents Committee of Advisors on Science and Technology, Panel on Educational Technology. *Report to the President on the Use of Technology to Strengthen K-12 Education in the United States*. http://www.whitehouse.gov/sites/default/files/microsites/ostp/pcast-nov2007k12.pdf

[Petzold 2008] Charles Petzold. *The Annotated Turing: A Guided Tour through Alan Turing's Historic Paper on Computability and the Turing Machine*. Wiley.

[Rich et al. 2005] Lauren Rich, Heather Perry, and Mark Guzdial. "A CS1 Course Designed to Address Interests of Women" in *Proceedings ACM SIGCSE 2005*. pp. 190-194.

[Rico 2000] Gabriele Rico. Writing the Natural Way. Jeremy P. Tarcher/Putnam, 2000.

[Sahami et al. 2010] Mehran Sahami, Alex Aiken, and Julie Zelenski. "Expanding the frontiers of computer science: designing a curriculum to reflect a diverse field" in *Proceedings of ACM SIGCSE 2010*.

[Shaw 2009] CACM Staff. "A conversation with David E. Shaw" in *Communications of the ACM (CACM*), 52: 10, pp. 48-54.

[Simpson 1998]Rosemary Michelle Simpson. *Principles of Rich Indexing.* http://www.cs.brown.edu/~rms/IndexingPrinciples.html

[Smith 1996] Brian Cantwell Smith. *On the Origin of Objects.* MIT Press.

[Snow 1959] Charles P. Snow. *The Two Cultures and the Scientific Revolution.* Cambridge University Press.

[Stein 1998] Lynn Andrea Stein. "What we've Swept under the Rug: Radically Rethinking CS1" in *Computer Science Education* 8: 2, pp. 118-129.

[Stein 2001] Lynn Andrea Stein. *Reconceptualizing Computation: Radically Rethinking CS1.* http://faculty.olin.edu/~las/2001/07/www.ai.mit.edu/people/las/papers/cs101-proposal.html.

[Stein 2006a] Lynn Andrea Stein. "A Small Footprint Curriculum for Computing: (And Why on Earth Anyone Would Want Such a Thing)" in *J. of Comput. Small Coll.* 21: 6, pp. 3-3.

[Stein 2006b] Lynn Andrea Stein. "Interaction, Computation, and Education" in [Goldin et al. 2006] Dina Goldin, Scott A. Smolka, and Peter Wegner, editors. *Interactive Computation: The New Paradigm*. Springer.

[Stevens 1974] Peter S. Stevens. *Patterns in Nature.* Atlantic Monthly Press - Little, Brown and Company.

[Taraban & Blanton 2008] Roman Taraban and Richard L. Blanton, editors. *Creating Effective Undergraduate Research Programs in Science: The Transformation from Student to Scientist.* Columbia University Teachers College Press.

[Thomas & Brown 2009] Douglas Thomas and John Seely Brown. *Learning for a World of Constant Change: Homo Sapiens, Homo Faber & Homo Ludens Revisited*. 7th Glion Colloquium, University of Southern California, June 2009. http://www.johnseelybrown.com/Learning%20for%20a%20World%20of%20Constant%20Change.pdf

[Tobias 1990] Sheila Tobias. *They're Not Dumb, They're Different.* Research Corporation.

[Tufte 2006] Edward Tufte. *Beautiful Evidence.* Graphics Press LLC

[van Dam 2003] Andries van Dam. "Grand Challenge 3. Provide a Teacher for Every Learner" in *Grand Research Challenges in Information Systems*. Anita Jones and William Wulf, editors, pp. 17-22, Computing Research Association. URL: http://archive.cra.org/reports/gc.systems.pdf

[Wing 2006] Jeannette Wing. "Computational Thinking" in *Communications of the ACM (CACM),* 49: 3, March 2006, pp. 33-35.

[Wing 2008] Jeannette Wing. "Computational thinking and thinking about computing" in *Philosophical Transactions of The Royal Society*, 366: 1881, pp. 3717-3725.

[Yarosh & Guzdial 2008] Svetlana Yarosh and Mark Guzdial. "Narrating Data Structures: The Role of Context in CS2" in *Proceedings of the ACM Third International Workshop on Computing Education Research*, pp. 87-98.

[Zyda 2009] Michael Zyda. "Computer Science in the Conceptual Age" in *Communications of the ACM (CACM),* 52: 12, pp. 67-72, http://gamepipe.usc.edu/USC_GamePipe_Laboratory/Home.html.

URLs

This set of URLs is alphabetically organized by the main topic. See the index for additional pointers to these URLS, e.g., bibliography; courses and programs; agencies, institutions, and centers; and projects. All URLs were accessed on Monday, 9 August 2010.

Bibliography

[ACM CSTA 2006] http://www.csta.acm.org/Curriculum/sub/CurrFiles/K-12ModelCurr2ndEd.pdf.

[ACM 2010] http://www.acm.org/education/curricula-recommendations.

[ACM/IEEE 2005] http://www.acm.org/education/education/curric_vols/CC2005-March06Final.pdf.

[ACM/IEEE 2008] http://www.acm.org/education/curricula/ComputerScience2008.pdf.

[AHC 2005] http://www.knaw.nl/publicaties/pdf/20051064.pdf

[Boonstra et al. 2004] http://www.iono.noa.gr/hellinomnimon/Books%20and%20Papers/historical%20information%20science.pdf

[Boyd 2008] http://www.danah.org/papers/TakenOutOfContext.pdf.

[Breck et al. 2008] http://www.cs.cornell.edu/home/kleinber/aaaiss08-courses.pdf.

[Brown 2008] http://www.johnseelybrown.com/.

[Catmull 2008] http://corporatelearning.hbsp.org/corporate/assets/content/Pixararticle.pdf.

[Cornell 2010] http://www.cs.cornell.edu/ugrad/CSMajorTransition08-09.htm; http://www.cs.cornell.edu/ugrad/vectors.htm.

[COSEPUP 1997] http://www.nap.edu/openbook.php?record_id=5789.

[CRA 2000a] http://www.cra.org/uploads/documents/resources/workforce_history_reports/rrwomen.pdf.

[CRA 2000b] http://www.cra.org/uploads/documents/resources/workforce_history_reports/rrminorities.pdf.

[CRA 2006a] http://www.cra.org/uploads/documents/resources/workforce_history_reports/gradrr07.pdf.

[CRA 2006b] http://people.virginia.edu/~jlc6j/gradrr/.

[CRA 2007] http://www.cra.org/resources/taulbee/.

[Cuny 20009a] http://un-gaid.ning.com/profiles/blogs/computational-thinking-will.

[Cuny 2009b] http://opas.ous.edu/Committees/Resources/Publications/NSF_AP_CS_10000ExecSumm_Ed.pdf.

[Darwin 1839] http://darwin-online.org.uk/EditorialIntroductions/Freeman_JournalofResearches.html.

[Davidson & Goldberg 2009] http://mitpress.mit.edu/catalog/item/default.asp?ttype=2&tid=11841

[Denning 2007] http://cs.gmu.edu/cne/pjd/GP/GP-site/welcome.html.

[Engel 2009] net.educause.edu/ir/library/pdf/ff0914s.pdf

[Furst & DeMillo 2006] http://www.cc.gatech.edu/sites/default/files/Threads%20Whitepaper.pdf.

[Hartmanis & Lin 1992] http://www.nap.edu/catalog.php?record_id=1982 .

[Harvey Mudd 2007] http://www.eng.hmc.edu/EngWebsite/Clinic/07-08ClinicHandbook.pdf.

[Markoff 2010] http://www.nytimes.com/2010/08/05/science/05protein.html.

[Martin 2009] http://www.rotman.utoronto.ca/integrativethinking/definition.htm

[Morrison et al. 1982] http://www.powersof10.com/.

[Nickel 2009] http://today.brown.edu/faculty/2009/fonseca.

[NRC 2010] http://www.nap.edu/catalog/12840.html.

[Olin 2002] http://www.olin.edu/academics/olin_history/cdmb_report.html.

[PCAST 1997] http://www.whitehouse.gov/sites/default/files/microsites/ostp/pcast-nov2007k12.pdf.

[Simpson 1998] http://www.cs.brown.edu/~rms/IndexingPrinciples.html.

[Stein 2001] http://faculty.olin.edu/~las/2001/07/www.ai.mit.edu/people/las/papers/cs101-proposal.html.

[Thomas & Brown 2009] http://www.johnseelybrown.com/Learning%20for%20a%20World%20of%20Constant%20Change.pdf.

[van Dam 2003] http://archive.cra.org/reports/gc.systems.pdf.

[Zyda 2009] http://gamepipe.usc.edu/USC_GamePipe_Laboratory/Home.html.

Courses and programs

ASU (Arizona State University)

http://engineering.asu.edu/undergraduate/cse

Augsburg College

Computational philosophy major - http://www.augsburg.edu/cs/degree_requirements.html

Brown University

CS931 - http://www.cs.brown.edu/courses/csci0931/

CS020 - http://www.cs.brown.edu/courses/csci0020.html

Mentoring - http://www.cs.brown.edu/~jfh/working/working.htm

CS040 - http://www.cs.brown.edu/courses/csci0040/

CS053: The Matrix in Computer Science - http://www.cs.brown.edu/courses/cs053/

CS015 - http://www.cs.brown.edu/courses/csci0150/

CS016 - http://www.cs.brown.edu/courses/csci0160/

CS017 - http://www.cs.brown.edu/courses/csci0170/

CS018 - http://www.cs.brown.edu/courses/csci0180/

CS019 - http://www.cs.brown.edu/courses/cs019/2009/

CS931: Introduction to Computation for the Humanities and Social Sciences - http://www.cs.brown.edu/courses/csci0931/

CS 237 - Interdisciplinary Scientific Visualization http://www.cs.brown.edu/courses/csci2370/

http://www.cs.brown.edu/ugrad/concentrations/cs-econ_scb-reqs.html

CMU

CS15-251 - Great Theoretical Ideas in Computer Science - http://www.cs.cmu.edu/~15251/

http://www.cmu.edu/interdisciplinary/programs/bcsaprogram.html

http://www.cs.cmu.edu/~tcortina/15-105sp09/courseinfo.html

http://www.csd.cs.cmu.edu/education/bscs/currreq.html

Cornell

Vectors -

http://www.cs.cornell.edu/degreeprogs/ugrad/CSMajor/Vectors/index.htm

http://www.cs.cornell.edu/ugrad/vectors.htm

Introductory AI-centered courses - http://www.cs.cornell.edu/home/llee/papers/teachai.home.html

CS Major Changes -

http://www.cs.cornell.edu/ugrad/CSMajorTransition08-09.htm
http://www.cs.cornell.edu/ugrad/InfoMtgPresent_12-2008.pdf
http://www.cs.cornell.edu/ugrad/TownHallPresent_1-2009.pdf

Georgia Tech

CS1315 Introduction to Media Computation - http://coweb.cc.gatech.edu/cs1315
Computational Media -
http://lcc.gatech.edu/compumedia/
http://www.cc.gatech.edu/future/undergraduates/bscm
Computing for Good - http://www.cc.gatech.edu/about/advancing/c4g
Threads - http://www.cc.gatech.edu/future/undergraduates/bscs/threads/
Computer Science Minor - http://www.cc.gatech.edu/future/undergraduates/csminor

Great Principles of Computing

http://cs.gmu.edu/cne/pjd/GP/gp_overview.html
http://cs.gmu.edu/cne/pjd/GP/GP-site/welcome.html

Harvard

Introduction to Computer Science - http://www.cs50.net/

Harvey Mudd

CS Minor - http://www.cs.hmc.edu/program/cs-minor
CS and Mathematics Joint Major - http://www.cs.hmc.edu/program/csmath-major
Engineering Clinic Handbook - http://www.eng.hmc.edu/EngWebsite/Clinic/07-08ClinicHandbook.pdf
Engineering Clinic - http://www.eng.hmc.edu/EngWebsite/index.php?page=Clinic.php
Off Campus Major -
http://www.hmc.edu/academicsclinicresearch/majors/offcampusmajor.html
Research, Academics, and Clinichttp://www.hmc.edu/academicsclinicresearch1.html
CS for Scientists course - http://www.hmc.edu/newsandevents/Grants%20Fall09.html

MIT

6.01 - Introduction to EECS
http://mit.edu/6.01/mercurial/fall09/www/index.html
http://mit.edu/6.01/mercurial/spring10/www/index.html
6.00 - Introduction to Computer Science and Programming
http://web.mit.edu/6.00/www/info.shtml
EECS New Curriculum
http://www.eecs.mit.edu/ug/newcurriculum/index.html
http://www.eecs.mit.edu/ug/newcurriculum/SBCS_6-3.html
http://www.eecs.mit.edu/ug/newcurriculum/ugcur-newsletter06.html

Olin

Curriculum
http://www.olin.edu/academics/curriculum.aspx

http://www.olin.edu/academics/olin_history/cdmb_report.html

http://www.olin.edu/academics/curriculum_facts.aspx

http://www.olin.edu/academics/docs/Yr1-4%20website%20diagram%2008-09.pdf

PGSS (Pennsylvania Governor's School for the Sciences)

PGSS (Pennsylvania Governor's School for the Sciences). Wikipedia description and Danah Boyd's eulogy,
http://en.wikipedia.org/wiki/Pennsylvania_Governor%27s_School_for_the_Sciences.

http://www.zephoria.org/thoughts/archives/2009/04/20/rip_pennsylvani.html.

Princeton

The Computational Universe - http://www.cs.princeton.edu/courses/archive/spring08/cos116/

Purdue

CS4EDU Project. http://cs4edu.cs.purdue.edu

SECANT Project. http://secant.cs.purdue.edu/

Research training course.
http://www.cs.purdue.edu/academic_programs/graduate/curriculum/doctoral.sxhtml#Research

EPICS Project. https://engineering.purdue.edu/EPICS/About

Rice

Computational Logic Course website http://www.cs.rice.edu/~vardi/comp409/

Stanford

Tracks. http://csmajor.stanford.edu/Tracks.shtml

Symbolic Systems. http://symsys.stanford.edu/

Stony Brook

Digitial Arts Minor - http://www.art.sunysb.edu/undergrad.html

Towson

Everyday Computational Thinking -
http://triton.towson.edu/users/dierbach/Pages/Courses/Honors%20ECT/HONR223.htm

UC Berkeley

GamesCrafters -

http://gamescrafters.berkeley.edu/

http://www.eecs.berkeley.edu/Courses/Data/485.html

Internet of Everyday Things -

http://www-inst.eecs.berkeley.edu/~cs194-5/sp08/

http://www.eecs.berkeley.edu/~culler/eecs194/

Mechatronics Design Lab -

http://www-inst.eecs.berkeley.edu/~ee192/sp09/

http://www-inst.eecs.berkeley.edu/~ee192/sp10/

Signals and Systems - http://www-inst.eecs.berkeley.edu/~ee120/fa08/

Software Engineering - http://www-inst.eecs.berkeley.edu/~cs169/sp10/doku.php?id=info

UC Davis

Electrical and Computer Engineering - http://www.ece.ucdavis.edu/undergrad/programdescription.html

UIUC

Computational Science and Engineering - http://www.cse.uiuc.edu/

Union/Lafayette

Union-Lafayette NSF CPATH grant. Creating a Campus-wide Computation Initiative. http://cs.union.edu/~barrv/Grants/computational-science.html.

USC

BS in Computer Games -

http://www.cs.usc.edu/academics/undergrad.html

http://www.cs.usc.edu/brochures/ugcsgm.pdf

Utah

CS 3200 - Introduction to Scientific Computing. http://www.eng.utah.edu/~cs3200/

Engineering Scholars Program - http://www.coe.utah.edu/current-undergrad/esp.php

Access Program for Women in Science and Mathematics - http://www.science.utah.edu/access.html

Virginia

Masters Degree in Digital Humanities - http://www.iath.virginia.edu/hcs/MDST.MA.html

Gender Diversity Workshops - http://people.virginia.edu/~jlc6j/gradrr/

Virginia Tech

Tracks - http://www.cs.vt.edu/undergraduate/tracks

Washington

BeneFIT: Fluency with Information Technology (FIT) http://courses.washington.edu/benefit/FIT100/

Capstone: Technology for Low-Income Regions - http://www.cs.washington.edu/education/courses/477/08sp/

Agencies, Institutions, Centers

[ABET] http://www.abet.org/

[CMU Center for Computational Thinking. 2010] http://www.cs.cmu.edu/~CompThink/.

[CRA] http://www.cra.org

Projects

CRA-W Projects - http://www.cra-w.org/projects

Digital Promise Project 2010 - http://www.digitalpromise.org/

Strand Maps

[NSDL 2007] NSDL. AAAS Science Literacy Maps - Atlas of Science Literacy Science Literacy Maps. http://strandmaps.nsdl.org/ Strand Map Service - http://strandmaps.nsdl.org/cms1-2/docs/index.jsp

[SECANT 2010] SECANT: Science Education in Computational Thinking Project, Purdue University, http://secant.cs.purdue.edu/.

Exemplar Programs
Great Principles of Computing

Extract from the Great Principles Overview webpage -

http://cs.gmu.edu/cne/pjd/GP/gp_overview.html

Peter Denning

George Mason University

What is Computation?

What is computation? Information? What can we know through computing? What can we not know through computing?

Computer scientists have studied these questions since the 1940s. But today, people in all fields of science, engineering, business, and even politics are asking the same questions.

The tradition that computing is the science of phenomena surrounding computers is being superseded. Computing is -- in fact, always has been -- the science of information processes. Starting around 1995, many scientists began announcing they had discovered natural information processes in their fields. Their discoveries initiated a new tradition: computing is a science of the natural as well as the artificial.

In its older tradition, computing was most naturally described by the ideas in its core technologies -- such as programming, graphics, networks, or supercomputing. However, the new tradition calls for a description in terms of fundamental principles. A principles-oriented descriptive framework reveals computing's deep structures and how they apply in many fields. The framework reveals common aspects of technologies and creates opportunities for innovation. It opens entirely new ways to stimulate the excitement and curiosity of young people about the world of computing.

In the 1940s, computation was seen as a tool for solving equations, cracking codes, analyzing data, and managing business processes. By the 1980s, computation had advanced to become a new method in science, joining the traditional theory and experiment. During the 1990s, computation advanced even further as people in many fields discovered they were dealing with information processes buried in their deep structures -- for example, quantum waves in physics, DNA in biology, brain patterns in cognitive science, information flows in economic systems. Computation has entered everyday life with new ways to solve problems, new forms of art, music, motion pictures, and commerce, new approaches to learning, and even new slang expressions.

The fundamental questions of computing, listed at the start, have become important in many fields, which rely heavily on computation and computational methods to advance their work. In fact, studying which aspects of computing are most useful traditional science fields helps to identify fundamental computing principles.

Computational metaphors have entered the common language with everyday phrases like "I am programmed to react that way," and "My brain crashed and had to be rebooted." The University of Washington has developed "fluency in information technology", widely used in high schools and colleges to help students learn and apply basic computational principles[90]. Many people now speak of "computational thinking,"[91] referring to the use of computational principles in many fields and everyday life. Computation is everywhere.

---

[90] University of Washington course BENEFIT - Fluency with Information Technology. http://courses.washington.edu/benefit/FIT100/; [CSTB 1999] "Being Fluent with Information Technology",

[91] [Wing 2006] "Computational Thinking"

There is another advantage to a principles-framework: it's easier to learn than a technology framework. Describing the field in terms of technology ideas was a good approach in the early days when the core technologies were few. In 1989, ACM listed 9 core technologies. In 2001, however, ACM listed about three dozen. For the newcomer, learning the inner workings of 36 technologies and their 630 possible direct interactions presents a daunting challenge.

In the computing field, we have not yet developed an articulation of the field in terms of fundamental principles. Principles-based approaches are common in other fields. Computing has only recently reached a level of maturity where it can do the same.

Project Objectives

The Great Principles project aims to develop and maintain a principles articulation of computing. The benefits of doing this are:

- Expose the deep structure of the field. Doing so can reduce the apparent complexity of the field, contributing to greater understanding, better designs, and simpler, more reliable systems.
- Enable designers and users to see connections among technologies based on similar principles. This will facilitate sound designs, cross fertilization among technologies, new discoveries, and innovations.
- Establish a new relationship with people from other fields by offering computing principles in language that shows them how to map the principles into their own fields.
- Provide inspiring stories about the development of the field and its principles for young people.
- Develop new approaches to teaching computing that inspire curiosity and excitement.

A principles framework will complement the existing technology frameworks for understanding computing. We will discuss this further below in the section on using a principles oriented representation of a body of knowledge.

The project is ongoing because the body of principles needs to be constantly updated. Some principles once ascendant will be retired as they go out of use -- for example, construction of logic gates from discrete transistors. We will surely discover new principles that will enable the solutions of contemporary issues -- for example, problems with user interfaces, identity theft, network security, spam, information overload, dependable software, hastily formed networks, distance learning, and discovering terrorist plots. Thus a great principles framework is a living depiction of the field, always open to births and retirements.

Outline of the Framework

By a principle, we mean a statement that guides or constrains future action. Computing principles are of two kinds: (1) recurrences, including laws, processes, and methods that describe repeatable cause-effect relationships, and (2) guidelines for conduct. An example of a law is that the fastest sorting algorithms take time at least order of n log n to arrange n items in order. An example of a conduct guideline is that network programmers should divide protocol software into layers. The purpose of such principles is to reduce apparent complexity, increase understanding, and enable good design.

We analyzed many computing technologies to identify the principles on which they are based, and we studied how what aspects of computation are influencing other fields. From this, we concluded that computing principles can be grouped into seven categories:

Computation (meaning and limits of computation)

Communication (reliable data transmission)

Coordination (cooperation among networked entities)

Recollection (storage and retrieval of information)

Automation (meaning and limits of automation)

Evaluation (performance prediction and capacity planning)

Design (building reliable software systems)

These categories resulted from a functional analysis of many computing technologies and applications:

1. Computing systems are built from processing elements that process and store information (computation, recollection);
2. Processing elements exchange information (communication);
3. Processing elements cooperate toward common goals (coordination);
4. Humans delegate tasks to systems of processing elements (automation);
5. Humans predict the speed and capacity of systems (evaluation); and
6. Humans decompose systems into processing elements and organize their construction (design).

These categories are like windows into the one computing knowledge space rather than slices of the space into separate pieces. Each window sees the space in a distinctive way; but the same thing can be seen in more than one window. Internet protocols, for example, are sometimes seen as means for data communication, sometimes as means of coordination, and sometimes as a means for recollection.

We also found that most computing technologies draw principles from all seven categories. This finding confirmed our suspicion that a principles interpretation will help us see many common factors among technologies.

This set of categories satisfied our goal to have a framework with a manageable number of categories. While the list of computing technologies may continue to grow, the number of categories is likely to remain stable for a long time.

Exemplar Programs Core plus Tracks, Threads, and Vectors

CMU (Carnegie Mellon University)

Thoughts on Foundations, Tracks, and Refactoring Content From a CMU Perspective

Peter Lee

Carnegie Mellon University

Problem Statement

How does one foster a broad view of computer science while simultaneously ensuring technical depth and the ability to solve problems collaboratively?

Solutions/Applicability

CMU offers several computing-related programs at the undergraduate level. They have grown up organically rather than conceived through any overarching master plan. Hence, what is described here is just my personal view. However, much of this also reflects certain precepts of the institution.

- *The broad view of the field*: "Computer science is the study of all phenomena related to computers."
- *The need for depth*: "Every graduate should know what it takes to think deeply, and in a sustained manner."
- *Team problem-solving skills*: The ability to work in teams, to solve problems collaboratively, is a necessary skill for all scientists and engineers.

The CMU undergraduate program in computer science reflects the broad view of the field by attempting to encompass all computing-related areas with a primary focus on problem solving. Toward this end, the program is not owned by the CS Department. It operates at the School of Computer Science level, drawing faculty and grad student teaching support from all six academic units (CS, Machine Learning, Language Technologies, Human-Computer Interaction, Robotics, and Computation/Organization/Policy).

The program is unusual in being a full four-year CS curriculum, meaning that students enter CMU as CS majors from day one. This means that most students complete a relatively comprehensive CS core before the end of the sophomore year:

- Data structures and algorithms. The standard course.
- "Great ideas in computer science". An intensive, problem-solving course in computational thinking[92].
- Introduction to computer systems. Taught from the Bryant/O'Hallaron book, giving a software-oriented introduction to the low-level structure of computers
- Principles of programming. Taught in ML, designed to challenge conventional thinking in programming and analysis/proofs of programs

[92] CS15-251 - Great Theoretical Ideas in Computer Science - http://www.cs.cmu.edu/~15251/

Most courses, starting with the four-course core, require students to work in teams. This is intended to build up, through experience, team problem-solving skills.

At the completion of this core, there are very few hard requirements. Students must take one "Deep Thinking" course, drawn from a menu of courses that the faculty have deemed provide opportunities to develop deep, sustained thinking. Students must also take an advanced algorithms course, and at least one advanced course in systems, in foundations, and in applications.

Other than that, students are encouraged to explore. We provide "suggested options", which are simply small groupings of upper-level courses that constitute coherent directions in computer science. The currently listed options are:

- artificial intelligence
- cognitive modeling
- computational biology
- computer systems
- entrepreneurship
- graphics/VR
- HCI
- language technologies
- robotics
- scientific computation
- software systems
- theory

The course offerings are extensive, and in principle the fact that they are all included in a single overarching CS program means there are no barriers preventing students from exploring and including the full range of computing-related courses into their CS major.

A proliferation of computing-related majors puts pressure on the core, leading to consideration of tracks.

While the CS program expresses an approach to achieving breadth in concept and depth in ability, it is focused on a particular "paradigm" of computational thinking, namely the problem-solving paradigm. As the field has expanded, however, a need has arisen for truly distinct programs. A sample of these include:

- *Computational Biology*. Distinguished by laboratory science, observation of phenomena, the search for fundamental truths.
- *Human-Computer Interaction*. Distinguished by human-oriented exploration and experimentation, involving human subjects, industrial design, social science.
- *Computer Science and the Arts*. Distinguished by creative exploration, studio activity, exhibition/performance.
- *Robotics*. Distinguished by significant engineering and field experimentation.

While the rationale for these programs may be well justified, each chooses different parts of, and puts different stresses on, the four-course core.

Cornell University

CS Department curriculum reform: "Vectors"

Lillian Lee

Cornell University

The Cornell CS Department, which offers the (same) CS major to students in both the College of Arts and Sciences and the College of Engineering, is in the midst of a multi-year revision of our educational offerings. We began with a several-year build-up of an array of novel introductory non-required courses to attract students to computing --- one example is David Easley and Jon Kleinberg's "Networks" course, which regularly draws over 300 students from all 7 undergraduate colleges at Cornell, representing over 30 different majors. A description of some of these courses can be found in a AAAI symposium paper.[93]

Having worked very hard on introductory "attractor" courses, we have now turned our attention to education within the CS major.

Problem(s) being solved with the new curriculum

How can we balance "core" versus "specialization" in a way that can seamlessly preserve the "product" of the two while our field continues to quickly evolve? "Core" knowledge should be that which is fundamental for all CS undergraduates to know, but deciding what deserves to be "core" can be tricky, especially as the "cutting edge" moves forward at lightning pace.

"Specialization" is, by dint of dichotomy, not "core", but we still want to provide support for students to learn about rich and mature or maturing sub-areas that have developed either within CS or in interactions with other fields. This is no minor consideration, given that students often have very little inkling of how broad, deep, and enabling computer science is.

Unfortunately, framing the debate explicitly as one of "what is core, versus what isn't" can lead to standoffs based on (perfectly reasonable) philosophical differences.

What is the solution

We instead framed our central question as, what are some "*vectors*"(so called to suggest "directions of study"), that we want a CS education to enable our students to be able to pursue, and what knowledge supports these vectors singly or as a whole?

The term "vector" is meant to be evocative. Vectors have a direction (intellectual coherence) and a magnitude (coursework requirements), and need not be even close to orthogonal, but rather can have high inner product (overlap). We thus did not take a "top-down" approach of trying to divide CS up into a relatively few distinct sub-fields, but rather, a "bottom-up" approach, where we can create new vectors on the fly as the field evolves. Thus, we have vectors for traditional fields ("graphics"), newly-emerging fields ("network science"), fields that are quite close to each other ("AI" and "human-language technologies", or "systems" and "security and trustworthy computing"), and collections of knowledge or practice such as "software engineering/code warrior" and a breadth-emphasizing "Renaissance/basis vector" (think "Renaissance person"). (In the latter two cases, one can also think of the "vector" notion as representing the normal to a "cross-cutting plane".) The full set of 12 vectors we currently offer is:

- Renaissance (Basis)
- Artificial Intelligence
- Computational Science and Engineering
- Data-Intensive Computing
- Graphics

[93] http://www.cs.cornell.edu/home/llee/papers/teachai.home.html

- Human-Language Technologies
- Network Science
- Programming Languages
- Security and Trustworthy Systems
- Software Engineering / Code Warrior
- Systems
- Theory

(Again, recall that these represent vectors we can currently reliably offer, not a conception of a partition of CS. We considered other vectors during our study, as well, and we expect to add more pending future faculty hires.)

Our subsequent analysis of what knowledge supports the vast majority of the vectors --- and thus can be deemed "core" --- triggered the following major changes. (Many other changes increasing flexibility were made, too; see the URLs below[94] for more info). We added a requirement that students take a full rigorous upper-level course in probability (we specified a menu of courses across the university, and allow the course to double-count towards other requirements, easing the burden of the addition). This meant that room was made in our early required discrete-math course, since much of the probability material was moved out. We removed the requirement of an automata/computability course. The finite-state-machines material was re-located to the "probability hole" in the required discrete math course. The undecidability material was re-located to the required algorithms course. We also removed the scientific computing requirement.

This makes our core courses beyond the two intro programming courses as follows: discrete math, probability, functional programming + data structures, algorithms, computer architecture, and systems. Furthermore, we added the requirement of completion of at least one vector, and increased flexibility in myriad technical ways. So, in terms of number of semester-long courses required by the major (ignoring requirements of the student's College, which could be either Arts & Sciences or Engineering), we have:

Students must take the following courses

(A) 2 intro CS/programming courses (many get AP credit for the 1st)

(B) 5 core CS courses

(C) 3 CS electives

(D) 5 CS practicum

(E) 3 technical electives (e.g., math, chemistry, psychology)

(F) 3 upper level external specialization courses (coherent program in something other than CS)

(G) 1 free elective (the Colleges have additional free elective requirements)

The courses the students take to fulfill the above or their College requirements must also include a probability course and must fulfill at least one vector. Thus, the probability and vector requirement are implemented as "predicates" on a student's transcript: so that courses can be counted both towards a vector and towards another requirement; this is part of the reason why it is easy for students to complete more than one vector, and for us to allow different vectors to require different numbers of courses.

[94] http://www.cs.cornell.edu/ugrad/InfoMtgPresent_12-2008.pdf
http://www.cs.cornell.edu/ugrad/TownHallPresent_1-2009.pdf
http://www.cs.cornell.edu/ugrad/CSMajorTransition08-09.htm

Success - what can we say so far, either quantitatively or qualitatively?

The new rules were only instituted in Spring 2009, so it is too early to measure impact on enrollments, placement, etc. However, here are some preliminary data of interest.

The CS classes of '10, '11, and '12 could choose between the "old major" and the "vectors major" because they entered Cornell when the "old major" was in place. Having completed most of the old-major requirements, 30% of class of '10 went with the old major; but 70% of '11 chose the vectors one, and while most of class of '12 hasn't affiliated yet, 100% of our currently very small sample have chosen vectors.

Moreover, the students' vector selections across these three classes are surprisingly evenly distributed (except that the Renaissance/basis vector is very popular, as can be expected because we encourage all students to complete it whether or not they choose other vectors as well). Thus, we believe that we are identifying and satisfying the diverse interests of our student body.

In what contexts/situations is this solution applicable

We believe the vector-driven analysis led to much more productive discussions than simply asking, "should course X be required?" We also think a "bottom-up" approach, in which a new vector can be added any time a subfield arises, makes the approach very adaptable to future changes, avoids sometimes irresolvable debates about what is "most important", and allows for smaller fields to be brought to students' attention.

Georgia Institute of Technology

College of Computing - Threads

Jim Foley

Georgia Institute of Technology

"The underlying goal of Threads™ is to increase the value of an undergraduate computer science degree from The College of Computing at Georgia Tech – to produce graduates who will be in high demand and who will continuously contribute value throughout successful careers. An aim is to produce graduates who have a broad set of skills and are not easily outsourceable. Facing the challenges of an increasingly global economy and competitive information technology workforce, the rapid convergence of technologies … Threads™… empowers students with the directions, tools, and opportunities needed to figure out what kind of computationalists they want to become. Threads™ aims to attract a diverse undergraduate population and produce lifetime-learning graduates tuned to the future, globally competitive economy."[95]

Problem

Declining enrollments / interest, believed to be in part caused by:

- Concern that traditional CS jobs might be outsourced[96] – what we were producing might not match future job opportunities.
- CS seen as uninteresting – not clear what it is good for – why study it? Students may not see enough of how computing relates to the real world.

Companies might in future not want to hire our grads.

Computing has become too broad for an undergrad to know something about each aspect of computing (just as has happened to engineering over the past 150 years).

Solution

'Contextualize' computing – make it more purpose/goal oriented. Not just bunch of CS courses without knowing why they are being studied. Contextualizing has been shown to increase learning *and* ability to transfer that learning to other contexts.

Do this by developing a set of threads[97] – a collection of required and elective courses, from CS and other disciplines, such that any two threads, along with university general education requirements, constitutes a BSCS degree and satisfies ABET accreditation requirements. GT has 8 threads and thus 28 combinations of 2 threads that lead to a degree. The threads are: Modeling and Simulation; Devices; Theory; Information Internetworks; Intelligence; Media: People; Platforms.[98]

All 8 threads are built on a set of core courses totaling 23 semester credit hours.[99]

Applicability

The threads approach can be applied by any computing school with sufficient courses to define at least three threads[100], which thus provides students with three paths to a degree (4 threads => 6 paths, 5 threads => 10 paths, etc).

According to Mark Guzdial, the greatest challenges of Threads are administrative:

---

[95] [Furst & DeMillo 2006] "Creating Symphonic-Thinking Computer Science Graduates for an Increasingly Competitive Global Environment"

[96] See [Friedman 2005] "The World Is Flat: A Brief History of the Twenty-first Century"

[97] [Furst et al. 2007] "Threads™: how to restructure a computer science curriculum for a flat world"

[98] Threads BSCS web site: http://www.cc.gatech.edu/future/undergraduates/bscs/threads/

[99] http://www.cc.gatech.edu/future/undergraduates/bscs/corereq

[100] See the Southern Polytechnic State University NSF CPATH project: http://cse.spsu.edu/jwang/research/NSF-CPATH/main.html

- How do you advise students when they are choosing among C(8,2) = 28 degree options?
- How do you predict what classes you should offer when you don't know students' combination/thread intentions? This is a variant of the pre-req problem, but it's forward planning rather than resolving courses backwards.
- How do you deal with students having a variety of different Thread interests and perhaps different Foundation courses (e.g., an introduction to computer science focusing on robots vs. media, the Gameboy programming in in our computer architecture course for media students vs. the Yale Patt processor simulator for learning low-level issues) in later classes? In your OS class, for example, how do you deal with students' prior knowledge when it can be so radically different?

MIT (Massachusetts Institute of Technology)

Department of Electrical Engineering and Computer Science: New Undergraduate Curriculum

— Members of the EECS Curriculum Innovation Committee, Tomás Lozano-Pérez, chairman

http://www.eecs.mit.edu/ug/newcurriculum/ugcur-newsletter06.html- text of the webpage

The department is undertaking its first major curriculum revision in a dozen years. A key aim of this revision is to take significant advantage of our joint EECS department. The intersections between EE and CS, as technical disciplines, are deep and varied. One visible point of contact is in computer architecture and digital design; but there are also important contacts between artificial intelligence and estimation and control; between computer networking and information theory and coding; between numerical methods and computational biology; between hearing and speech and natural language; between computer vision and speech and signal processing. Our goal is to have students experience EECS, not just EE and CS. To that end, we will immerse them early in an integrated experience, exposing them to the breadth and richness of the field.

The field of EECS is so broad, however, that no student can be grounded in everything EECS has to offer. Traditionally, we have focused on a small set of "core" topics that all students are required to study.

Because there is a huge range of important, elementary material, any particular subset will necessarily omit many fascinating and fundamental topics. Instead, our new approach is to insist that students study a broad set of fundamentals, but not that every student study exactly the same set. We believe that a combination of the integrated introductory experience and early exposure to a broader choice of subjects will help students appreciate the range of possible intellectual and career opportunities in EECS.

While breadth is important, it is also crucial for students to attain mastery in some area. This gives satisfaction and a sense of achievement, as well as the confidence and ability to go on to master new areas. In this curriculum, we will ask undergraduates to choose two specialization areas to study in depth, and to build a curriculum of foundational subjects that support study in those areas. This experience will serve well students who do not continue on for the Masters degree; and it will provide a depth of knowledge that will enhance and deepen the MEng experience for those who do.

The current proposed SB requirements can be described in terms of a 4-level classification of subjects:

- The Introductory subjects are fully integrated introductions to EECS that introduce the big ideas of EECS in an applied context. All students will take the same two introductory subjects. In addition students are required to take two Mathematics subjects beyond the Institute-required calculus subjects.
- The Foundation subjects are intended to lay the technical foundations for study in EECS. Students will select three of these subjects from a list that currently includes: Circuits and Electronics, Signals and Systems, Computation Structures, Software Design, Analysis and Design of Algorithms.
- The Concentration subjects begin an in-depth exploration of the major areas of EECS. Students will select three subjects from a list that currently includes seven subjects; initially this list is very similar to our existing Header[101] subjects. In addition, the students must select one laboratory subject from a list that includes approximately 10 subjects.
- Advanced subjects are undergraduate subjects that build on the concentration subjects and go

[101] An MIT Header subject is similar to a Stanford gateway, e.g., an introduction to a particular field or specialization or track.

deeper into some area. Students will take two advanced subjects. One of our key objectives is to ensure that students explore some of the concentrations in department

The intent is that each level of subject has, in general, prerequisites from the previous level, with the introductory level having prerequisites in the general Institute requirements. Our expectation is that this new curriculum will balance breadth and depth, give students a range of foundational knowledge, and provide mastery in some subareas of EECS.

Stanford University

CS Department - Track Models

Pat Hanrahan
Stanford University

Problem

The redesign of the Stanford program was motivated by several concerns:

- The curriculum was dated and badly in need of a thorough update
- The growing every-increasing importance of computing throughout the university and the rapid increase in interdisciplinary aspect of these programs
- The desire to market computer science to students with diverse interests. Potential majors were electing to major in programs like symbolic systems and management sciences instead of computer science.

Solution

Stanford has changed the curriculum to a core and tracks model[102]. The core is what every computer science major would take. Each person selects a track[103]. The core contains 6 courses. Each course is one quarter long. There are two subsequences: theory and systems.

The theory core consists of three courses:

Theory 1: Mathematical foundations of Computing

Theory 2: Introduction to Probability for Computer Scientists

Theory 3: Data structures and algorithms

The systems core consists of:

Systems 1: Programming Abstractions

Systems 2: Computer Organization

Systems 3: Principles of Computer Systems

The number of courses required by all students has decreased. Some topics that were required but are no longer required include logic, automata theory, and artificial intelligence. These courses have been moved into the theoretical computer science and artificial intelligence track. Previously, we required a separate probability course taught in another department, but will now be taught by CS faculty. Finally, the systems track is a condensed version of a series of operating systems courses.

The initial set of tracks includes: artificial intelligence, theoretical computer science, tracks, computer systems, human-computer interaction, graphics, information sciences, bio-computation, and traditional computer science.

Tracks typically consist of 4 required courses.

Success

The new curriculum was deployed in the fall of 2008. Faculty and students have enthusiastically endorsed it, but it is too early to judge its success. Most of the core courses are being taught by the first time, and we expect it will take several years to optimize the courses.

---

[102] [Sahami et al. 2010] "Expanding the frontiers of computer science: designing a curriculum to reflect a diverse field"

[103] http://csmajor.stanford.edu/Tracks.shtml

Exemplar Programs
Design under Constraints

MIT (Massachusetts Institute of Technology)

EECS 6.01 Introduction to EECS I

http://mit.edu/6.01/mercurial/spring10/www/index.html

"6.01 explores fundamental ideas in electrical engineering and computer science, in the context of working with mobile robots. Key engineering principles, such as abstraction and modularity, are applied in the design of computer programs, electronic circuits, discrete-time controllers, and noisy and/or uncertain systems."

UC Berkeley

EECS 120: Signals and Systems

http://inst.eecs.berkeley.edu/~ee120/fa08/

"One of the key abilities of an engineer is system-level thinking. Taking EECS 120 will help you develop this skill. In particular, you will see how the math and physics you have learned in other courses help you understand rather complex systems that occur in engineering and computer science (with applications to communication systems, biomedical imaging, control, and robotics). The knowledge and skills that you will acquire in EECS 120 are at the heart of an entire series of senior-level and graduate classes, including 121, 123, 125, 128, 192, 221A, 224, and 226A. EECS 126 (Probability and Random Processes) is not required for this course and gives a complementary set of tools needed for advanced material, especially in the areas of communications and signal processing. We assume that you have familiarity with lower division physics and circuits since these are the source of many examples."

CS169 Software Engineering

http://www-inst.eecs.berkeley.edu/~cs169/sp10/doku.php?id=info

"Building large software systems is hard, but experience shows that building large software systems that actually work is even harder. And trying to do all this before your competitors has proved fatal to many software projects. This course covers techniques for dealing with the complexity of software systems. We will focus on the technology of software engineering for the individual and small team, rather than business or management issues. Topics will include, among others, specifications, principles of design and software architecture, testing, debugging, static analysis, and version control. You are expected to actively participate in the classes."

EE192 Mechatronics Design Lab

http://inst.eecs.berkeley.edu/~ee192/sp09/

"The Mechatronics Design Lab is a design project course focusing on application of theoretical principles in electrical engineering and computer science to control of mechatronic systems incorporating sensors, actuators and intelligence. This course gives you a chance to use your knowledge of (or learn about) power electronics, filtering and signal processing, control, electromechanics, microcontrollers, and real-time embedded software in designing a racing robot."

EECS194 – Internet of Everyday Things

http://www.eecs.berkeley.edu/~culler/eecs194/

"Everyday we deal with a myriad of sophisticated devices that have sensors, controllable actions, and intelligence that transforms inputs and intention into action. These devices are the appliances in the kitchen, the gadgets in living room, the lighting, heating, cooling, watering, draining facilities in the building, the array of thermometers, scales, and health meters, the many forms of recreational

equipment, and so on. They are stand-alone and fixed function. The intelligence is sometimes digital, often analog, and almost inevitably human."

GamesCrafters

http://gamescrafters.berkeley.edu/

"At the core is a group project called GAMESMAN, an open-source AI architecture developed for solving, playing, and analyzing two-person abstract strategy games (e.g., Tic-Tac-Toe or Chess). Given the description of a game as input, our system generates a command-line interface and graphical application that will solve it (in the strong sense), and then play it perfectly. Programmers can easily prototype a new game with multiple rule variants, learn the strategy via color-coded value moves (win = go = green, tie = caution = yellow, lose = stop = red), and perform extended analysis."

University of Washington

CSE477 Capstone Design - Technology for Low-Income Regions

http://www.cs.washington.edu/education/courses/477/08sp/

"The theme for the course is 'Technology for Low-Income Regions'. In the fall quarter's seminar we used a set of readings to familiarize ourselves with several interesting problem domains. By the end of the quarter, we had determined some possible project ideas. The past quarter, we developed and refined those ideas into detailed implementation plans for the spring quarter (the CSE477 CompE capstone design course). This quarter it is time to turn these ideas into working prototypes."

Prototypes and Example Integrated Joint Majors

Computers in the Arts and Digital Media

Jim Foley

Georgia Institute of Technology

"The Bachelor of Science in Computational Media (BSCM) was developed in recognition of computing's significant role in communication and expression, and is a joint offering between the College of Computing and the School of Literature, Communication, and Culture within the Ivan Allen College of Liberal Arts. The two schools offer CM majors a thorough education in understanding the computer as an expressive medium, and in creating interactive media not only with commercial software packages but also by writing code. … CM majors work as interns in the design and development of video games, animation and special effects, and user interfaces. Most CM alumni have gone on to work for major video-game studios and interactive-media firms. Some are now pursuing graduate degrees in digital media, human-computer interaction, and even film studies.

"The BSCM curriculum gives students a grasp of the computer as a medium: the technical, the historical-critical, and the applied. Students gain significant hands-on and theoretical knowledge of computing, as well as an understanding of visual design and the history of media. Our graduates are uniquely positioned to plan, create, and critique new digital media forms for entertainment, education and business." [104]

Problem

- Declining enrollments / interest
- Concern that traditional CS jobs might be outsourced[105] – what we were producing might not match future job opportunities
- CS seen as uninteresting – not clear what it is good for – why study it?

Solution

'Contextualize' computing – make it more purpose/goal oriented. Not just bunch of CS courses without knowing why they are being studied. Contextualizing has been shown to increase learning *and* ability to transfer that learning to other contexts.

Do this by defining a degree that includes computing courses and new media courses. The latter set of courses provides the context (goal, use) for the computing courses. Computing is an ends to a means, not a means in and of itself.

Applicability

Interdisciplinary degrees such as the BSCM can be developed given the existence of two schools such as CoC and LCC, appropriate courses, and faculty and administrative champions in both schools.

[104] Computational Media web site: http://lcc.gatech.edu/compumedia/

[105] See [Friedman 2005] "The World Is Flat: A Brief History of the Twenty-first Century"

Computational biology

David Shaw
D. E. Shaw Research

Departmental involvement:

Computer science

Biology

Possibly one or more faculty members from, for example, chemistry, physics, applied mathematics and/or statistics who have related research interests

Distinctive subject matter (required and/or elective)

Bioinformatics and systems biology

- Exposure to simple applications of statistical analysis and data mining
- Use of large databases containing genomic, proteomic, and other biological data

Biomolecular modeling and simulation

- Focus on molecular structures and dynamics
- Use of iterative numerical algorithms
- Applications of basic computational physics (mechanics, electrostatics, etc.)

Visualization of biological data

- Graphical representation of statistical data and network models
- Abstraction, rendering, and animation of three-dimensional molecular structures

Sample Course Outline

Overview

This document is intended to provide an informal sketch of a hypothetical survey course on (a) various *applications* of computer science within the biological sciences, and (b) the *understanding* of biological systems from the perspective of a computer scientist. I've chosen to focus on biology only because my own research involves the design of algorithms and machine architectures for biochemical applications. Biology should thus be regarded only as an *example* domain, which could just as easily be replaced by any of a number of other domains.

One of the goals of the course described below would be to convey to students who have had little exposure to the field of computer science a feeling for the power of computational methods and technologies within non-obvious application domains. Of equal importance, the course is intended to develop—by way of example rather than description—some feeling for the conceptual framework sometimes referred to as "computational thinking."

Intended Audience

The canonical student for whom the course was designed might be a college sophomore who has not declared a major in computer science, but who might become interested in the field if exposed to intellectually compelling applications and ideas that fall outside the province generally associated with the stereotypical computer nerd.

A different version of the course could be designed that might be more appropriate for junior- and senior-level students who have already gained some familiarity with various topics and tools within the fields of computer science, biology, chemistry, mathematics, or any of several engineering disciplines. Analogous

courses could also be developed for students majoring in other fields within not only the physical and social sciences, but the arts and humanities as well.

Prerequisites

The prerequisites for a version of the course designed for sophomores might include nothing more than a one-semester introduction to computer programming in any commonly used high-level language that might be available at the institution for use in completing homework assignments or class projects. (Such programming would typically involve straightforward computational experimentation with very simple mathematical models, though many such exercises would in any case involve the use of domain-specific applets written by the instructor or curriculum designer.)

A somewhat more advanced version of the course might target students who have already completed one-semester freshman- or sophomore-level courses in biology, chemistry and physics, but who would not be assumed to have any additional background (beyond the above introductory programming course) in computer science or mathematics.

Course Outline

Living things are computing devices

DNA is a programming language

The genome is a computer program

Application: Reconstructing the human genome from DNA fragments

Genes describe proteins

Proteins attach themselves to small molecules and to other proteins

The molecular jigsaw puzzle

Lots of missing pieces

No picture on the box

What do proteins do?

Some proteins are building blocks

Usually lots of them, all stuck together

Hair and fingernails

Application: Cellulosic biofuels (making energy out of corn husks)

Some proteins serve as plumbing

Pipes, pumps and faucets

Nerves and brains

Some proteins carry messages

Proteins talk to each other

Application: Systems biology (who's talking with whom?)

Application: Computers in evolutionary biology (who's your daddy?)

Some proteins form machines

Rotors, sliders, and bead-stringers

Application: Nanotechnology (building molecular machines)

How proteins work

- What they look like
- *Application area: Building a molecular microscope*
- How they get that way
- *Application area: The protein folding problem*
- How they fit together
- How they move
- Basically, we still don't know
- *Application area: Molecular dynamics simulations in science and medicine*

How drugs work

- Drugs are small molecules (usually)
- Drugs attach themselves to specific proteins (usually)
- *Application: "Rational" drug design (molecular matchmaking)*
- Drugs gum up the machinery (sometimes)
- *Application area: Molecular dynamics simulations in science and medicine*
- Discovering drugs is very hard and very expensive
- Combinatorial chemistry
- If you don't know the answer, try everything
- Side effects: A case of mistaken identity
- Gene therapy: Designing the perfect impostor

Stranger than fiction

- Computing with biological molecules
- DNA-based computers
- Biomimetic materials
- *Application: Biomimetic materials (walking on water and flying like a bird)*
- Computer-designed organisms
- Quantum chemistry: The weirdness under the hood

Computer Engineering

Randy Katz
UC Berkeley

From Wikipedia

"Computer Engineering is a discipline that combines elements of both Electrical Engineering and Computer Science. Computer engineers usually have training in electrical engineering, software design and hardware-software integration instead of only software engineering or electrical engineering. Computer engineers are involved in many aspects of computing, from the design of individual microprocessors, personal computers, and supercomputers, to circuit design. Usual tasks involving computer engineers include writing software and firmware for embedded microcontrollers, designing VLSI chips, designing analog sensors, designing mixed signal circuit boards, and designing system and control/monitoring software."

Observations

- A mature interdisciplinary area between CS and EE; it's been around for decades
- Popular among undergraduates
- Students with this background have traditionally been in high demand by employers
- Also a foundation for more advanced work at the graduate level
- Deep background in signal processing, devices, circuits, digital logic, computer architecture, system and real-time software
- Integrated curriculum in the sense of a coherent collection of required courses and sequences from CS and EE disciplines
- Integrative "capstone" design courses, e.g., Hardware/Software Co-design or Design Studio Courses

What Made Computer Engineering a Successful Joint Program:

- Industry demand
- Relatively close intellectual proximity of computer science and electrical engineering

Examples

Bachelor of Science in Engineering in Computer Systems Engineering, Arizona State University

http://engineering.asu.edu/undergraduate/cse

The Computer Systems Engineering program is concerned with the analysis, design and evaluation of computer systems, both hardware and software. The program emphasizes computer organization and architecture, systems programming, operating systems and digital hardware design.

Computer engineers often find themselves focusing on problems or challenges that result in new "state of the art" products that integrate computer capabilities. They work on the design, planning, development, testing and even the supervision of manufacturing of computer hardware -- including everything from chips to device controllers. They also focus on computer networks for the transmission of data and multimedia. They work on the interface between different pieces of hardware and strive to provide new capabilities to existing and new systems or products.

The work of a computer engineer is grounded in the hardware -- from circuits to architecture -- but also focuses on operating systems and software. Computer engineers must understand logic design, microprocessor system design, computer architecture and computer interfacing, while continually focusing on system requirements and design.

Students are required to complete 120 credit hours to earn a B.S.E. in Computer Systems Engineering. In addition to general studies courses, students complete math, science, introduction to engineering, circuits and computer science foundation courses. Upper division courses include software engineering, computing ethics, data structures and algorithms, computer architecture, computer networks, digital hardware, embedded micro systems and operating systems. Students can also choose from a number of 400-level electives, including special topics courses. As a culminating experience in the program, students take a two semester sequence of Capstone courses, where they work on a real-world project provided by industry in a team setting. Students can choose to concentrate their studies in information assurance.

Electrical and Computer Engineering Curriculum, UC Davis
http://www.ece.ucdavis.edu/undergrad/programdescription.html

The program in Computer Engineering provides the student with a broad and well-integrated background in the concepts and methodologies that are needed for the analysis, design, development, organization, theory, programming, and application of information processing systems. Although such systems are popularly called "computers," they involve a far wider range of disciplines than merely computation, and the Computer Engineering curriculum is correspondingly broad. The programs present the essential material in electronic circuits, digital logic, discrete mathematics, computer programming, data structures, and other topics.

The Computer Engineering curriculum prepares students for careers in computer engineering or graduate studies by providing a solid background in mathematics, physical sciences, and the traditional computer engineering subjects: electronics, computer hardware, and computer software. Here electronics refers to the five Electrical Engineering specialty areas (1) physical electronics, (2) electromagnetics, (3) analog electronics, (4) digital electronics, and (5) communications, control, and signal processing. The upper division units required in electronics, computer hardware and computer software consist of 13 units in electronics courses, 18 units in computer hardware courses, and 12 units in computer software courses. The remaining units consist of 11 units of design electives and 9 units of technical electives. By carefully selecting these 20 design and technical electives, students can focus on electronics, computer hardware, or computer software, or distribute these units among the three areas. Students who complete the Computer Engineering curriculum will receive a Bachelor of Science in Computer Engineering.

Computational finance and financial engineering

David Shaw
D.E. Shaw Research

Departmental involvement:

Computer science

Economics and/or business school finance department

Possibly one or more faculty members from, for example, applied mathematics, statistics or operations research who have related research interests

Distinctive subject matter (required and/or elective):

Underlying theory

- Tobin's separability theorem and Markowitz mean-variance optimization
- Diversification theory, Sharpe's Capital Asset Pricing Model, and factor models
- The Black-Scholes options model and path-dependent options

Methods

- Algorithms for the static and dynamic construction of optimal portfolios
- Techniques for prediction based on financial time series data
- Analytical and numerical methods for option valuation

Applications

- Risk management (within private and public sectors)
- Managing institutional assets (e.g., pension funds, university endowments)
- Economic policy, systemic risk, long-term investments in "public goods"

Computational Methods in the Humanities and Social Sciences

Randy Katz
UC Berkeley

Definition from Wikipedia

"Digital humanities is a field of study, research, teaching, and invention concerned with the intersection of computing and the disciplines of the humanities. It is methodological by nature and interdisciplinary in scope. It involves investigation, analysis, synthesis and presentation of knowledge using computational media. It studies how these media affect the disciplines in which they are used, and what these disciplines have to contribute to our knowledge of computing. Academic departments of the digital humanities typically include technical practitioners as well as traditionally trained scholars with experience or expertise in digital media. Such departments tend to be heavily involved in collaborative research projects with colleagues in other departments. The interdisciplinary position of the digital humanities is comparable to that of comparative literature in relation to literary studies. It involves experts in both research and teaching; in all of the traditional arts and humanities disciplines (history, philosophy, linguistics, literature, art, archaeology, and music of many cultures, for example); specialists in electronic publication and computational analysis, in project design and visualization, in data archiving and retrieval."

Observations

- Digital Humanities: Information management, with elements of digital media creation and management
- New approaches for on-line publication and dissemination of materials, including such things as virtual environments for exploring historical times and places (e.g., UVa project on Shenandoah Valley during the Civil War)
- Library science for the 21st Century? Emerging MS programs
- Simulation/Counter-factual history, e.g., fivethirtyeight.com

These areas are on the leading edge of research in the humanities and social sciences, are mostly realized as new graduate programs, and have not as yet filtered down to the undergraduate level. There is not much evidence of joint programs involving computer scientists and humanities undergraduates (there are some double majors, such as CS + Cog Sci, and alternative programs in "Schools of Information" that are focused on these areas).

Examples

Master's Degree in Digital Humanities, Media Studies Program, University of Virginia
http://www.iath.virginia.edu/hcs/MDST.MA.html

"making humanities content tractable to computational methods"

"Successful completion of this MA program requires students to have, or to acquire, a working familiarity with major computer operating systems (PC, Macintosh, Unix) and software more specialized than the usual office applications (e.g., visual programming software, multimedia authoring tools, databases), as well as with markup languages (e.g., SGML, XML) and programming languages (e.g., Perl, Java)."

Computational Science and Engineering

Randy Katz
UC Berkeley

Definition from Wikipedia

"Computational science is the field of study concerned with constructing mathematical models and numerical solution techniques and using computers to analyze and solve scientific, social scientific and engineering problems. In practical use, it is typically the application of computer simulation and other forms of computation to problems in various scientific disciplines. The field is distinct from computer science (the mathematical study of computation, computers and information processing). It is also different from theory and experiment, which are the traditional forms of science and engineering. The scientific computing approach is to gain understanding, mainly through the analysis of mathematical models implemented on computers. Scientists and engineers develop computer programs, application software, that model systems being studied and run these programs with various sets of input parameters."

Observations

- A mature field: the earliest uses of computer were to solve scientific and engineering applications
- Requires knowledge of applied mathematics, a science or engineering domain, and computer science
- Such broad and deep knowledge is difficult to obtain in a four year undergraduate program, so these are more typically graduate level joint programs
- Traditionally have been more attractive to scientists and engineers that seek knowledge of computational methods for their research than for computer scientists who seek application of their computing technology
- Mostly domain-specific courses (e.g., "computational materials science") with some integrative courses (e.g., "applications of parallel computers")

What Makes Computational Science and Engineering A Less Successful Joint Program?

- Application-domain students learn some computing e.g., "E77-Introduction to Computer Programming for Scientists and Engineers"[106]; the computer science students don't study much science (or engineering) or mathematics beyond the freshman-sophomore level; modest undergraduate interest
- Computer science students prefer more advanced mathematics classes when they are taught be computer scientists, e.g., discrete math courses taught within CS. Generally, interest is numerical analysis has waned, and these have not been replaced with more modern courses on mathematics for computation.

Expected Outcomes of E77-Introduction to Computer Programming for Scientists and Engineers

Ability to apply knowledge of mathematics, science, and engineering to answer subject matter questions appropriate to first year studies using computing techniques. Ability to design and conduct computational experiments, as well as to analyze and interpret data generated from numerical computations. Ability to design a computational system (program), component (subroutine, object), or process (code fragment) to meet desired needs within realistic constraints such as realizability within a

[106] http://www.me.berkeley.edu/ABET/2005/courses/E77web.shtml

fixed computational environment. Ability to identify, formulate, and solve basic level engineering problems using modern computational techniques. Ability to communicate technical methods and results effectively. Recognize the need for, and an ability to engage in life-long learning about computing. Ability to use the techniques, skills, and modern engineering computation for engineering practice.

Topics

Course Introduction; MATLAB Basics. MATLAB Arrays, Vectors, Matrices. Control Structures. Functions and writing MATLAB. Data Structures and Classes. Systems of Linear Equations. Least-Squares. Approximation by polynomials. Internal representation of numbers. Numerical Root. Numerical Integration. Numerical Differentiation. Numerical Solution of ODEs. Linear Recursion and Tree Recursion. Sorting and Searching.

Contrast with a first course in computing for computer science students

This first course concentrates mostly on the idea of abstraction, allowing the programmer to think in terms appropriate to the problem rather than in low-level operations dictated by the computer hardware. ... We are interested in teaching you about programming, not about any particular programming language. We consider a series of techniques for controlling program complexity, such as functional programming, data abstraction, object-oriented programming, and query systems.

Examples

Computational Science and Engineering, University of Illinois

http://www.cse.uiuc.edu/

CSE Graduate Option Programs in Participating Departments

CSE is an interdisciplinary graduate option program with many participating departments whose individual CSE Options are broadly similar but may vary in some details. Each department has determined a set of specific requirements that students must satisfy to complete the CSE Option for a given degree program in that particular department. The departmental program descriptions specify required and recommended courses as well as any relevant examination or thesis requirements. Upon satisfying the degree requirements of the students' graduate department and the department's CSE Option requirements, the student is awarded a CSE Certificate signifying successful completion of the CSE Option.

- Aerospace Engineering
- Agricultural and Biological Engineering
- Astronomy
- Atmospheric Sciences
- Bioengineering
- Biophysics and Computational Biology
- Chemical and Biomolecular Engineering
- Chemistry
- Civil and Environmental Engineering
- Computer Science
- Electrical and Computer Engineering
- Industrial and Enterprise Systems Engineering
- Materials Science and Engineering
- Mathematics
- Mechanical Science and Engineering
- Nuclear, Plasma and Radiological Engineering

- Physics

Premedical computer science programs

John Guttag
MIT

Departmental involvement

Computer science

Biology

Chemistry

Distinctive subject matter (required and/or elective)

Requirements for admission to medical school

Physics

Biology

Chemistry/bio-chemistry

Algorithmic thinking

Computational aids for decision making

Probability and statistics

Machine learning

Visualization and analysis of continuous and categorical data

Comments

I have spent the last 30+ years teaching at MIT, so my views are heavily shaped by my interactions with an atypical group of undergraduates. Similarly, my views on medical school education are shaped primarily by interactions with Harvard Medical School.

- Many undergraduates who profess an interest in medicine are unsure whether their long term interest lies in treating patients or contributing to medicine by developing technology or doing research. These students often choose "pre-med" majors to keep their options open.
- Typical pre-med course requirements are similar in character to much of the medical school curriculum in their emphasis on the mastery of facts. In contrast, many engineering programs (including computer science) place more of an emphasis on problem solving.
- Premier medical schools understand that the skills acquired in studying engineering are extremely valuable, in part because they expect many of the their graduates to advance as well as practice medicine.
- HMS requires roughly the following of their applicants:
    - Biology: 2 terms
    - Chemistry/bio-chemistry: 4 terms
    - Physics: 2 terms
    - Mathematics: 2 terms of calculus (don't ask me why, since they don't appear to make any use of it)

Even for a student with no AP credits, this only totals 10 terms. At MIT, it is only 4 terms beyond the general requirements. At all universities engineering majors have an overlap with these requirements.

- A joint major between computer science and any of biology, chemistry, chemical engineering, bio-chemistry, or bio-engineering would likely satisfy the admissions requirements of most medical

schools.

- A student would be well-advised to choose such a joint major. If the student elects to go on to medical school, the computer science training will make them a better physician and better prepared for many kinds of medical research. If the student decides on a different career path (even medical research), the CS background will be a great asset.

Introduction to Index Use

Just as the Web is a hypertextual structure that encourages both directed search and exploratory browsing strategies, so this index is a paper hypertext. The See and See Also trails provide alternative perspectives, while the substructure of the main entries pull together the different contexts in which a term or phrase appears. These miniature structures provide alternative structures to the main structure of the report and help in clarifying complex concepts whose components are sometimes widely separated. [Simpson 1998] describes these principles in more detail.