# Computer Science Curricula 2013

Ironman Draft
(Version 1.0)

February 2013

The Joint Task Force on Computing Curricula
Association for Computing Machinery
IEEE-Computer Society

# CS2013 Steering Committee

**ACM Delegation**

Mehran Sahami, *Chair* (Stanford University)

Andrea Danyluk (Williams College)

Sally Fincher (University of Kent)

Kathleen Fisher (Tufts University)

Dan Grossman (University of Washington)

Beth Hawthorne (Union County College)

Randy Katz (UC Berkeley)

Rich LeBlanc (Seattle University)

Dave Reed (Creighton University)

**IEEE-CS Delegation**

Steve Roach, *Chair* (Univ. of Texas, El Paso)

Ernesto Cuadros-Vargas (Univ. Catolica San Pablo)

Ronald Dodge (US Military Academy)

Robert France (Colorado State University)

Amruth Kumar (Ramapo Coll. of New Jersey)

Brian Robinson (ABB Corporation)

Remzi Seker (Embry-Riddle Aeronautical Univ.)

Alfred Thompson (Microsoft, retired)

# Table of Contents

# Chapter 1: Introduction

ACM and IEEE-Computer Society have a long history of sponsoring efforts to establish international curricular guidelines for undergraduate programs in computing on roughly a ten-year cycle, starting with the publication of Curriculum 68 [1] over 40 years ago. This volume is the latest in this series of curricular guidelines. As the field of computing has grown and diversified, so too have the curricular recommendations, and there are now curricular volumes for Computer Engineering, Information Systems, Information Technology, and Software Engineering in addition to Computer Science [3]. These volumes are updated regularly with the aim of keeping computing curricula modern and relevant. The last complete Computer Science curricular volume was released in 2001 (CC2001) [2], and an interim review effort concluded in 2008 (CS2008) [4].

This volume, Computer Science Curricula 2013 (CS2013), represents a comprehensive revision. The CS2013 guidelines include a redefined body of knowledge, a result of rethinking the essentials necessary for a Computer Science curriculum. It also seeks to identify exemplars of actual courses and programs to provide concrete guidance on curricular structure and development in a variety of institutional contexts.

The development of curricular guidelines for Computer Science has always been challenging given the rapid evolution and expansion of the field. The growing diversity of topics potentially relevant to an education in Computer Science and the increasing integration of computing with other disciplines create particular challenges for this effort. Balancing topical growth with the need to keep recommendations realistic and implementable in the context of undergraduate education is particularly difficult. As a result, the CS2013 Steering Committee made considerable effort to engage the broader computer science education community in a dialog to better understand new opportunities and local needs, and to identify successful models of computing curricula – whether established or novel.

## Charter

The ACM and IEEE-Computer Society chartered the CS2013 effort with the following directive:

> *To review the Joint ACM and IEEE-CS Computer Science volume of Computing Curricula 2001 and the accompanying interim review CS 2008, and develop a revised and enhanced version for the year 2013 that will match the latest developments in the discipline and have lasting impact.*
>
> *The CS2013 task force will seek input from a diverse audience with the goal of broadening participation in computer science. The report will seek to be international in scope and offer curricular and pedagogical guidance applicable to a wide range of institutions. The process of producing the final report will include multiple opportunities for public consultation and scrutiny.*

The process by which the volume was produced followed directly from this charter.

## Overview of the CS2013 Process

The ACM and IEEE-Computer Society respectively appointed the Steering Committee co-chairs, who, in turn, recruited the other members of the Steering Committee in the latter half of 2010. This group received its charter and began work in Fall 2010, starting with a survey of Computer Science department chairs (described below). The Steering Committee met for the first time in February 2011, beginning work with a focus on revising the Body of Knowledge (BoK). This initial focus was chosen because both the CS2008 report and the results of the survey of department chairs pointed to a need for creation of new knowledge areas in the Body of Knowledge.

The Steering Committee met in person roughly every 6 months throughout the process of producing this volume and had conference call meetings at monthly intervals. Once the set of areas in the new Body of Knowledge was determined, a subcommittee was appointed to revise or create each Knowledge Area (KA). Each of these subcommittees was chaired by a member of the Steering Committee and included at least two additional Steering Committee members as well as other experts in the area chosen by the subcommittee chairs. As the subcommittees produced drafts of their Knowledge Areas, others in the community were asked to provide feedback, both through presentations at conferences and direct review requests. The Steering Committee also collected community input through an online review and comment process. The

KA subcommittee Chairs (as members of the CS2013 Steering Committee) worked to resolve conflicts, eliminate redundancies and appropriately categorize and cross-reference topics between the various KAs. Thus, the computer science community beyond the Steering Committee played a significant role in shaping the Body of Knowledge throughout the development of CS2013. This two-year process ultimately converged on the version of the Body of Knowledge presented here.

Beginning at its summer meeting in 2012, the Steering Committee turned much of its focus to course and curricular exemplars. In this effort, a broad community engagement was once again a key component of the process of collecting exemplars for inclusion in the volume. The results of these efforts are seen in Appendix C, which includes these exemplars.

## Survey Input

To lay the groundwork for CS2013, the Steering Committee conducted a survey of the use of the CC2001 and CS2008 volumes. The survey was sent to approximately 1500 Computer Science (and related discipline) department chairs and directors of undergraduate studies in the United States and an additional 2000 department chairs internationally. We received 201 responses, representing a wide range of institutions (self-identified):

- research-oriented universities (55%)
- teaching-oriented universities (17.5%)
- undergraduate-only colleges (22.5%)
- community colleges (5%)

The institutions also varied considerably in size, with the following distribution:

- less than 1,000 students (6.5%)
- 1,000 to 5,000 students (30%)
- 5,000 to 10,000 students (19%)
- more than 10,000 students (44.5%)

In response to questions about how they used the CC2001/CS2008 reports, survey respondents reported that the Body of Knowledge (i.e., the outline of topics that should appear in undergraduate Computer Science curricula) was the most used component of the reports. When questioned about new topical areas that should be added to the Body of Knowledge, survey respondents indicated a strong need to add the topics of *Security* as well as *Parallel and Distributed Computing*. Indeed, feedback during the CS2008 review had also indicated the importance of these two areas, but the CS2008 steering committee had felt that creating new KAs was beyond their purview and deferred the development of those areas to the next full curricular report. CS2013 includes these two new KAs (among others): *Information Assurance and Security*, and *Parallel and Distributed Computing*.

## High-level Themes

In developing CS2013, several high-level themes provided an overarching guide for the development of this volume. These themes, which embody and reflect the CS2013 Principles (described in detail in the next chapter of this volume) are:

- *The "Big Tent" view of CS*. As CS expands to include more cross-disciplinary work and new programs of the form "Computational Biology," "Computational Engineering," and "Computational X" are developed, it is important to embrace an outward-looking view that sees CS as a discipline actively seeking to work with and integrate into other disciplines.
- *Managing the size of the curriculum.* Although the field of computer science continues to rapidly expand, it is not feasible to proportionately expand the size of the curriculum. As a result, CS2013 seeks to re-evaluate the essential topics in computing to make room for new topics without requiring more total instructional hours than the CS2008 guidelines. At the same time, the circumscription of curriculum size promotes more flexible models for curricula without losing the essence of a rigorous CS education.
- *Actual course exemplars.* CS2001 took on the significant challenge of providing descriptions of six *curriculum models* and forty-seven possible *course descriptions* variously incorporating the knowledge units as defined in that report. While this effort was valiant, in retrospect such course guidance did not seem to have much impact on actual course design. CS2013 plans to take a different approach: to identify and describe existing successful courses and curricula to show how relevant knowledge units are addressed and incorporated in actual programs.
- *Institutional needs.* CS2013 aims to be applicable in a broad range of geographic and cultural contexts, understanding that curricula exist within specific institutional needs, goals, and resource constraints. As a result, CS2013 allows for explicit flexibility in curricular structure through a tiered set of core topics, where a small set of Core-Tier 1 topics are considered essential for all CS programs, but individual programs choose their coverage of Core-Tier 2 topics. This tiered structure is described in more detail in Chapter 4 of this report.

## Knowledge Areas

The CS2013 Body of Knowledge is organized into a set of 18 Knowledge Areas (KAs), corresponding to topical areas of study in computing. The Knowledge Areas are:

- AL - Algorithms and Complexity
- AR - Architecture and Organization
- CN - Computational Science
- DS - Discrete Structures
- GV - Graphics and Visual Computing
- HCI - Human-Computer Interaction
- IAS - Information Assurance and Security
- IM - Information Management
- IS - Intelligent Systems
- NC - Networking and Communications
- OS - Operating Systems
- PBD - Platform-based Development
- PD - Parallel and Distributed Computing
- PL - Programming Languages
- SDF - Software Development Fundamentals
- SE - Software Engineering
- SF - Systems Fundamentals
- SP - Social Issues and Professional Practice

Many of these Knowledge Areas are derived directly from CC2001/CS2008, but have been revised—in some cases quite significantly—in CS2013; other KAs are new to CS2013. Some represent new areas that have grown in significance since CC2001 and are now integral to studies in computing. For example, the increased importance of computer and network security in the past decade led to the development of IAS-Information Assurance and Security. Other new KAs represent a restructuring of knowledge units from CC2001/CS2008, reorganized in a way to make them more relevant to modern practices. For example, SDF-Software Development Fundamentals pulls together basic knowledge and skills related to software development, including knowledge units that were formerly spread across Programming Fundamentals, Software Engineering, Programming Languages, and Algorithms and Complexity. Similarly, SF-Systems Fundamentals brings together fundamental, cross-cutting systems concepts that can serve as a foundation for more advanced work in a number of areas.

It is important to recognize that Knowledge Areas are interconnected and that concepts in one KA may build upon or complement material from other KAs. The reader should take care in reading the Body of Knowledge as a whole, rather than focusing on any given Knowledge Area in isolation. Chapter 4 contains a more comprehensive overview of the KAs, including motivations for the new additions.

## Professional Practice

The education that undergraduates in Computer Science receive must adequately prepare them for the workforce in a more holistic way than simply conveying technical facts. Indeed, soft skills (such as teamwork, verbal and written communication, time management, problem solving, and flexibility) and personal attributes (such as risk tolerance, collegiality, patience, work ethic, identification of opportunity, sense of social responsibility, and appreciation for diversity) play a critical role in the workplace. Successfully applying technical knowledge in practice often requires an ability to tolerate ambiguity and to negotiate and work well with others from different backgrounds and disciplines. These overarching considerations are important for promoting successful professional practice in a variety of career paths.

Students will gain some soft skills and personal attributes through the general college experience (e.g., patience, time management, work ethic, and an appreciation for diversity), and others through specific curricula. CS2013 includes examples of ways in which an undergraduate Computer Science program encourages the development of soft skills and personal attributes. Core hours for teamwork and risk management are covered in the SE-Software Engineering Knowledge Area under Project Management. The ability to tolerate ambiguity is also core in SE-Software Engineering under Requirements Engineering. Written and verbal communications are also part of the core in the SP-Social Issues and Professional Practice Knowledge Area under Professional Communication. The inclusion of core hours in the SP-Social Issues and Professional Practice KA under the Social Context knowledge unit helps to promote a greater understanding of the implications of social responsibility among students. The importance of lifelong learning as well as professional development is described in the preamble of the SP-Social Issues and Professional Practice Knowledge Area as well as in both Chapter 2 (Principles) and Chapter 3 (Characteristics of Graduates).

## Exemplars of Curricula and Courses

The CS2013 Ironman draft includes examples of actual fielded courses—from a variety of universities and colleges—to illustrate how topics in the Knowledge Areas may be covered and combined in diverse ways. Importantly, we believe that the collection of such exemplar courses and curricula provides a tremendous opportunity for further community involvement in the development of the CS2013 volume. We invite members of the computing community to contribute courses and curricula from their own institutions (or other institutions that they may be familiar with). Those interested in potentially mapping courses/curricula to the CS2013 Body of Knowledge are encouraged to contact members of the CS2013 Steering Committee for more details.

## Timeline

The CS2013 curricular guidelines development effort is in its final year. This Ironman repoty version 1.0 , includes drafts of all of the components planned for the final report. A summary of the CS2013 timeline for the rest of the project is as follows:

| | |
|---|---|
| February 2013: | CS2013 Ironman report version 1.0 released |
| June 2013: | Comment period for Ironman draft closes |
| Fall 2013: | CS2013 Final report planned for release |

## Opportunities for Involvement

We believe it is essential for endeavors of this kind to engage the broad computing community to review and critique successive drafts. To this end, the development of this report has already benefited from the input of more than 100 contributors beyond the Steering Committee. We welcome further community engagement on this effort in multiple ways, including (but not limited to):

- Comments on the Ironman version 1.0 draft.
- Contribution of exemplar courses/curricula that are mapped against the Body of Knowledge.
- Descriptions of pedagogic approaches and instructional designs (both time-tested and novel) that address professional practice within undergraduate curricula.
- Sharing of institutional challenges, and solutions to them.

Comments on all aspects of this report are welcome and encouraged via the CS2013 website:

**`http://cs2013.org`**

## References


[1] ACM Curriculum Committee on Computer Science. 1968. Curriculum 68: Recommendations for Academic Programs in Computer Science. *Comm. ACM* 11, 3 (Mar. 1968), 151-197.

[2] ACM/IEEE-CS Joint Task Force on Computing Curricula. 2001. ACM/IEEE Computing Curricula 2001 Final Report. http://www.acm.org/sigcse/cc2001.

[3] ACM/IEEE-CS Joint Task Force for Computer Curricula 2005. Computing Curricula 2005: An Overview Report. http://www.acm.org/education/curric_vols/CC2005-March06Final.pdf

[4] ACM/IEEE-CS Joint Interim Review Task Force. 2008. Computer Science Curriculum 2008: An Interim Revision of CS 2001, Report from the Interim Review Task Force. http://www.acm.org/education/curricula/ComputerScience2008.pdf

## Acknowledgments

The CS2013 draft reports have benefited from the input of many individuals, including: Alex Aiken (Stanford University), Ross Anderson (Cambridge University), Florence Appel (Saint Xavier University), Helen Armstrong (Curtin university), Colin Armstrong (Curtin university), Krste Asanovic (UC Berkeley), Radu F. Babiceanu (University of Arkansas at Little Rock), Mike Barker (Massachusetts Institute of Technology), Michael Barker (Nara Institute of Science and Technology), Paul Beame (University of Washington), Robert Beck (VIllanova University), Matt Bishop (University of California, Davis), Alan Blackwell (Cambridge University), Don Blaheta (Longwood University), Olivier Bonaventure (Universite Catholique de Louvain), Roger Boyle (University of Leeds), Clay Breshears (Intel), Bo Brinkman (Miami University), David Broman (Linkoping University), Kim Bruce (Pomona College), Jonathan Buss (University of Waterloo), Netiva Caftori (Northeastern Illinois University, Chicago), Paul Cairns (University of York), Alison Clear (Christchurch Polytechnic Institute of Technology), Curt Clifton (Rose-Hulman and The Omni Group), Yvonne Cody (University of Victoria), Tony Cowling (University of Shefffield), Joyce Currie-Little (Towson University), Ron Cytron (Washington University in St. Louis), Melissa Dark (Purdue University), Janet Davis (Grinnell College), Marie DesJardins (University of Maryland, Baltimore County), Zachary Dodds (Harvey Mudd College), Paul Dourish (University of California, Irvine), Lynette Drevin (North-West Universit), Scot Drysdale (Dartmouth College), Kathi Fisler (Worcester Polytechnic Institute), Susan Fox (Macalester College), Edward Fox (Virginia Tech), Eric Freudenthal (University of Texas El Paso), Stephen Freund (Williams College), Lynn Futcher (Nelson Mandela Metropolitan University), Greg Gagne (Wesminister College), Dan Garcia (UC Berkeley), Judy Gersting (Indiana University-Purdue University Indianapolis), Yolanda Gil (University of Southern California), Michael Gleicher (University Wisconsin, Madison), Frances Grodzinsky (Sacred Heart University), Anshul Gupta (IBM), Mark Guzdial (Georgia Tech), Brian Hay (University of Alaska, Fairbanks), Brian Henderson-Sellers (University of Technology, Sydney), Matthew Hertz (Canisius College), Tom Hilburn (Embry-Riddle Aeronautical University), Tony Hosking (Purdue University), Johan Jeuring (Utrecht University), Yiming Ji (University of South Carolina Beaufort), Maggie Johnson (Google), Matt Jones (Swansea University), Frans Kaashoek (MIT), Lisa Kaczmarczyk (ACM Education Council), Jennifer Kay (Rowan

University), Scott Klemmer (Stanford University), Jim Kurose (University of Massachusetts, Amherst), Doug Lea (SUNY Oswego), Terry Linkletter (Central Washington University), David Lubke (NVIDIA), Bill Manaris (College of Charleston), Samuel Mann (Otago Polytechnic ), C. Diane Martin (George Washington University ), Andrew McGettrick (University of Strathclyde), Morgan Mcguire (Williams College), Keith Miller (University of Illinois at Springfield), Narayan Murthy (Pace University), Kara Nance (University of Alaska, Fairbanks), Todd Neller (Gettysburg College), Reece Newman (Sinclair Community College), Christine Nickell (Information Assurance Center for Computer Network Operations, CyberSecurity, and Information Assurance), James Noble (Victoria University of Wellington), Peter Norvig (Google), Joseph O'Rourke (Smith College), Jens Palsberg (UCLA), Robert Panoff (Shodor.org), Sushil Prasad (Georgia State University), Michael Quinn (Seattle University), Matt Ratto (University of Toronto), Penny Rheingans (U. Maryland Baltimore County), Carols Rieder (Lucerne University of Applied Sciences), Eric Roberts (Stanford University), Arny Rosenberg (Northeastern and Colorado State University), Ingrid Russell (University of Hartford), Dino Schweitzer (United States Air Force Academy), Michael  Scott (University of Rochester), Robert Sedgewick (Princeton University), Helen Sharp (Open Univeristy), the SIGCHI executive committee, Robert Sloan (University of Illinois, Chicago), Ann Sobel (Miami University), Carol Spradling (Northwest Missouri State University), Michelle Strout (Colorado State University), Alan Sussman (University of Maryland, College Park), Blair Taylor (Towson University), Simon Thompson (University of Kent), Gerrit van der Veer (Open University Netherlands), Johan Vanniekerk (Nelson Mandela Metropolitan University), Christoph von Praun (Georg-Simon-Ohm Hochschule Nürnberg), Rossouw Von Solms (Nelson Mandela Metropolitan University), John Wawrzynek (UC Berkeley), Charles Weems (Univ. of Massachusettes, Amherst), David Wetherall (University of Washington), and Michael Wrinn (Intel).

Additionally, review of various portions of draft CS2013 report took place in several venues, including: the 42nd ACM Technical Symposium of the Special Interest Group on Computer Science Education (SIGCSE-11), the 24th IEEE-CS Conference on Software Engineering Education and Training (CSEET-11), the 2011 IEEE Frontiers in Education Conference (FIE-11), the 2011 Federated Computing Research Conference (FCRC-11), the 2nd Symposium on Educational Advances in Artificial Intelligence (EAAI-11), the Conference of ACM Special Interest Group on Data Communication 2011 (SIGCOMM-11), the 2011 IEEE International

Joint Conference on Computer, Information, and Systems Sciences and Engineering (CISSE-11), the 2011 Systems, Programming, Languages and Applications: Software for Humanity Conference (SPLASH-11), the 15th Colloquium for Information Systems Security Education, the 2011 National Centers of Academic Excellence in IA Education (CAE/IAE) Principles meeting, the 7th IFIP TC 11.8 World Conference on Information Security Education (WISE), the 43rd ACM Technical Symposium of the Special Interest Group on Computer Science Education (SIGCSE-12), the Special Session of the Special Interest Group on Computers and Society at SIGCSE-12, the Computer Research Association Snowbird Conference 2012, and the 2012 IEEE Frontiers in Education Conference (FIE-12), among others.

A number of organizations and working groups also provided valuable feedback to the CS2013 effort, including: the ACM Education Board and Council, the IEEE-CS Educational Activities Board, the ACM Practitioners Board, the ACM SIGPLAN Education Board, the ACM Special Interest Group Computers and Society, the Liberal Arts Computer Science Consortium (LACS), the NSF/IEEE-TCPP Curriculum Initiative on Parallel and Distributed Computing Committee, the Intel/NSF sponsored workshop on Security, and the NSF sponsored project on Curricular Guidelines for Cybersecurity.

# Chapter 2: Principles

Early in its work, the 2013 Steering Committee agreed on a set of principles to guide the development of this volume. The principles adopted for CS2013 overlap significantly with the principles adopted for previous curricular efforts, most notably CC2001 and CS2008. As with previous ACM/IEEE curricula volumes, there are a variety of constituencies for CS2013, including individual faculty members and instructors at a wide range of colleges, universities, and technical schools on any of six continents; CS programs and the departments, colleges, and institutions housing them; accreditation and certification boards; authors; and researchers. Other constituencies include pre-college preparatory schools and advanced placement curricula as well as graduate programs in computer science. These principles were developed in consideration of these constituencies, as well as taking account of issues related to student outcomes, development of curricula, and the review process. The order of presentation is not intended to imply relative importance.

1. *Computer Science curricula should be designed to provide students with the flexibility to work across many disciplines.* Computing is a broad field that connects to and draws from many disciplines, including mathematics, electrical engineering, psychology, statistics, fine arts, linguistics, and physical and life sciences. Computer Science students should develop the flexibility to work across disciplines.

2. *Computer Science curricula should be designed to prepare graduates for a variety of professions, attracting the full range of talent to the field.* Computer Science impacts nearly every modern endeavor. CS2013 takes a broad view of the field that includes topics such as "computational-x" (e.g., computational finance or computational chemistry) and "x-informatics" (e.g., eco-informatics or bio-informatics). Well-rounded CS graduates will have a balance of theory and application, as described in Chapter 3: Characteristics of Graduates.

3. *CS2013 should provide guidance for the expected level of mastery of topics by graduates*. It should suggest outcomes indicating the intended level of mastery and provide exemplars of instantiated courses and curricula that cover topics in the Body of Knowledge.

4. *CS2013 must provide realistic, adoptable recommendations that provide guidance and flexibility, allowing curricular designs that are innovative and track recent developments in the field.* The guidelines are intended to provide clear, implementable goals, while also providing the flexibility that programs need in order to respond to a rapidly changing field. CS2013 is intended as guidance, not as a minimal standard against which to evaluate a program.

5. *The CS2013 guidelines must be relevant to a variety of institutions.* Given the wide range of institutions and programs (including 2-year, 3-year, and 4-year programs; liberal arts, technological, and research institutions; and institutions of every size), it is neither possible nor desirable for these guidelines to dictate curricula for computing. Individual programs will need to evaluate their constraints and environments to construct curricula.

6. *The size of the essential knowledge must be managed*. While the range of relevant topics has expanded, the size of undergraduate education has not. Thus, CS2013 must carefully choose among topics and recommend the essential elements.

7. *Computer Science curricula should be designed to prepare graduates to succeed in a rapidly changing field.* Computer Science is rapidly changing and will continue to change for the foreseeable future. Curricula must prepare students for lifelong learning and must include professional practice (e.g., communication skills, teamwork, ethics) as components of the undergraduate experience. Computer science students must learn to integrate theory and practice, to recognize the importance of abstraction, and to appreciate the value of good engineering design.

8. *CS2013 should identify the fundamental skills and knowledge that all computer science graduates should possess while providing the greatest flexibility in selecting topics.* To this end, we have introduced three levels of knowledge description: Tier-1 Core, Tier-2 Core, and Elective. For a full discussion of Tier-1 Core, Tier-2 Core, and Elective, see Chapter 4: Introduction to the Body of Knowledge.

9. *CS2013 should provide the greatest flexibility in organizing topics into courses and curricula.* Knowledge areas are not intended to describe specific courses. There are many

novel, interesting, and effective ways to combine topics from the Body of Knowledge into courses.

10. *The development and review of CS2013 must be broadly based.* The CS2013 effort must include participation from many different constituencies including industry, government, and the full range of higher education institutions involved in Computer Science education. It must take into account relevant feedback from these constituencies.

# Chapter 3: Characteristics of Graduates

Graduates of Computer Science programs should have fundamental competency in the areas described by the Body of Knowledge (see Chapter 4), particularly the core topics contained there. However, there are also competences that graduates of CS programs should have that are not explicitly listed in the Body of Knowledge. Professionals in the field typically embody a characteristic style of thinking and problem solving, a style that emerges from the experiences obtained through study of the field and professional practice. Below, we describe the characteristics that we believe should be attained at least at an elementary level by graduates of Computer Science programs. These characteristics will enable their success in the field and further professional development. Some of these characteristics and skills also apply to other fields. They are included here because the development of these skills and characteristics should be explicitly addressed and encouraged by Computer Science programs. This list is based on a similar list in CC2001 and CS2008. The substantive changes that led to this new version were influenced by responses to a survey conducted by the CS2013 Steering Committee.

**At a broad level, the expected characteristics of computer science graduates include the following**:

### *Technical understanding of Computer Science*

Graduates should have a mastery of computer science as described by the core of the Body of Knowledge.

### *Familiarity with common themes and principles*

Graduates need understanding of a number of recurring themes, such as abstraction, complexity, and evolutionary change, and a set of general principles, such as sharing a common resource, security, and concurrency. Graduates should recognize that these themes and principles have broad application to the field of computer science and should not consider them as relevant only to the domains in which they were introduced.

### *Appreciation of the interplay between theory and practice*

A fundamental aspect of computer science is understanding the interplay between theory and practice and the essential links between them. Graduates of a Computer Science program need to understand how theory and practice influence each other.

### *System-level perspective*

Graduates of a computer science program need to think at multiple levels of detail and abstraction. This understanding should transcend the implementation details of the various components to encompass an appreciation for the structure of computer systems and the processes involved in their construction and analysis. They need to recognize the context in which a computer system may function, including its interactions with people and the physical world.

### *Problem solving skills*

Graduates need to understand how to apply the knowledge they have gained to solve real problems, not just write code and move bits. They should to be able to design and improve a system based on a quantitative and qualitative assessment of its functionality, usability and performance. They should realize that there are multiple solutions to a given problem and that selecting among them is not a purely technical activity, as these solutions will have a real impact on people's lives.  Graduates also should be able to communicate their solution to others, including why and how a solution solves the problem and what assumptions were made.

### *Project experience*

To ensure that graduates can successfully apply the knowledge they have gained, all graduates of computer science programs should have been involved in at least one substantial project. In most cases, this experience will be a software development project, but other experiences are also appropriate in particular circumstances. Such projects should challenge students by being integrative, requiring evaluation of potential solutions, and requiring work on a larger scale than typical course projects.   Students should have opportunities to develop their interpersonal communication skills as part of their project experience.

### *Commitment to life-long learning*

Graduates should realize that the computing field advances at a rapid pace, and graduates must possess a solid foundation that allows and encourages them to maintain relevant skills as the field evolves. Specific languages and technology platforms change over time. Therefore, graduates need to realize that they must continue to learn and adapt their skills throughout their careers. To develop this ability, students should be exposed to multiple programming languages, tools, paradigms, and technologies as well as the fundamental underlying principles throughout their education. In addition, graduates are now expected to manage their own career development and advancement. Graduates seeking career advancement often engage in professional development activities, such as certifications, management training, or obtaining domain-specific knowledge.

### *Commitment to professional responsibility*

Graduates should recognize the social, legal, ethical, and cultural issues inherent in the discipline of computing. They must further recognize that social, legal, and ethical standards vary internationally. They should be knowledgeable about the interplay of ethical issues, technical problems, and aesthetic values that play an important part in the development of computing systems. Practitioners must understand their individual and collective responsibility and the possible consequences of failure. They must understand their own limitations as well as the limitations of their tools.

### *Communication and organizational skills*

Graduates should have the ability to make effective presentations to a range of audiences about technical problems and their solutions. This may involve face-to-face, written, or electronic communication. They should be prepared to work effectively as members of teams. Graduates should be able to manage their own learning and development, including managing time, priorities, and progress.

### *Awareness of the broad applicability of computing*

Platforms range from embedded micro-sensors to high-performance clusters and distributed clouds. Computer applications impact nearly every aspect of modern life. Graduates should understand the full range of opportunities available in computing.

### *Appreciation of domain-specific knowledge*

Graduates should understand that computing interacts with many different domains. Solutions to many problems require both computing skills and domain knowledge. Therefore, graduates need to be able to communicate with, and learn from, experts from different domains throughout their careers.

# Chapter 4: Introduction to the Body of Knowledge

This chapter provides an introduction to the structure and rationale for the Body of Knowledge. It further describes the most substantial innovations in the Body of Knowledge. It does not propose a particular set of courses or curriculum structure -- that is the role of the course and curriculum exemplars. Rather, this chapter emphasizes the flexibility that the Body of Knowledge allows in adapting curricula to institutional needs and the continual evolution of the field. In Computer Science terms, one can view the Body of Knowledge as a specification of the content to be covered and a curriculum as an implementation. A large variety of curricula can meet the specification.

The following points are elaborated:

- Knowledge Areas are not intended to be in one-to-one correspondence with particular courses in a curriculum: We expect curricula will have courses that incorporate topics from multiple Knowledge Areas.
- Topics are identified as either "Core" or "Elective" with the core further subdivided into "Tier-1" and "Tier-2."
    - A curriculum should include all topics in the Tier-1 core and ensure that all students cover this material.
    - A curriculum should include all or almost all topics in the Tier-2 core and ensure that all students cover the vast majority of this material.
    - A curriculum should include significant elective material: Covering only "Core" topics is insufficient for a complete curriculum.
- Because it is a hierarchical outline, the Body of Knowledge under-emphasizes some key issues that must be considered when constructing a curriculum, such as the ways in which a curriculum allows students to develop the characteristics outlined in Chapter 3: *Characteristics of Graduates*.

- The learning outcomes and hour counts in the Body of Knowledge provide guidance on the depth of coverage toward which curricula should aim.
- There are several new Knowledge Areas that reflect important changes in the field.

## Knowledge Areas are Not Necessarily Courses (and Important Examples Thereof)

It is naturally tempting to associate each Knowledge Area with a course. We explicitly discourage this practice in general, even though many curricula will have some courses containing material from only one Knowledge Area or, conversely, all the material from one Knowledge Area in one course. We view the hierarchical structure of the Body of Knowledge as a useful way to group related information, not as a stricture for organizing material into courses. Beyond this general flexibility, in several places we expect many curricula to integrate material from multiple Knowledge Areas, in particular:

- *Introductory courses:* There are diverse successful approaches to introductory courses in computer science. Many focus on the topics in Software Development Fundamentals together with a subset of the topics in Programming Languages or Software Engineering, while leaving most of the topics in these other Knowledge Areas to advanced courses. But *which* topics from other Knowledge Areas are covered in introductory courses can vary. Some courses use object-oriented programming, others functional programming, others platform-based development (thereby covering topics in the Platform-Based Development Knowledge Area), etc. Conversely, there is no requirement that all Software Development Fundamentals be covered in a first or second course, though in practice most topics will usually be covered in these early courses. A separate chapter discusses introductory courses more generally.
- *Systems courses:* The topics in the Systems Fundamentals Knowledge Area can be presented in courses designed to cover general systems principles or in those devoted to particular systems areas such as computer architecture, operating systems, networking, or distributed systems. For example, an Operating Systems course might be designed to cover more general systems principles, such as low-level programming, concurrency and

synchronization, performance measurement, or computer security, in addition to topics more specifically related to operating systems. Consequently, such courses will likely draw on material in several Knowledge Areas. Certain fundamental systems topics like latency or parallelism will likely arise in many places in a curriculum. While it is important that such topics do arise, preferably in multiple settings, the Body of Knowledge does not specify the particular settings in which to teach such topics. The course exemplars in Appendix C show multiple ways that such material may be organized into courses.

- *Parallel computing:* Among the changes to the Body of Knowledge from previous reports is a new Knowledge Area in Parallel and Distributed Computing. An alternative structure for the Body of Knowledge would place relevant topics in other Knowledge Areas: parallel algorithms with algorithms, programming constructs in software-development focused areas, multi-core design with computer architecture, and so forth. We chose instead to provide guidance on the essential parallelism topics in one place. Some, but not all, curricula will likely have courses dedicated to parallelism, at least in the near term.

## Core Tier-1, Core Tier-2, Elective: What These Terms Mean, What is Required

As described at the beginning of this chapter, computer-science curricula should cover all the Core Tier-1 topics, all or almost all of the Core Tier-2 topics, and significant depth in many of the Elective topics (i.e., the core is not sufficient for an undergraduate degree in computer science). Here we provide additional perspective on what "Core Tier-1," "Core Tier-2", and "Elective" mean, including motivation for these distinctions.

***Motivation for subdividing the core:*** Earlier curricular guidelines had only "Core" and "Elective" with every topic in the former being required. We departed from this strict interpretation of "everything in the core must be taught to every student" for these reasons:

- Many strong computer-science curricula were missing at least one hour of core material. It is misleading to suggest that such curricula are outside the definition of an undergraduate degree in computer science.
- As the field has grown, there is ever-increasing pressure to grow the core and to allow students to specialize in areas of interest. Doing so simply becomes impossible within the short time-frame of an undergraduate degree. Providing some flexibility on coverage of core topics enables curricula and students to specialize if they choose to do so.

Conversely, we could have allowed for *any* core topic to be skipped provided that the vast majority was part of every student's education. By retaining a smaller Core Tier-1 of required material, we provide additional guidance and structure for curriculum designers. In the Core Tier-1 are the topics that are fundamental to the structure of any computer-science program.

***On the meaning of Core Tier-1:*** A Core Tier-1 topic should be a required part of every Computer Science curriculum for every student. While Core Tier-2 and Elective topics are important, the Core Tier-1 topics are those with widespread consensus for inclusion in every program. While most Core Tier-1 topics will typically be covered in introductory courses, others may be covered in later courses.

***On the meaning of Core Tier-2:*** Core Tier-2 topics are generally essential in an undergraduate computer-science degree. Requiring the vast majority of them is a *minimum* expectation, and if a program prefers to cover all of the Core Tier-2 topics, we encourage them to do so. That said, Computer Science programs can allow students to focus in certain areas in which some Core Tier-2 topics are not required. We also acknowledge that resource constraints, such as a small number of faculty or institutional limits on degree requirements, may make it prohibitively difficult to cover every topic in the core while still providing advanced elective material. **A computer-science curriculum should aim to cover 90-100% of the Core Tier-2 topics for every student, with 80% considered a minimum.**

There is no expectation that Core Tier-1 topics necessarily precede all Core Tier-2 topics in a curriculum. In particular, we expect introductory courses will draw on both Core Tier-1 and

Core Tier-2 (and possibly elective) material and that some core material will be delayed until later courses.

***On the meaning of Elective:*** A program covering only core material would provide insufficient breadth and depth in computer science. Most programs will not cover all the elective material in the Body of Knowledge and certainly few, if any, students will cover all of it within an undergraduate program. Conversely, the Body of Knowledge is by no means exhaustive, and advanced courses may often go beyond the topics and learning outcomes contained in it. Nonetheless, the Body of Knowledge provides a useful guide on material appropriate for a computer-science undergraduate degree, and all students of computer science should deepen their understanding in multiple areas via the elective topics.

A curriculum may well require material designated elective in the Body of Knowledge. Many curricula, especially those with a particular focus, will require some elective topics, by virtue of them being covered in required courses.

***The size of the core:*** The size of the core (Tier-1 plus Tier-2) is a few hours larger than in previous curricular guidelines, but this is counterbalanced by our more flexible treatment of the core. As a result, we are not increasing the number of required courses a curriculum should need. Indeed, a curriculum covering 90% of the Tier-2 hours would have the same number of core hours as a curriculum covering the core in the CS2008 volume, and a curriculum covering 80% of the Tier-2 hours would have fewer core hours than even a curriculum covering the core in the CC2001 volume (the core grew from 2001 to 2008). A more thorough quantitative comparison is presented at the end of this chapter.

***A note on balance:*** Computer Science is an elegant interplay of theory, software, hardware, and applications. The core in general and Tier-1 in particular, when viewed in isolation, may seem to focus on programming, discrete structures, and algorithms. This focus results from the fact that these topics typically come early in a curriculum so that advanced courses can use them as prerequisites. Essential experience with systems and applications can be achieved in more disparate ways using elective material in the Body of Knowledge. Because all curricula will include appropriate elective material, an overall curriculum can and should achieve an appropriate balance.

## Further Considerations in Designing a Curriculum

As useful as the Body of Knowledge is, it is important to complement it with a thoughtful understanding of cross-cutting themes in a curriculum, the “big ideas” of computer science. In designing a curriculum, it is also valuable to identify curriculum-wide objectives, for which the Principles and the Characteristics of Graduates chapters of this volume should prove useful.

In the last few years, two on-going trends have had deep effects on many curricula. First, the continuing growth of computer science has led to many programs organizing their curricula to allow for *intradisciplinary* specialization (using terms such as threads, tracks, vectors, etc.). Second, the importance of computing to almost every other field has increasingly led to the creation of *interdisciplinary* programs (joint majors, double majors, etc.) and incorporating interdisciplinary material into computer-science programs. We applaud both trends and believe a flexible Body of Knowledge, including a flexible core, supports them. Conversely, such specialization is not required: Many programs will continue to offer a broad yet thorough coverage of computer science as a distinct and coherent discipline.

## Organization of the Body of Knowledge

The CS2013 Body of Knowledge is presented as a set of Knowledge Areas (KAs), organized on topical themes rather than by course boundaries. Each KA is further organized into a set of Knowledge Units (KUs), which are summarized in a table at the head of each KA section. We expect that the topics within the KAs will be organized into courses in different ways at different institutions.

## Curricular Hours

Continuing in the tradition of CC2001/CS2008, we define the unit of coverage in the Body of Knowledge in terms of **lecture hours**, as being the sole unit that is understandable in (and transferable to) cross-cultural contexts. An “hour” corresponds to the time required to present the material in a traditional lecture-oriented format; the hour count does not include any additional work that is associated with a lecture (e.g., in self-study, lab classes, assessments, etc.). Indeed, we expect students to spend a significant amount of additional time outside of class developing facility with the material presented in class. As with previous reports, we maintain the principle that the use of a lecture-hour as the unit of measurement does not require or endorse the use of traditional lectures for the presentation of material.

The specification of topic hours represents the minimum amount of time we expect such coverage to take. Any institution may opt to cover the same material in a longer period of time as warranted by the individual needs of that institution.

## Courses

Throughout the Body of Knowledge, when we refer to a "course" we mean an institutionally-recognized unit of study. Depending on local circumstance, full-time students will take several "courses" at any one time, typically several per academic year. While "course" is a common term at some institutions, others will use other names, for example "module" or "paper."

## Guidance on Learning Outcomes

Each KU within a KA lists both a set of topics and the learning outcomes students are expected to achieve with respect to the topics specified. Each learning outcome has an associated *level of mastery*. In defining different levels we drew from other curriculum approaches, especially Bloom's Taxonomy, which has been well explored within computer science. We did not directly apply Bloom's levels in part because several of them are driven by pedagogic context, which would introduce too much plurality in a document of this kind; in part because we intend the mastery levels to be indicative and not to impose theoretical constraint on users of this document.

There are three levels of mastery, defined as:

- ***Familiarity***: The student understands what a concept is or what it means. This level of mastery concerns a basic awareness of a concept as opposed to expecting real facility with its application. It provides an answer to the question "What do you know about this?"

- ***Usage***: The student is able to use or apply a concept in a concrete way. Using a concept may include, for example, appropriately using a specific concept in a program, using a particular proof technique, or performing a particular analysis. It provides an answer to the question "What do you know how to do?"
- ***Assessment***: The student is able to consider a concept from multiple viewpoints and/or justify the selection of a particular approach to solve a problem. This level of mastery implies more than using a concept; it involves the ability to select an appropriate approach from understood alternatives. It provides an answer to the question "Why would you do that?"

As a concrete, although admittedly simplistic, example of these levels of mastery, we consider the notion of iteration in software development, for example for-loops, while-loops, iterators. At the level of "Familiarity," a student would be expected to have a definition of the concept of iteration in software development and know why it is a useful technique. In order to show mastery at the "Usage" level, a student should be able to write a program properly using a form of iteration. Understanding iteration at the "Assessment" level would require a student to understand multiple methods for iteration and be able to appropriately select among them for different applications.

## Overview of New Knowledge Areas

While computer science encompasses technologies that change rapidly over time, it is defined by essential concepts, perspectives, and methodologies that are constant. As a result, much of the core Body of Knowledge remains unchanged from earlier curricular volumes. However, new developments in computing technology and pedagogy mean that some aspects of the core evolve over time, and some of the previous structures and organization may no longer be appropriate for describing the discipline. As a result, CS2013 has modified the organization of the Body of Knowledge in various ways, adding some new KAs and restructuring others. We highlight these changes in the remainder of this section.

### IAS- Information Assurance and Security

IAS is a new KA in recognition of the world's critical reliance on information technology and computing. IAS as a domain is the set of controls and processes, both technical and policy, intended to protect and defend information and information systems. IAS draws together topics that are pervasive throughout other KAs.  Topics germane to *only* IAS are presented in depth in this KA, whereas other topics are noted and cross referenced to the KAs that contain them. As such, this KA is prefaced with a detailed table of cross-references to other KAs.

#### NC-Networking and Communication

CC2001 introduced a KA entitled "Net-Centric Computing" which encompassed a combination of topics including traditional networking, web development, and network security.  Given the growth and divergence in these topics since the last report, we renamed and re-factored this KA to focus specifically on topics in networking and communication.  Discussions of web applications and mobile device development are now covered in the new PBD-Platform-Based Development KA.  Security is covered in the new IAS-Information Assurance and Security KA.

### PBD-Platform-Based Development

PBD is a new KA that recognizes the increasing use of platform-specific programming environments, both at the introductory level and in upper-level electives.  Platforms such as the Web or mobile devices enable students to learn within and about environments constrained by hardware, APIs, and special services (often in cross-disciplinary contexts). These environments are sufficiently different from "general purpose" programming to warrant this new (wholly elective) KA.

### PD-Parallel and Distributed Computing

Previous curricular volumes had parallelism topics distributed across disparate KAs as electives. Given the vastly increased importance of parallel and distributed computing, it seemed crucial to identify essential concepts in this area and to promote those topics to the core.  To highlight and coordinate this material, CS2013 dedicates a KA to this area.  This new KA includes material on programming models, programming pragmatics, algorithms, performance, computer architecture, and distributed systems.

### SDF-Software Development Fundamentals

This new KA generalizes introductory programming to focus on more of the software development process, identifying concepts and skills that should be mastered in the first year of a computer-science program. As a result of its broad purpose, the SDF KA includes fundamental concepts and skills that could appear in other software-oriented KAs (e.g., programming constructs from Programming Languages, simple algorithm analysis from Algorithms and Complexity, simple development methodologies from Software Engineering). Likewise, each of those KAs will contain more advanced material that builds upon the fundamental concepts and skills in SDF. Compared to previous volumes, key approaches to programming -- including object-oriented programming, functional programming, and event-driven programming -- are kept in one place, namely the Programming Languages KA, with an expectation that any curriculum will cover some of these topics in introductory courses.

### SF-Systems Fundamentals

In previous curricular volumes, the interacting layers of a typical computing system, from hardware building blocks, to architectural organization, to operating system services, to application execution environments (particularly for parallel execution in a modern view of applications), were presented in independent knowledge areas. The new Systems Fundamentals KA presents a unified systems perspective and common conceptual foundation for other KAs (notably Architecture and Organization, Network and Communications, Operating Systems, and Parallel and Distributed Algorithms). An organizational principle is “programming for performance”: what a programmer needs to understand about the underlying system to achieve high performance, particularly in terms of exploiting parallelism.

## Core Hours in Knowledge Areas

An overview of the number of core hours (both Tier-1 and Tier-2) by KA in the CS2013 Body of Knowledge is provided below.   For comparison, the number of core hours from both the previous CS2008 and CC2001 reports are provided as well.

| | **CS2013** | | **CS2008** | **CC2001** |
|---|---|---|---|---|
| **Knowledge Area** | **Tier1** | **Tier2** | **Core** | **Core** |
| AL-Algorithms and Complexity | 19 | 9 | 31 | 31 |
| AR-Architecture and Organization | 0 | 16 | 36 | 36 |
| CN-Computational Science | 1 | 0 | 0 | 0 |
| DS-Discrete Structures | 37 | 4 | 43 | 43 |
| GV-Graphics and Visual Computing | 2 | 1 | 3 | 3 |
| HCI-Human-Computer Interaction | 4 | 4 | 8 | 8 |
| IAS-Security and Information Assurance | 3 | 6 | -- | -- |
| IM-Information Management | 1 | 9 | 11 | 10 |
| IS-Intelligent Systems | 0 | 10 | 10 | 10 |
| NC-Networking and Communication | 3 | 7 | 15 | 15 |
| OS-Operating Systems | 4 | 11 | 18 | 18 |
| PBD-Platform-based Development | 0 | 0 | -- | -- |
| PD-Parallel and Distributed Computing | 5 | 10 | -- | -- |
| PL-Programming Languages | 8 | 20 | 21 | 21 |
| SDF-Software Development Fundamentals | 43 | 0 | 47 | 38 |
| SE-Software Engineering | 6 | 21 | 31 | 31 |
| SF-Systems Fundamentals | 18 | 9 | -- | -- |
| SP-Social Issues and Professional Practice | 11 | 5 | 16 | 16 |
| **Total Core Hours** | **165** | **142** | **290** | **280** |

| | |
|---|---|
| **All Tier1 + All Tier2 Total** | **307** |
| **All Tier1 + 90% of Tier2 Total** | **292.8** |
| **All Tier1 + 80% of Tier2 Total** | **278.6** |

As seen above, in CS2013 the total Tier-1 hours together with the entirety of Tier-2 hours slightly exceeds the total core hours from previous reports.  However, it is important to note that the tiered structure of the core in CS2013 explicitly provides the flexibility for institutions to select topics from Tier-2 (to include at least 80%). As a result, it is possible to implement the CS2013 guidelines with comparable hours to previous curricular guidelines.

# Chapter 5: Introductory Courses

Computer science, unlike many technical disciplines, does not have a well-described list of topics that appear in virtually all introductory courses. In considering the changing landscape of introductory courses, we look at the evolution of such courses from CC2001 to CS2013. CC2001 classified introductory course sequences into six general models: *Imperative-first, Objects-first, Functional-first, Breadth-first, Algorithms-first*, and *Hardware-first*. While introductory courses with these characteristic features certainly still exist today, we believe that advances in the field have led to a more diverse set of approaches in introductory courses than the models set out in CC2001. Moreover, the approaches employed in introductory courses are in a greater state of flux.

An important challenge for introductory courses, and a key reason the content of such courses remains a vigorous discussion topic after decades of debate, is that not everything (programming, software processes, algorithms, abstraction, performance, security, professionalism, etc.) can be taught from day one. In other words, not everything can come first and as a result some topics must be pushed further back in the curriculum, in some cases significantly so. Some topics will not appear in a first course or even a second course, meaning that students who do not continue further (for example, non-majors) will lose exposure to some topics. Ultimately, choosing what to cover in introductory courses results in a set of tradeoffs that must be considered when trying to decide what should be covered early in a curriculum.

## Design Dimensions

We structure this chapter as a set of design dimensions relevant to crafting introductory courses, concluding each dimension with a summary of the trade-offs that are in tension along the dimension. A given introductory course, or course sequence, in computer science will represent different decisions within the multidimensional design space (described below) and achieve different outcomes as a result. We note that our discussion here focuses on introductory courses meant as part of a undergraduate program in computer science. Notably, we do not discuss "CS0" courses, which are increasingly offered -- often focusing on computer fluency or computational thinking. Such courses are not part of this Body of Knowledge and are beyond the scope of our consideration.

## Pathways Through Introductory Courses

We recognize that introductory courses are not constructed in the abstract, but rather are designed for specific target audiences and contexts. Departments know their institutional contexts best and must be sensitive to their own needs and target audiences.

Introductory courses differ across institutions, especially with regard to the nature and length of an introductory *sequence* of courses (that is, the number of courses that a student must take before any branching is allowed). A sequence of courses may also accommodate students with significant differences in previous computing experience and/or who come from a wide diversity of backgrounds. Having multiple pathways into and through the introductory course sequence can help to better align students' abilities with the appropriate level of coursework. It can also help create more flexibility with articulation from two-year to four-years institutions, and smooth the transition for students transferring from other colleges/programs. Additionally, having multiple pathways through introductory courses may provide greater options to students who choose late in their college programs to take courses in computing.

Building courses for diverse audiences--not just students who are already sure of a major in computer science--is essential for making computing accessible to a wide range of students. Given the importance of computation across many disciplines, the appeal of introductory programming courses has significantly broadened beyond just students in engineering fields. For target audiences with different backgrounds, and different expectations, the practice of having thematically-focused introductory courses (e.g., computational biology, robotics, digital media manipulation, etc.) has become popular. In this way, material is made relevant to the expectations and aspirations of students with a variety of disciplinary orientations.

***Tradeoffs***: Providing multiple pathways into and through introductory course sequences can make CS more accessible to different audiences, but requires greater investment (in work and resources) by a department to construct such pathways and/or provide different themed options to students. Moreover, care must be taken to give students (often with no prior computing experience) appropriate guidance with regard to choosing the appropriate introductory course pathway for them to take. By having longer introductory course sequences (i.e., longer or more structured pre-requisite chains), educators can assume more prior knowledge in each course, but such lengthy sequences sacrifice flexibility and increase the time before students are able to take advanced courses more focused on their areas of interest.

## Platform

While many introductory programming courses make use of traditional computing platforms (e.g., desktop/laptop computers) and are, as a result, somewhat "hardware agnostic", the past few years have seen a growing diversity in the set of programmable devices that are employed in such courses. For example, some introductory courses may choose to engage in web development or mobile device (e.g., smartphone, tablet) programming. Others have examined the used of specialty platforms, such as robots or game consoles, which may help generate more enthusiasm for the subject among novices as well as foregrounding interaction with the external world as an essential and natural focus. Recent developments have led to physically-small, feature-restricted computational devices constructed specifically for the purpose of facilitating learning programming (e.g., raspberry-pi). In any of these cases, the use of a particular platform brings with it attendant choices for programming paradigms, component libraries, APIs, and security. Working within the software/hardware constraints of a given platform is a useful software-engineering skill, but also comes at the cost that the topics covered in the course may likewise be limited by the choice of platform.

***Tradeoffs**:* The use of specific platforms can bring compelling real-world contexts into the classroom and platforms designed for pedagogy can have beneficial focus. However, it can take considerable effort not to let platform-specific details swamp pedagogic objectives. Moreover, the specificity of the platform may impact the transferability of course content to downstream courses.

## Programming Focus

The vast majority of introductory courses are *programming-focused*, in which students learn about concepts in computer science (e.g., abstraction, decomposition, etc.) through the explicit tasks of learning a given programming language and building software artifacts. A programming focus can provide early training in this crucial skill for computer science majors, and help elevate students with different backgrounds in computing to a more equal footing. Some introductory courses are designed to provide a broader introduction to concepts in computing without the constraints of learning the syntax of a programming language. They are consciously programming de-focused. Such a perspective is roughly analogous to the "Breadth-first" model in CC2001. Whether or not programming is the primary focus of their first course, it is important that students do not perceive computer science as *only* learning the specifics of particular programming languages. Care must be taken to emphasize the more general concepts in computing within the context of learning how to program.

***Tradeoffs****:* A programming-focused introductory course can help develop important software-engineering skills in students early on. This may be also be useful for students from other areas of study who wish to use programming as a tool in cross-disciplinary work. However, too narrow a programming focus in an introductory class, while giving immediate facility in a programming language, can also give students too a narrow view of the field. Such a narrow perspective may limit the appeal of computer science to some students.

## Programming Paradigm and Choice of Language

A defining factor for many introductory courses is the choice programming paradigm, which then drives the choice of programming language. Indeed, half of the six introductory course models listed in CC2001 were described by programming paradigm (Imperative-first, Objects-first, Functional-first). Such paradigm-based introductory courses still exist and their relative merits continue to be debated. We note that rather than a particular paradigm or language coming to be favored over time, the past decade has only broadened the list of programming languages now considered in introductory courses. There does, however, appear to be a growing trend toward "safer" or more managed languages (for example, moving from C to Java) as well as the use of more dynamic languages, such as Python or JavaScript. Visual programming languages, such as Alice and Scratch, have also become popular choices to provide a “syntax-light” introduction to programming; these are often (although not exclusively) used with non-majors or at the start of an introductory course. Some introductory course sequences choose to provide a presentation of alternative programming paradigms, such as scripting vs. procedural programming or functional vs. object-oriented programming, to give students a greater appreciation of the diverse perspectives in programming, to avoid language-feature fixation, and to disabuse them of the notion that there is a single “correct” or “best” programming language.

***Tradeoffs****:* The use of “safer” or more managed languages can help increase programmer productivity and provide a more forgiving programming environment. But, such languages may provide a level of abstraction that obscures an understanding of actual machine execution and makes is difficult to evaluate performance trade-offs. The decision as to whether to use a “lower-level” language to promote a mental model of program execution that is closer to the actual execution by the machine is often a matter of local audience needs.

### Software Development Practices

While programming is the means by which software is constructed, an introductory course may choose to present additional practices in software development to different extents. For example, the use of software development best practices, such as unit testing, version control systems, industrial integrated development environments (IDEs), and programming patterns may be stressed to different extents in different introductory courses. The inclusion of such software development practices can help students gain an early appreciation of some of the challenges in developing real software projects. On the other hand, while all computer scientists should have solid software development skills, those skills need not always be the primary focus of the first introductory programming course, especially if the intended audience is not just computer science majors. Care should be taken in introductory courses to balance the use of software development best practices from the outset with making introductory courses accessible to a broad population.

***Tradeoffs**:* The inclusion of software development practices in introductory courses can help students develop important aspects of real-world software development early on. The extent to which such practices are included in introductory courses may impact and be impacted by the target audience for the course.

### Parallel Processing

Traditionally, introductory courses have assumed the availability of a single processor, a single process, and a single thread, with the execution of the program being completely driven by the programmer's instructions and expectation of sequential execution. Recent hardware and software developments have prompted educators to re-think these assumptions, even at the introductory level -- multicore processors are now ubiquitous, user interfaces lend themselves to asynchronous event-driven processing, and "big data" requires parallel processing and distributed storage. As a result, some introductory courses stress parallel processing from the outset (with traditional single threaded execution models being considered a special case of the more general parallel paradigm). While we believe this is an interesting model to consider in the long-term, we anticipate that introductory courses will still be dominated by the "single thread of execution" model (perhaps with the inclusion of GUI-based event-driven programming) for the foreseeable future. As more successful pedagogical approaches are developed to make parallel processing more readily understandable to novice programmers and paradigms for parallel programming become more commonplace, we may begin to see more elements of parallel programming appearing in introductory courses.

***Tradeoffs**:* Understanding parallel processing is of growing importance for computer science majors and learning such models early on can give students more practice in this arena. On the other hand, parallel programming can be difficult to understand, especially for novice programmers.

## Mapping to the Body of Knowledge

Practically speaking, an introductory course sequence should not be construed as simply containing only the topics from the Software Development Fundamentals (SDF) Knowledge

Area. Rather we encourage implementers of the CS2013 guidelines to think about the design space dimensions outlined above to draw on materials from multiple KAs for inclusion in an introductory course sequence. For example, even a fairly straightforward introductory course sequence will likely augment material from SDF with topics from the Programming Languages Knowledge Area related to the choice of language used in the course and potentially some concepts from Software Engineering. More broadly, a course using non-traditional platforms will draw from topics in Platform-Based Development and those emphasizing multi-processing will naturally include material from Parallel and Distributed Computing. We encourage readers to think of the CS2013 Body of Knowledge as an invitation for the construction of creative new introductory course sequences that best fit the needs of students at one's local institution.

# Chapter 6: Institutional Challenges

While the Body of Knowledge provides a detailed specification of what content should be included in an undergraduate computer science curriculum, it is not to be taken as the sum total of what an undergraduate curriculum in computing should impart. In a rapidly moving field such as Computer Science, the particulars of what is taught are complementary to promoting a sense of on-going inquiry, helping students construct a framework for the assimilation of new knowledge, and advancing students' development as responsible professionals. Critical thinking, problem solving, and a foundation for life-long learning are skills that students need to develop throughout their undergraduate career. Education is not just the transmission of information, but at its best inspires passion for a subject, gives students encouragement to experiment and allows them to experience excitement in achievement. These things, too, need to be reflected in computer science curriculum and pedagogy.

## Localizing CS2013

Successfully deploying an updated computer science curriculum at any individual institution requires sensitivity to local needs. CS2013 should not be read as a set of topical "check-boxes" to tick off, in a one-to-one mapping of classes to Knowledge Areas. Rather, we encourage institutions to think about ways in which the Body of Knowledge may be best integrated into a unique set of courses that reflect an institution's mission, faculty strength, student needs, and employer demands. Indeed, we created the two-tier structure of the Core precisely to provide such flexibility, keeping the Core Tier-1 material to an essential minimum to allow institutions greater leeway in selecting Core Tier-2 material to best suit their needs.

## Actively Promoting Computer Science

Beyond coursework, we also stress the importance of advising, mentoring, and fostering relationships among faculty and students. Many students, perhaps especially those coming from disadvantaged backgrounds, may not appreciate the full breadth of career options that a degree in Computer Science can provide. Advertising and promoting the possibilities opened by studying computer science, especially when customized to local employer needs, provides two benefits. First, it serves students by giving them information regarding career options they may not have considered. Second, it serves the department by helping to attract more students (potentially from a broader variety of backgrounds) into computer science courses. Offering a healthy computer science program over time requires maintaining a commitment to attracting students to the field regardless of current enrollment trends (which have ebbed and flowed quite widely in recent decades).

It is important to note also that many students still feel that studying computer science is equated with working as a "programmer," which in turn raises negative and incorrect stereotypes of isolated and rote work. At the same time, some students believe that if they do not already have significant prior programming experience, they will not be competitive in pursuing a degree in computer science. We strongly encourage departments to challenge both these perceptions. Extra-curricular activities aimed at showcasing potential career paths opened by a degree in computer science (for example, by inviting alumni to talk to current students) can help to show both that there are many possibilities beyond "being a programmer" and also that software development is a significantly creative and collaborative process. In these efforts, an accessible curriculum with multiple entry points, allowing students with or without prior experience to smoothly transfer into a computer science degree program, is an important desideratum.

## Broadening Participation

There is no doubt that there is a tremendous demand for students with computing skills. Indeed, vast shortfalls in information technology workers in the coming decade have been predicted [3]. As a result, there is a pressing need to broaden participation in the study of computer science and attract the full range of talent to the field, regardless of ethnicity, gender, or economic status. Institutions should make efforts to bring a wide range of students into the computer science pipeline and provide support structures to help all students successfully complete their programs.

## Computer Science Across Campus

An argument can be made that Computer Science is becoming one of the core disciplines of a 21st Century University education; that is, something that any educated individual must possess some level of proficiency and understanding. This transcends its role as a tool and methodology for research broadly across disciplines; it is likely that in the near future, at many universities, every undergraduate student will take some instruction in computer science, in recognition of computational thinking as being one of the fundamental skills desired of all graduates. There are implications for Institutional resources to support such a significant scaling up of the teaching mission of computer science departments, particularly in terms of instructors and laboratories.

While CS2013 provides guidelines for undergraduate programs in Computer Science, we believe it is important for departments to provide computing education across a broad range of subject areas. To this end, computing departments may consider providing courses, especially at the introductory level, which are accessible and attractive to students from many disciplines. This also serves the dual purpose of attracting more students to the computing field who may not have had an initial inclination otherwise.

More broadly, as computing becomes an essential tool in other disciplines, it benefits computer science departments to be "outward facing", building bridges to other departments and curriculum areas, encouraging students to engage in multidisciplinary work, and promoting programs which span Computer Science and other fields of study (for example, programs in "Computational X", where X represents other disciplines such as biology or economics).

## Computer Science Minors

Further to positioning Computer Science as one of the core disciplines of the university, departments may also consider providing minors in Computer Science. A minor should provide flexible options for students to gain coherent knowledge of computer science beyond that captured in one or two courses, yet encompass less than a full program. Indeed, the use of minors can provide yet another means to allow students majoring in other disciplines to gain a solid foundation in computing for future work at the intersections of their fields.

It is well-known that students often make undergraduate major choices with highly varied levels of actual knowledge about different programs. As a result some students choose to not pursue a major in Computer Science simply as a result of knowing neither what computer science actually entails nor whether they might like the discipline, due to lack of prior exposure. A minor in Computer Science allows such students to still gain some credential in computing, if they discover late in their academic career that they have an interest in computing and what it offers. To give students the ability to major in Computer Science, "taster" courses should seek to reach students as soon as possible in their undergraduate studies.

## Computing Resources

Programs in computer science have a need for adequate computing resources, both for students and faculty. The needs of computer science programs often extend beyond traditional infrastructure (general campus computing labs) and may include specialized hardware and software, and/or large-scale computing infrastructure. Having adequate access to such resources is especially important for project and capstone courses. Moreover, institutions need to consider the growing heterogeneity of computing devices (e.g., smartphones, tablets, etc.) which can be used as a platform for coursework.

## Maintaining a Flexible and Healthy Faculty

The foundation of a strong program in computer science is sufficient (and sufficiently experienced) faculty to keep a department healthy and vibrant. Departmental hiring should provide not only sufficient capacity to keep a program viable, but also allow for existing faculty to have time for professional development and exploration of new ideas. To respond to rapid changes in the field, computer science faculty must have the opportunities to build new skills, learn about new areas, and stay abreast of new technologies. While there can be tension between teaching new technologies versus fundamental principles, focusing too far on either extreme will be a disservice to students. Faculty need to be given the time to acquire new ideas and technologies and bring them into courses and curricula. In this way, departments can model the value of professional and lifelong learning, as faculty incorporate new materials and approaches.

In addition to professional development, it is especially important for computer science programs to maintain a healthy capacity to respond to enrollment fluctuations. Indeed, Computer Science as a discipline has gone through several boom-and-bust cycles in the past decades which have resulted in significant enrollment changes in programs all over the world and across virtually all types of institutions. A department should take care to create structures to help it maintain resilience in the face of enrollment downturns, for example by making courses more broadly accessible, building interdisciplinary programs with other departments, and offering service courses.

In the face of large sustained enrollment increases (as has been witnessed in recent years), the need for sufficient faculty hiring can become acute. Without sufficient capacity, faculty can be strained by larger course enrollments (each course requiring more sections and more student assessment) and more teaching obligations (more courses must be taught by each faculty member), which can result in lower quality instruction and potential faculty burn-out. The former issue causes students to abandon computer science. These outcomes are highly detrimental given the need to produce more, and more skilled, computing graduates as discussed above. Excellent arguments for the need to maintain strong faculty capacity in the face of growing enrollment have been extended, both in relation to the most recent boom [5] and extending back more than three decades [2].

## Teaching Faculty

Permanent faculty, whose primary criteria for evaluation is based on teaching and educational contributions (broadly defined), can be instrumental in helping to build accessible courses, engage in curricular experimentation and revision, and provide outreach efforts to bring more students into the discipline. As with all institutional challenges, such appointments represent a balance of political and pragmatic issues. However, the value of this type of position was originally observed in CC2001; that value has not diminished in the intervening decades, and has more recently received additional endorsement [7].

## Undergraduate Teaching Assistants

Whilst research universities have traditionally drawn on postgraduate students to serve as teaching assistants in the undergraduate curriculum, over the past 20 years, growing numbers of departments have found it valuable to engage advanced undergraduates as teaching assistants in introductory computing courses. The reported benefits to the undergraduate teaching assistants include learning the material themselves when they are put in the role of helping teach it to someone else, better time management, improved ability dealing with organizational responsibilities, and presentation skills [4, 6]. Students in the introductory courses also benefit by having a larger course staff available, more accessible staff, and getting assistance from a "near-peer", someone with a recent familiarity in the kind of questions and struggles the student is likely facing.

## Online Education

It has been suggested that there is a tsunami coming to higher education, brought on by online learning, and lately, MOOCs [1]. Discussing the full scope of the potential and pitfalls of online education is well beyond the scope of this document. Rather, we simply point out some aspects of online learning that may impact the ways in which departments deploy these guidelines.

First, online educational materials need not be structured as just full term-long classes. As a result, it may be possible to teach online mini-courses or modules (which are less than a term long, potentially significantly so), that nevertheless contain coherent portions of the CS2013 Body of Knowledge. In this way, some departments, especially those with limited faculty resources, may choose to seek out and leverage online materials offered elsewhere. Blended learning is another model that has and can be pursued to accrue the benefits of both face-to-face and online learning in the same course.

Part of the excitement that has been generated by Massive Open Online Courses (MOOCs) is that they allow for ready scaling to large numbers of students. There are technological challenges in assessing programming assignments at scale, and there are those who believe that this represents a significant new research opportunity for computer science. The quantitative ability that MOOC platforms provide for assessing the effectiveness of how students learn has the potential to transform the teaching of computer science itself.

While we appreciate the value of scaling course availability, we also note that there are important aspects of education that are not concerned with course content or the transmission of information, e.g., pedagogy, scaffolding learning. Then again, while MOOCs are a powerful medium for content delivery, we note that it is important to make sure that characteristics of CS graduates are still developed.

## References


[1] Ken Auletta, April 30, 2012. *Get Rich U.*, The New Yorker.

[2] Kent K. Curtis. Computer manpower: Is there a crisis? National Science Foundation, 1982.

[3] Microsoft Corporation. *A National Talent Strategy: Ideas for Securing U.S. Competitiveness and Economic Growth*. 2012

[4] Stuart Reges, John McGrory, and Jeff Smith, "The effective use of undergraduates to staff large introductory CS courses," Proceedings of the Nineteenth SIGCSE Technical Symposium on Computer Science Education, Atlanta, Georgia, February 1988.

[5] Eric Roberts, "Meeting the challenges of rising enrollments," *ACM Inroads*, September 2011.

[6] Eric Roberts, John Lilly, and Bryan Rollins, "Using undergraduates as teaching assistants in introductory programming courses: an update on the Stanford experience," Proceedings of the Twenty-sixth SIGCSE Technical Symposium on Computer Science Education, Nashville, Tennessee, March 1995.

[7] Steve Wolfman, Owen Astrachan, Mike Clancy, Kurt Eiselt, Jeffrey Forbes, Diana Franklin, David Kay, Mike Scott, and Kevin Wayne. "Teaching-Oriented Faculty at Research Universities." Communications of the ACM. November 2011, v. 54. n 11. pp. 35-37.

# Appendix A: The Body of Knowledge

## Algorithms and Complexity (AL)

Algorithms are fundamental to computer science and software engineering. The real-world performance of any software system depends on: (1) the algorithms chosen and (2) the suitability and efficiency of the various layers of implementation. Good algorithm design is therefore crucial for the performance of all software systems. Moreover, the study of algorithms provides insight into the intrinsic nature of the problem as well as possible solution techniques independent of programming language, programming paradigm, computer hardware, or any other implementation aspect.

An important part of computing is the ability to select algorithms appropriate to particular purposes and to apply them, recognizing the possibility that no suitable algorithm may exist. This facility relies on understanding the range of algorithms that address an important set of well-defined problems, recognizing their strengths and weaknesses, and their suitability in particular contexts. Efficiency is a pervasive theme throughout this area.

This knowledge area defines the central concepts and skills required to design, implement, and analyze algorithms for solving problems. Algorithms are essential in all advanced areas of computer science: artificial intelligence, databases, distributed computing, graphics, networking, operating systems, programming languages, security, and so on. Algorithms that have specific utility in each of these are listed in the relevant knowledge areas. Cryptography, for example, appears in the new knowledge area on Information Assurance and Security, while parallel and distributed algorithms appear in PD-Parallel and Distributed Computing.

As with all knowledge areas, the order of topics and their groupings do not necessarily correlate to a specific order of presentation. Different programs will teach the topics in different courses and should do so in the order they believe is most appropriate for their students.

## AL. Algorithms and Complexity (19 Core-Tier1 hours, 9 Core-Tier2 hours)

| | **Core-Tier1 hours** | **Core-Tier2 hours** | **Includes Electives** |
|---|---|---|---|
| **AL/Basic Analysis** | **2** | **2** | **N** |
| **AL/Algorithmic Strategies** | **5** | **1** | **N** |
| **AL/Fundamental Data Structures and Algorithms** | **9** | **3** | **N** |
| **AL/Basic Automata, Computability and Complexity** | **3** | **3** | **N** |
| **AL/Advanced Computational Complexity** | | | **Y** |
| **AL/Advanced Automata Theory and Computability** | | | **Y** |
| **AL/Advanced Data Structures, Algorithms, and Analysis** | | | **Y** |

# AL/Basic Analysis

*[2 Core-Tier1 hours, 2 Core-Tier2 hours]*

***Topics:***

[Core-Tier1]

- Differences among best, expected, and worst case behaviors of an algorithm
- Asymptotic analysis of upper and expected complexity bounds
- Big O notation: formal definition
- Complexity classes, such as constant, logarithmic, linear, quadratic, and exponential
- Empirical measurements of performance
- Time and space trade-offs in algorithms

[Core-Tier2]

- Big O notation: use
- Little o, big omega and big theta notation
- Recurrence relations
- Analysis of iterative and recursive algorithms
- Some version of a Master Theorem

***Learning Outcomes:***

[Core-Tier1]

1. Explain what is meant by “best”, “expected”, and “worst” case behavior of an algorithm. [Familiarity]
2. In the context of specific algorithms, identify the characteristics of data and/or other conditions or assumptions that lead to different behaviors. [Assessment]
3. Determine informally the time and space complexity of simple algorithms. [Usage]

4. Understand the formal definition of big O. [Familiarity]
5. List and contrast standard complexity classes. [Familiarity]
6. Perform empirical studies to validate hypotheses about runtime stemming from mathematical analysis. Run algorithms on input of various sizes and compare performance. [Assessment]
7. Give examples that illustrate time-space trade-offs of algorithms. [Familiarity]

[Core-Tier2]

8. Use big O notation formally to give asymptotic upper bounds on time and space complexity of algorithms. [Usage]
9. Use big O notation formally to give expected case bounds on time complexity of algorithms. [Usage]
10. Explain the use of big omega, big theta, and little o notation to describe the amount of work done by an algorithm. [Familiarity]
11. Use recurrence relations to determine the time complexity of recursively defined algorithms. [Usage]
12. Solve elementary recurrence relations, e.g., using some form of a Master Theorem. [Usage]

## AL/Algorithmic Strategies

### *[5 Core-Tier1 hours, 1 Core-Tier2 hours]*

An instructor might choose to cover these algorithmic strategies in the context of the algorithms presented in "Fundamental Data Structures and Algorithms" below. While the total number of hours for the two knowledge units (18) could be divided differently between them, our sense is that the 1:2 ratio is reasonable.

***Topics:***

[Core-Tier1]

- Brute-force algorithms
- Greedy algorithms
- Divide-and-conquer (cross-reference SDF/Algorithms and Design/Problem-solving strategies)
- Recursive backtracking
- Dynamic Programming

[Core-Tier2]

- Branch-and-bound
- Heuristics
- Reduction: transform-and-conquer

***Learning Outcomes:***

[Core-Tier1]

1. For each of the above strategies, identify a practical example to which it would apply. [Familiarity]
2. Have facility mapping pseudocode to implementation, implementing examples of algorithmic strategies from scratch, and applying them to specific problems. [Usage]
3. Use a greedy approach to solve an appropriate problem and determine if the greedy rule chosen leads to an optimal solution. [Usage, Assessment]
4. Use a divide-and-conquer algorithm to solve an appropriate problem. [Usage]
5. Use recursive backtracking to solve a problem such as navigating a maze. [Usage]
6. Use dynamic programming to solve an appropriate problem. [Usage]

[Core-Tier2]

7. Describe various heuristic problem-solving methods. [Familiarity]
8. Use a heuristic approach to solve an appropriate problem. [Usage]
9. Describe the trade-offs between brute force and other strategies. [Assessment]

## AL/Fundamental Data Structures and Algorithms

### *[9 Core-Tier1 hours, 3 Core-Tier2 hours]*

This knowledge unit builds directly on the foundation provided by Software Development Fundamentals (SDF), particularly the material in SDF/Fundamental Data Structures and SDF/Algorithms and Design.

***Topics:***

[Core-Tier1]

Implementation and use of:

- Simple numerical algorithms, such as computing the average of a list of numbers, finding the min, max, and mode in a list, approximating the square root of a number, or finding the greatest common divisor
- Sequential and binary search algorithms
- Worst case quadratic sorting algorithms (selection, insertion)
- Worst or average case O(N log N) sorting algorithms (quicksort, heapsort, mergesort)
- Hash tables, including strategies for avoiding and resolving collisions
- Binary search trees
  - Common operations on binary search trees such as select min, max, insert, delete, iterate over tree
- Graphs and graph algorithms
  - Representations of graphs (e.g., adjacency list, adjacency matrix)
  - Depth- and breadth-first traversals

[Core-Tier2]

- Heaps
- Graphs and graph algorithms
  - Shortest-path algorithms (Dijkstra's and Floyd's algorithms)
  - Minimum spanning tree (Prim's and Kruskal's algorithms)
- Pattern matching and string/text algorithms (e.g., substring matching, regular expression matching, longest common subsequence algorithms)

***Learning Outcomes:***

[Core-Tier1]

1. Implement basic numerical algorithms. [Usage]
2. Implement simple search algorithms and explain the differences in their time complexities. [Usage, Assessment]
3. Be able to implement common quadratic and O(N log N) sorting algorithms. [Usage]
4. Understand the implementation of hash tables, including collision avoidance and resolution. [Familiarity]
5. Discuss the runtime and memory efficiency of principal algorithms for sorting, searching, and hashing. [Familiarity]
6. Discuss factors other than computational efficiency that influence the choice of algorithms, such as programming time, maintainability, and the use of application-specific patterns in the input data. [Familiarity]
7. Solve problems using fundamental graph algorithms, including depth-first and breadth-first search. [Usage]

8. Demonstrate the ability to evaluate algorithms, to select from a range of possible options, to provide justification for that selection, and to implement the algorithm in a particular context. [Usage, Assessment]

[Core-Tier2]

9. Understand the heap property and the use of heaps as an implementation of priority queues. [Familiarity]
10. Solve problems using graph algorithms, including single-source and all-pairs shortest paths, and at least one minimum spanning tree algorithm. [Usage]
11. Be able to implement a string-matching algorithm. [Usage]

## AL/Basic Automata Computability and Complexity

### *[3 Core-Tier1 hours, 3 Core-Tier2 hours]*

***Topics:***

[Core-Tier1]

- Finite-state machines
- Regular expressions
- The halting problem

[Core-Tier2]

- Context-free grammars (cross-reference PL/Syntax Analysis)
- Introduction to the P and NP classes and the P vs NP problem
- Introduction to the NP-complete class and exemplary NP-complete problems (e.g., SAT, Knapsack)

***Learning Outcomes:***

[Core-Tier1]

1. Discuss the concept of finite state machines. [Familiarity]
2. Design a deterministic finite state machine to accept a specified language. [Usage]
3. Generate a regular expression to represent a specified language. [Usage]
4. Explain why the halting problem has no algorithmic solution. [Familiarity]

[Core-Tier2]

5. Design a context-free grammar to represent a specified language. [Usage]
6. Define the classes P and NP. [Familiarity]
7. Explain the significance of NP-completeness. [Familiarity]

## AL/Advanced Computational Complexity

### *[Elective]*

***Topics:***

- Review definitions of the classes P and NP; introduce P-space and EXP
- NP-completeness (Cook's theorem)
- Classic NP-complete problems
- Reduction Techniques

***Learning Outcomes:***

1. Define the classes P and NP. (Also appears in AL/Basic Automata, Computability, and Complexity) [Familiarity]
2. Define the P-space class and its relation to the EXP class. [Familiarity]
3. Explain the significance of NP-completeness. (Also appears in AL/Basic Automata, Computability, and Complexity) [Familiarity]
4. Provide examples of classic NP-complete problems. [Familiarity]
5. Prove that a problem is NP-complete by reducing a classic known NP-complete problem to it. [Usage]

## AL/Advanced Automata Theory and Computability

## *[Elective]*

***Topics:***

- Sets and languages
    - Regular languages
    - Review of deterministic finite automata (DFAs)
    - Nondeterministic finite automata (NFAs)
    - Equivalence of DFAs and NFAs
    - Review of regular expressions; their equivalence to finite automata
    - Closure properties
    - Proving languages non-regular, via the pumping lemma or alternative means
- Context-free languages
    - Push-down automata (PDAs)
    - Relationship of PDAs and context-free grammars
    - Properties of context-free languages
- Turing machines, or an equivalent formal model of universal computation
- Nondeterministic Turing machines
- Chomsky hierarchy
- The Church-Turing thesis
- Computability
- Rice's Theorem
- Examples of uncomputable functions
- Implications of uncomputability

***Learning Outcomes:***

1. Determine a language's place in the Chomsky hierarchy (regular, context-free, recursively enumerable). [Assessment]
2. Prove that a language is in a specified class and that it is not in the next lower class. [Assessment]
3. Convert among equivalently powerful notations for a language, including among DFAs, NFAs, and regular expressions, and between PDAs and CFGs. [Usage]
4. Explain the Church-Turing thesis and its significance. [Familiarity]
5. Explain Rice's Theorem and its significance. [Familiarity]
6. Provide examples of uncomputable functions. [Familiarity]
7. Prove that a problem is uncomputable by reducing a classic known uncomputable problem to it. [Usage]

## AL/Advanced Data Structures Algorithms and Analysis

*[Elective]*

Many programs will want their students to have exposure to more advanced algorithms or methods of analysis. Below is a selection of possible advanced topics that are current and timely but by no means exhaustive.

***Topics:***

- Balanced trees (e.g., AVL trees, red-black trees, splay trees, treaps)
- Graphs (e.g., topological sort, finding strongly connected components, matching)
- Advanced data structures (e.g., B-trees, Fibonacci heaps)
- String-based data structures and algorithms (e.g., suffix arrays, suffix trees, tries)
- Network flows (e.g., max flow [Ford-Fulkerson algorithm], max flow – min cut, maximum bipartite matching)
- Linear Programming (e.g., duality, simplex method, interior point algorithms)
- Number-theoretic algorithms (e.g., modular arithmetic, primality testing, integer factorization)
- Geometric algorithms (e.g., points, line segments, polygons [properties, intersections], finding convex hull, spatial decomposition, collision detection, geometric search/proximity)
- Randomized algorithms
- Approximation algorithms
- Amortized analysis
- Probabilistic analysis
- Online algorithms and competitive analysis

***Learning Outcomes:***

1. Understand the mapping of real-world problems to algorithmic solutions (e.g., as graph problems, linear programs, etc.) [Usage, Assessment]
2. Use advanced algorithmic techniques (e.g., randomization, approximation) to solve real problems. [Usage]
3. Apply advanced analysis techniques (e.g., amortized, probabilistic, etc.) to algorithms. [Usage, Assessment]

# Architecture and Organization (AR)

Computing professionals should not regard the computer as just a black box that executes programs by magic. AR-Architecture and Organization builds on SF-Systems Fundamentals to develop a deeper understanding of the hardware environment upon which all of computing is based, and the interface it provides to higher software layers. Students should acquire an understanding and appreciation of a computer system's functional components, their characteristics, performance, and interactions, and, in particular, the challenge of harnessing parallelism to sustain performance improvements now and into the future. Students need to understand computer architecture to develop programs that can achieve high performance through a programmer's awareness of parallelism and latency. In selecting a system to use, students should to able to understand the tradeoff among various components, such as CPU clock speed, cycles per instruction, memory size, and average memory access time.

The learning outcomes specified for these topics correspond primarily to the core and are intended to support programs that elect to require only the minimum 16 hours of computer architecture of their students.  For programs that want to teach more than the minimum, the same AR topics can be treated at a more advanced level by implementing a two-course sequence.  For programs that want to cover the elective topics, those topics can be introduced within a two-course sequence and/or be treated in a more comprehensive way in a third course.

## AR. Architecture and Organization (0 Core-Tier 1 hours, 16 Core-Tier 2 hours)

| | Core-Tier 1 hours | Core-Tier 2 Hours | Includes Elective |
|---|---|---|---|
| AR/Digital logic and digital systems | | 3 | N |
| AR/Machine level representation of data | | 3 | N |
| AR/Assembly level machine organization | | 6 | N |
| AR/Memory system organization and architecture | | 3 | N |
| AR/Interfacing and communication | | 1 | N |
| AR/Functional organization | | | Y |
| AR/Multiprocessing and alternative architectures | | | Y |
| AR/Performance enhancements | | | Y |

# AR/Digital logic and digital systems

*[3 Core-Tier2 hours]*

***Topics***:

- Overview and history of computer architecture
- Combinational vs. sequential logic/Field programmable gate arrays as a fundamental combinational + sequential logic building block
- Multiple representations/layers of interpretation (hardware is just another layer)
- Computer-aided design tools that process hardware and architectural representations
- Register transfer notation/Hardware Description Language (Verilog/VHDL)
- Physical constraints (gate delays, fan-in, fan-out, energy/power)

***Learning outcomes***:

1. Describe the progression of computer technology components from vacuum tubes to VLSI, from mainframe computer architectures to the organization of warehouse-scale computers [Familiarity].
2. Comprehend the trend of modern computer architectures towards multi-core and that parallelism is inherent in all hardware systems [Familiarity].
3. Explain the implications of the “power wall” in terms of further processor performance improvements and the drive towards harnessing parallelism [Familiarity].
4. Articulate that there are many equivalent representations of computer functionality, including logical expressions and gates, and be able to use mathematical expressions to describe the functions of simple combinational and sequential circuits [Familiarity].
5. Design the basic building blocks of a computer: arithmetic-logic unit (gate-level), registers (gate-level), central processing unit (register transfer-level), memory (register transfer-level) [Usage].
6. Use CAD tools for capture, synthesis, and simulation to evaluate simple building blocks (e.g., arithmetic-logic unit, registers, movement between registers) of a simple computer design [Usage].

7. Evaluate the functional and timing diagram behavior of a simple processor implemented at the logic circuit level [Assessment].

## AR/Machine-level representation of data

### *[3 Core-Tier2 hours]*

***Topics***:

- Bits, bytes, and words
- Numeric data representation and number bases
- Fixed- and floating-point systems
- Signed and twos-complement representations
- Representation of non-numeric data (character codes, graphical data)
- Representation of records and arrays

***Learning outcomes***:

1. Explain why everything is data, including instructions, in computers [Familiarity].
2. Explain the reasons for using alternative formats to represent numerical data [Familiarity].
3. Describe how negative integers are stored in sign-magnitude and twos-complement representations [Familiarity].
4. Explain how fixed-length number representations affect accuracy and precision [Familiarity].
5. Describe the internal representation of non-numeric data, such as characters, strings, records, and arrays [Familiarity].
6. Convert numerical data from one format to another [Usage].
7. Write simple programs at the assembly/machine level for string processing and manipulation [Usage].

## AR/Assembly level machine organization

### *[6 Core-Tier2 hours]*

***Topics***:

- Basic organization of the von Neumann machine
- Control unit; instruction fetch, decode, and execution
- Instruction sets and types (data manipulation, control, I/O)
- Assembly/machine language programming
- Instruction formats
- Addressing modes
- Subroutine call and return mechanisms (xref PL/Language Translation and Execution)
- I/O and interrupts
- Heap vs. Static vs. Stack vs. Code segments
- Shared memory multiprocessors/multicore organization
- Introduction to SIMD vs. MIMD and the Flynn Taxonomy

***Learning outcomes*:**

1. Explain the organization of the classical von Neumann machine and its major functional units [Familiarity].
2. Describe how an instruction is executed in a classical von Neumann machine, with extensions for threads, multiprocessor synchronization, and SIMD execution [Familiarity].

3. Describe instruction level parallelism and hazards, and how they are managed in typical processor pipelines [Familiarity].
4. Summarize how instructions are represented at both the machine level and in the context of a symbolic assembler [Familiarity].
5. Demonstrate how to map between high-level language patterns into assembly/machine language notations [Familiarity].
6. Explain different instruction formats, such as addresses per instruction and variable length vs. fixed length formats [Familiarity].
7. Explain how subroutine calls are handled at the assembly level [Familiarity].
8. Explain the basic concepts of interrupts and I/O operations [Familiarity].
9. Write simple assembly language program segments [Usage].
10. Show how fundamental high-level programming constructs are implemented at the machine-language level [Usage].

## AR/Memory system organization and architecture

### *[3 Core-Tier2 hours]*

[Cross-reference OS/Memory Management--Virtual Machines]

***Topics***:

- Storage systems and their technology
- Memory hierarchy: importance of temporal and spatial locality
- Main memory organization and operations
- Latency, cycle time, bandwidth, and interleaving
- Cache memories (address mapping, block size, replacement and store policy)
- Multiprocessor cache consistency/Using the memory system for inter-core synchronization/atomic memory operations
- Virtual memory (page table, TLB)
- Fault handling and reliability
- Error coding, data compression, and data integrity (cross-reference SF/Reliability through Redundancy)

***Learning outcomes***:

1. Identify the main types of memory technology [Familiarity].
2. Explain the effect of memory latency on running time [Familiarity].
3. Describe how the use of memory hierarchy (cache, virtual memory) is used to reduce the effective memory latency [Familiarity].
4. Describe the principles of memory management [Familiarity].
5. Explain the workings of a system with virtual memory management [Familiarity].
6. Compute Average Memory Access Time under a variety of memory system configurations and workload assumptions [Usage].

## AR/Interfacing and communication

### *[1 Core-Tier2 hour]*

[Cross-reference OS Knowledge Area for a discussion of the operating system view of input/output processing and management. The focus here is on the hardware mechanisms for supporting device interfacing and processor-to-processor communications.]

***Topics***:

- I/O fundamentals: handshaking, buffering, programmed I/O, interrupt-driven I/O
- Interrupt structures: vectored and prioritized, interrupt acknowledgment
- External storage, physical organization, and drives
- Buses: bus protocols, arbitration, direct-memory access (DMA)
- Introduction to networks: networks as another layer of access hierarchy
- Multimedia support
- RAID architectures

***Learning outcomes*:**

1. Explain how interrupts are used to implement I/O control and data transfers [Familiarity].
2. Identify various types of buses in a computer system [Familiarity].
3. Describe data access from a magnetic disk drive [Familiarity].
4. Compare common network organizations, such as ethernet/bus, ring, switched vs. routed [Familiarity].
5. Identify interfaces needed for multimedia support, from storage, through network, to memory and display [Familiarity].
6. Describe the advantages and limitations of RAID architectures [Familiarity].

## AR/Functional organization

### *[Elective]*

[Note: elective for computer scientist; would be core for computer engineering curriculum]

***Topics***:

- Implementation of simple datapaths, including instruction pipelining, hazard detection and resolution
- Control unit: hardwired realization vs. microprogrammed realization
- Instruction pipelining
- Introduction to instruction-level parallelism (ILP)

***Learning outcomes***:

1. Compare alternative implementation of datapaths [Familiarity].
2. Discuss the concept of control points and the generation of control signals using hardwired or microprogrammed implementations [Familiarity].
3. Explain basic instruction level parallelism using pipelining and the major hazards that may occur [Familiarity].
4. Design and implement a complete processor, including datapath and control [Usage].
5. Determine, for a given processor and memory system implementation, the average cycles per instruction [Assessment].

## AR/Multiprocessing and alternative architectures

*[Elective]*

[Cross-reference PD/Parallel Architecture: The view here is on the hardware implementation of SIMD and MIMD architectures; in PD/Parallel Architecture, it is on the way that algorithms can be matched to the underlying hardware capabilities for these kinds of parallel processing architectures.]

***Topics***:

- Power Law
- Example SIMD and MIMD instruction sets and architectures
- Interconnection networks (hypercube, shuffle-exchange, mesh, crossbar)
- Shared multiprocessor memory systems and memory consistency
- Multiprocessor cache coherence

***Learning outcomes*:**

1. Discuss the concept of parallel processing beyond the classical von Neumann model [Familiarity].
2. Describe alternative architectures such as SIMD and MIMD [Familiarity].
3. Explain the concept of interconnection networks and characterize different approaches [Familiarity].
4. Discuss the special concerns that multiprocessing systems present with respect to memory management and describe how these are addressed [Familiarity].
5. Describe the differences between memory backplane, processor memory interconnect, and remote memory via networks [Familiarity].

## AR/Performance enhancements

*[Elective]*

***Topics***:

- Superscalar architecture
- Branch prediction, Speculative execution, Out-of-order execution
- Prefetching
- Vector processors and GPUs
- Hardware support for Multithreading
- Scalability
- Alternative architectures, such as VLIW/EPIC, and Accelerators and other kinds of Special-Purpose Processors

***Learning outcomes***:

1. Describe superscalar architectures and their advantages [Familiarity].
2. Explain the concept of branch prediction and its utility [Familiarity].
3. Characterize the costs and benefits of prefetching [Familiarity].
4. Explain speculative execution and identify the conditions that justify it [Familiarity].
5. Discuss the performance advantages that multithreading offered in an architecture along with the factors that make it difficult to derive maximum benefits from this approach [Familiarity].
6. Describe the relevance of scalability to performance [Familiarity].

## Computational Science (CN)

Computational Science is a field of applied computer science, that is, the application of computer science to solve problems across a range of disciplines. According to the book "Introduction to Computational Science", Shiflet & Shiflet offer the following definition: "the field of computational science combines computer simulation, scientific visualization, mathematical modeling, computer programming and data structures, networking, database design, symbolic computation, and high performance computing with various disciplines."Computer science, which largely focuses on the theory, design, and implementation of algorithms for manipulating data and information, can trace its roots to the earliest devices used to assist people in computation over four thousand years ago. Various systems were created and used to calculate astronomical positions. Ada Lovelace's programming achievement was intended to calculate Bernoulli numbers. In the late nineteenth century, mechanical calculators became available, and were immediately put to use by scientists. The needs of scientists and engineers for computation have long driven research and innovation in computing. As computers increase in their problem-solving power, computational science has grown in both breadth and importance. It is a discipline in its own right (President's Information Technology Advisory Committee, 2005, page 13) and is considered to be "one of the five college majors on the rise" (Fischer and Gleen, "5 College Majors on the Rise", The Chronicle of Higher Education, 2009.) An amazing assortment of sub-fields have arisen under the umbrella of Computational Science, including computational biology, computational chemistry, computational mechanics, computational archeology, computational finance, computational sociology and computational forensics.

Some fundamental concepts of computational science are germane to every computer scientist, and computational science topics are extremely valuable components of an undergraduate program in computer science. This area offers exposure to many valuable ideas and techniques, including precision of numerical representation, error analysis, numerical techniques, parallel architectures and algorithms, modeling and simulation, information visualization, software engineering, and optimization. At the same time, students who take courses in this area have an opportunity to apply these techniques in a wide range of application areas, such as: molecular and fluid dynamics, celestial mechanics, economics, biology, geology, medicine, and social network analysis. Many of the techniques used in these areas require advanced mathematics such as calculus, differential equations, and linear algebra. The descriptions here assume that students have acquired the needed mathematical background elsewhere.

In the computational science community, the terms *run*, *modify*, and *create* are often used to describe levels of understanding. This chapter follows the conventions of other chapters in this volume and uses the terms *familiarity*, *usage*, and *assessment*.

## CN. Computational Science (1 Core-Tier1 hours, 0 Core-Tier2 hours)

| | **Core-Tier1 hours** | **Core-Tier2 hours** | **Includes Electives** |
|---|---|---|---|
| **CN/Fundamentals** | ***1*** | | **N** |
| **CN/Modeling and Simulation** | | | **Y** |
| **CN/Processing** | | | **Y** |
| **CN/Interactive Visualization** | | | **Y** |
| **CN/Data, Information, and Knowledge** | | | **Y** |

## CN/Fundamentals

*[1 Core-Tier1 hours]*

Abstraction is a fundamental concept in computer science. A principal approach to computing is to abstract the real world, create a model that can be simulated on a machine. The roots of computer science can be traced to this approach, modeling things such as trajectories of artillery shells and the modeling cryptographic protocols, both of which pushed the development of early computing systems in the early and mid-1940's.

Modeling and simulation are essential topics for computational science. Any introduction to computational science would either include or presume an introduction to computing. Topics relevant to computational science include fundamental concepts in program construction (SDF/Fundamental Programming Concepts), algorithm design (SDF/Algorithms and Design), program testing (SDF/Development Methods), data representations (AR/Machine Representation of Data), and basic computer architecture (AR/Memory System Organization and Architecture). In addition, a general set of modeling and simulation techniques, data visualization methods, and software testing and evaluation mechanisms are also important CN fundamentals.

***Topics:***

- Models as abstractions of situations
- Simulations as dynamic modeling
- Simulation techniques and tools, such as physical simulations, human-in-the-loop guided simulations, and virtual reality.
- Foundational approaches to validating models

***Learning Outcomes:***

1. Explain the concept of modeling and the use of abstraction that allows the use of a machine to solve a problem. [Familiarity]
2. Describe the relationship between modeling and simulation, i.e., thinking of simulation as dynamic modeling. [Familiarity]
3. Create a simple, formal mathematical model of a real-world situation and use that model in a simulation. [Usage]
4. Differentiate among the different types of simulations, including physical simulations, human-guided simulations, and virtual reality. [Familiarity]
5. Describe several approaches to validating models. [Familiarity]

## CN/Modeling and Simulation

*[Elective]*

***Topics:***

- Purpose of modeling and simulation including optimization; supporting decision making, forecasting, safety considerations; for training and education.
- Tradeoffs including performance, accuracy, validity, and complexity.
- The simulation process; identification of key characteristics or behaviors, simplifying assumptions; validation of outcomes.
- Model building: use of mathematical formula or equation, graphs, constraints; methodologies and techniques; use of time stepping for dynamic systems.

- Formal models and modeling techniques: mathematical descriptions involving simplifying assumptions and avoiding detail. The descriptions use fundamental mathematical concepts such as set and function. Random numbers. Examples of techniques including:
    - Monte Carlo methods
    - Stochastic processes
    - Queuing theory
    - Petri nets and colored Petri nets
    - Graph structures such as directed graphs, trees, networks
    - Games, game theory, the modeling of things using game theory
    - Linear programming and its extensions
    - Dynamic programming
    - Differential equations: ODE, PDE
    - Non-linear techniques
    - State spaces and transitions
- Assessing and evaluating models and simulations in a variety of contexts; verification and validation of models and simulations.
- Important application areas including health care and diagnostics, economics and finance, city and urban planning, science, and engineering.
- Software in support of simulation and modeling; packages, languages.

***Learning Outcomes:***

1. Explain and give examples of the benefits of simulation and modeling in a range of important application areas. [Familiarity]
2. Demonstrate the ability to apply the techniques of modeling and simulation to a range of problem areas. [Usage]
3. Explain the constructs and concepts of a particular modeling approach. [Familiarity]
4. Explain the difference between validation and verification of a model; demonstrate the difference with specific examples[1]. [Assessment]
5. Verify and validate the results of a simulation. [Assessment]
6. Evaluate a simulation, highlighting the benefits and the drawbacks. [Assessment]
7. Choose an appropriate modeling approach for a given problem or situation. [Assessment]
8. Compare results from different simulations of the same situation and explain any differences. [Assessment]
9. Infer the behavior of a system from the results of a simulation of the system. [Assessment]
10. Extend or adapt an existing model to a new situation. [Assessment]

[1] *Verification* means that the computations of the model are correct. If we claim to compute total time, for example, the computation actually does that. *Validation* asks whether the model matches the real situation.

# CN/Processing

## *[Elective]*

The processing topic area includes numerous topics from other knowledge areas. Specifically, coverage of processing should include a discussion of hardware architectures, including parallel systems, memory hierarchies, and interconnections among processors. These are covered in AR/Interfacing and Communication, AR/Multiprocessing and Alternative Architectures, AR/Performance Enhancements.

***Topics:***

- Fundamental programming concepts:
  - The concept of an algorithm consisting of a finite number of well-defined steps, each of which completes in a finite amount of time, as does the entire process.
  - Examples of well-known algorithms such as sorting and searching.
  - The concept of analysis as understanding what the problem is really asking, how a problem can be approached using an algorithm, and how information is represented so that a machine can process it.
  - The development or identification of a workflow.
  - The process of converting an algorithm to machine-executable code.
  - Software processes including lifecycle models, requirements, design, implementation, verification and maintenance.
  - Machine representation of data computer arithmetic, and numerical methods, specifically sequential and parallel architectures and computations.
- Fundamental properties of parallel and distributed computation:
  - Bandwidth.
  - Latency.
  - Scalability.
  - Granularity.
  - Parallelism including task, data, and event parallelism.
  - Parallel architectures including processor architectures, memory and caching.
  - Parallel programming paradigms including threading, message passing, event driven techniques, parallel software architectures, and MapReduce.
  - Grid computing.
  - The impact of architecture on computational time.
  - Total time to science curve for parallelism: continuum of things.
- Computing costs, e.g., the cost of re-computing  a value vs. the cost of storing and lookup.

***Learning Outcomes:***

1. Explain the characteristics and defining properties of algorithms and how they relate to machine processing. [Familiarity]
2. Analyze simple problem statements to identify relevant information and select appropriate processing to solve the problem. [Assessment]
3. Identify or sketch a workflow for an existing computational process such as the creation of a graph based on experimental data. [Familiarity]
4. Describe the process of converting an algorithm to machine-executable code. [Familiarity]
5. Summarize the phases of software development and compare several common lifecycle models. [Familiarity]
6. Explain how data is represented in a machine. Compare representations of integers to floating point numbers. Describe underflow, overflow, round off, and truncation errors in data representations. [Familiarity]
7. Apply standard numerical algorithms to solve ODEs and PDEs. Use computing systems to solve systems of equations. [Usage]

8. Describe the basic properties of bandwidth, latency, scalability and granularity. [Familiarity]
9. Describe the levels of parallelism including task, data, and event parallelism. [Familiarity]
10. Compare and contrast parallel programming paradigms recognizing the strengths and weaknesses of each. [Assessment]
11. Identify the issues impacting correctness and efficiency of a computation. [Familiarity]
12. Design, code, test and debug programs for a parallel computation. [Usage]

## CN/Interactive Visualization

## *[Elective]*

This sub-area is related to modeling and simulation. Most topics are discussed in detail in other knowledge areas in this document. There are many ways to present data and information, including immersion, realism, variable perspectives; haptics and heads-up displays, sonification, and gesture mapping.

Interactive visualization in general requires understanding of human perception (GV/Basics); graphics pipelines, geometric representations and data structures (GV/Fundamental Concepts); 2D and 3D rendering, surface and volume rendering (GV/Rendering, GV/Modeling, and GV/Advanced Rendering); and the use of APIs for developing user interfaces using standard input components such as menus, sliders, and buttons; and standard output components for data display, including charts, graphs, tables, and histograms (HCI/GUI Construction, HCI/GUI Programming).

***Topics:***

- Principles of data visualization.
- Graphing and visualization algorithms.
- Image processing techniques.
- Scalability concerns.

***Learning Outcomes:***

1. Compare common computer interface mechanisms with respect to ease-of-use, learnability, and cost. [Assessment]
2. Use standard APIs and tools to create visual displays of data, including graphs, charts, tables, and histograms. [Usage]
3. Describe several approaches to using a computer as a means for interacting with and processing data. [Familiarity]
4. Extract useful information from a dataset. [Assessment]
5. Analyze and select visualization techniques for specific problems. [Assessment]
6. Describe issues related to scaling data analysis from small to large data sets. [Familiarity]

## CN/Data, Information, and Knowledge

## *[Elective]*

Many topics are discussed in detail in other knowledge areas in this document, specifically Information Management (IM/Information Management Concepts, IM/Database Systems, and IM/Data Modeling), Algorithms and Complexity (AL/Basic Analysis, AL/Fundamental Data Structures and Algorithms), and Software Development Fundamentals (SDF/Fundamental Programming Concepts, SDF/Development Methods).

***Topics:***

- Content management models, frameworks, systems, design methods (as in IM. Information Management).
- Digital representations of content including numbers, text, images (e.g., raster and vector), video (e.g., QuickTime, MPEG2, MPEG4), audio (e.g., written score, MIDI, sampled digitized sound track) and animations; complex/composite/aggregate objects; FRBR.
- Digital content creation/capture and preservation, including digitization, sampling, compression, conversion, transformation/translation, migration/emulation, crawling, harvesting.
- Content structure / management, including digital libraries and static/dynamic/stream aspects for:
    - Data: data structures, databases.
    - Information: document collections, multimedia pools, hyperbases (hypertext, hypermedia), catalogs, repositories.
    - Knowledge: ontologies, triple stores, semantic networks, rules.
- Processing and pattern recognition, including indexing, searching (including: queries and query languages; central / federated / P2P), retrieving, clustering, classifying/categorizing, analyzing/mining/extracting, rendering, reporting, handling transactions.
- User / society support for presentation and interaction, including browse, search, filter, route, visualize, share, collaborate, rate, annotate, personalize, recommend.
- Modeling, design, logical and physical implementation, using relevant systems/software.

***Learning Outcomes:***

1. Identify all of the data, information, and knowledge elements and related organizations, for a computational science application. [Assessment]
2. Describe how to represent data and information for processing. [Familiarity]
3. Describe typical user requirements regarding that data, information, and knowledge. [Familiarity]
4. Select a suitable system or software implementation to manage data, information, and knowledge. [Assessment]
5. List and describe the reports, transactions, and other processing needed for a computational science application. [Familiarity]
6. Compare and contrast database management, information retrieval, and digital library systems with regard to handling typical computational science applications. [Assessment]
7. Design a digital library for some computational science users / societies, with appropriate content and services. [Usage]

# Discrete Structures (DS)

Discrete structures are foundational material for computer science. By foundational we mean that relatively few computer scientists will be working primarily on discrete structures, but that many other areas of computer science require the ability to work with concepts from discrete structures. Discrete structures include important material from such areas as set theory, logic, graph theory, and probability theory.

The material in discrete structures is pervasive in the areas of data structures and algorithms but appears elsewhere in computer science as well. For example, an ability to create and understand a proof—either a formal symbolic proof or a less formal but still mathematically rigorous argument—is important in virtually every area of computer science, including (to name just a few) formal specification, verification, databases, and cryptography. Graph theory concepts are used in networks, operating systems, and compilers. Set theory concepts are used in software engineering and in databases. Probability theory is used in intelligent systems, networking, and a number of computing applications.

Given that discrete structures serves as a foundation for many other areas in computing, it is worth noting that the boundary between discrete structures and other areas, particularly Algorithms and Complexity, Software Development Fundamentals, Programming Languages, and Intelligent Systems, may not always be crisp. Indeed, different institutions may choose to organize the courses in which they cover this material in very different ways. Some institutions may cover these topics in one or two focused courses with titles like "discrete structures" or "discrete mathematics", whereas others may integrate these topics in courses on programming, algorithms, and/or artificial intelligence. Combinations of these approaches are also prevalent (e.g., covering many of these topics in a single focused introductory course and covering the remaining topics in more advanced topical courses).

## DS. Discrete Structures (37 Core-Tier1 hours, 4 Core-Tier2 hours)

| | Core-Tier1 hours | Core-Tier2 hours | Includes Electives |
|---|---|---|---|
| DS/Sets, Relations, and Functions | 4 | | N |
| DS/Basic Logic | 9 | | N |
| DS/Proof Techniques | 10 | 1 | N |
| DS/Basics of Counting | 5 | | N |
| DS/Graphs and Trees | 3 | 1 | N |
| DS/Discrete Probability | 6 | 2 | N |

# DS/Sets, Relations, and Functions

## *[4 Core-Tier1 hours]*

***Topics:***

[Core-Tier1]

- Sets
  - Venn diagrams
  - Union, intersection, complement
  - Cartesian product
  - Power sets
  - Cardinality of finite sets
- Relations
  - Reflexivity, symmetry, transitivity
  - Equivalence relations, partial orders
- Functions
  - Surjections, injections, bijections
  - Inverses
  - Composition

***Learning Outcomes:***

[Core-Tier1]

1. Explain with examples the basic terminology of functions, relations, and sets. [Familiarity]
2. Perform the operations associated with sets, functions, and relations. [Usage]
3. Relate practical examples to the appropriate set, function, or relation model, and interpret the associated operations and terminology in context. [Assessment]

## DS/Basic Logic

### *[9 Core-Tier1 hours]*

***Topics:***

[Core-Tier1]

- Propositional logic (cross-reference: Propositional logic is also reviewed in IS/Knowledge Based Reasoning)
- Logical connectives
- Truth tables
- Normal forms (conjunctive and disjunctive)
- Validity of well-formed formula
- Propositional inference rules (concepts of modus ponens and modus tollens)
- Predicate logic
  - Universal and existential quantification
- Limitations of propositional and predicate logic (e.g., expressiveness issues)

***Learning Outcomes:***

[Core-Tier1]

1. Convert logical statements from informal language to propositional and predicate logic expressions. [Usage]
2. Apply formal methods of symbolic propositional and predicate logic, such as calculating validity of formulae and computing normal forms. [Usage]
3. Use the rules of inference to construct proofs in propositional and predicate logic. [Usage]
4. Describe how symbolic logic can be used to model real-life situations or applications, including those arising in computing contexts such as software analysis (e.g., program correctness), database queries, and algorithms. [Usage]
5. Apply formal logic proofs and/or informal, but rigorous, logical reasoning to real problems, such as predicting the behavior of software or solving problems such as puzzles. [Usage]
6. Describe the strengths and limitations of propositional and predicate logic. [Familiarity]

## DS/Proof Techniques

### *[10 Core-Tier1 hours, 1 Core-Tier2 hour]*

***Topics:***

[Core-Tier1]

- Notions of implication, equivalence, converse, inverse, contrapositive, negation, and contradiction
- The structure of mathematical proofs
- Direct proofs
- Disproving by counterexample
- Proof by contradiction
- Induction over natural numbers
- Structural induction
- Weak and strong induction (i.e., First and Second Principle of Induction)
- Recursive mathematical definitions

[Core-Tier2]

- Well orderings

***Learning Outcomes:***

[Core-Tier1]

1. Identify the proof technique used in a given proof. [Familiarity]
2. Outline the basic structure of each proof technique described in this unit. [Usage]
3. Apply each of the proof techniques correctly in the construction of a sound argument. [Usage]
4. Determine which type of proof is best for a given problem. [Assessment]
5. Explain the parallels between ideas of mathematical and/or structural induction to recursion and recursively defined structures. [Assessment]
6. Explain the relationship between weak and strong induction and give examples of the appropriate use of each. [Assessment]

[Core-Tier2]

7. State the well-ordering principle and its relationship to mathematical induction. [Familiarity]

## DS/Basics of Counting

### *[5 Core-Tier1 hours]*

***Topics:***

[Core-Tier1]

- Counting arguments
  - Set cardinality and counting
  - Sum and product rule
  - Inclusion-exclusion principle
  - Arithmetic and geometric progressions
- The pigeonhole principle
- Permutations and combinations
  - Basic definitions
  - Pascal's identity
  - The binomial theorem
- Solving recurrence relations (cross-reference: AL/Basic Analysis)
  - An example of a simple recurrence relation, such as Fibonacci numbers
  - Other examples, showing a variety of solutions
- Basic modular arithmetic

***Learning Outcomes:***

[Core-Tier1]

1. Apply counting arguments, including sum and product rules, inclusion-exclusion principle and arithmetic/geometric progressions. [Usage]
2. Apply the pigeonhole principle in the context of a formal proof. [Usage]
3. Compute permutations and combinations of a set, and interpret the meaning in the context of the particular application. [Usage]

4. Map real-world applications to appropriate counting formalisms, such as determining the number of ways to arrange people around a table, subject to constraints on the seating arrangement, or the number of ways to determine certain hands in cards (e.g., a full house). [Usage]
5. Solve a variety of basic recurrence relations. [Usage]
6. Analyze a problem to determine underlying recurrence relations. [Usage]
7. Perform computations involving modular arithmetic. [Usage]

## DS/Graphs and Trees

### *[3 Core-Tier1 hours, 1 Core-Tier2 hour]*

(cross-reference: AL/Fundamental Data Structures and Algorithms, especially with relation to graph traversal strategies)

***Topics:***

[Core-Tier1]

- Trees
  - Properties
  - Traversal strategies
- Undirected graphs
- Directed graphs
- Weighted graphs

[Core-Tier2]

- Spanning trees/forests
- Graph isomorphism

***Learning Outcomes:***

[Core-Tier1]

1. Illustrate by example the basic terminology of graph theory, and some of the properties and special cases of each type of graph/tree. [Familiarity]
2. Demonstrate different traversal methods for trees and graphs, including pre, post, and in-order traversal of trees. [Usage]
3. Model *a variety of* real-world problems in computer science using appropriate forms of graphs and trees, such as representing a network topology or the organization of a hierarchical file system. [Usage]
4. Show how concepts from graphs and trees appear in data structures, algorithms, proof techniques (structural induction), and counting. [Usage]

[Core-Tier2]

5. Explain how to construct a spanning tree of a graph. [Usage]
6. Determine if two graphs are isomorphic. [Usage]

## DS/Discrete Probability

### *[6 Core-Tier1 hours, 2 Core-Tier2 hour]*

(Cross-reference IS/Basic Knowledge Representation and Reasoning, which includes a review of basic probability)

***Topics:***

[Core-Tier1]

- Finite probability space, events
- Axioms of probability and probability measures
- Conditional probability, Bayes' theorem
- Independence
- Integer random variables (Bernoulli, binomial)
- Expectation, including Linearity of Expectation
-

[Core-Tier2]

- Variance
- Conditional Independence

***Learning Outcomes:***

[Core-Tier1]

1. Calculate probabilities of events and expectations of random variables for elementary problems such as games of chance. [Usage]
2. Differentiate between dependent and independent events. [Usage]
3. Identify a case of the binomial distribution and compute a probability using that distribution. [Usage]
4. Make a probabilistic inference in a real-world problem using Bayes' theorem to determine the probability of a hypothesis given evidence. [Usage]
5. Apply the tools of probability to solve problems such as the average case analysis of algorithms or analyzing hashing. [Usage]

[Core-Tier2]

6. Compute the variance for a given probability distribution. [Usage]
7. Explain how events that are independent can be conditionally dependent (and vice-versa). Identify real-world examples of such cases. [Usage]

## Graphics and Visualization (GV)

*Computer graphics* is the term commonly used to describe the computer generation and manipulation of images. It is the science of enabling visual communication through computation. Its uses include cartoons, film special effects, video games, medical imaging, engineering, as well as scientific,   information, and knowledge visualization. Traditionally, graphics at the undergraduate level has focused on rendering, linear algebra, and phenomenological approaches. More recently, the focus has begun to include physics, numerical integration, scalability, and special-purpose hardware, In order for students to become adept at the use and generation of computer graphics, many implementation-specific issues must be addressed, such as file formats, hardware interfaces, and application program interfaces. These issues change rapidly, and the description that follows attempts to avoid being overly prescriptive about them. The area encompassed by Graphics and Visual Computing (GV) is divided into several interrelated fields:

- Fundamentals: Computer graphics depends on an understanding of how humans use vision to perceive information and how information can be rendered on a display device. Every computer scientist should have some understanding of where and how graphics can be appropriately applied and the fundamental processes involved in display rendering.
- Modeling: Information to be displayed must be encoded in computer memory in some form, often in the form of a mathematical specification of shape and form.
- Rendering: Rendering is the process of displaying the information contained in a model.
- Animation: Animation is the rendering in a manner that makes images appear to move and the synthesis or acquisition of the time variations of models.
- Visualization. The field of visualization seeks to determine and present underlying correlated structures and relationships in data sets from a wide variety of application areas. The prime objective of the presentation should be to communicate the information in a dataset so as to enhance understanding
- Computational Geometry: Computational Geometry is the study of algorithms that are stated in terms of geometry.

Graphics and Visualization is related to machine vision and image processing (in the Intelligent Systems KA) and algorithms such as computational geometry, which can be found in the Algorithms and Complexity KA. Topics in virtual reality can be found in the Human Computer Interaction KA.

This description assumes students are familiar with fundamental concepts of data representation, abstraction, and program implementation.

## GV. Graphics and Visualization (2 Core-Tier1 hours, 1 Core-Tier2 hours)

| | Core-Tier1 hours | Core-Tier2 hours | Includes Electives |
|---|---|---|---|
| GV/Fundamental Concepts | 2 | 1 | N |
| GV/Basic Rendering | | | Y |
| GV/Geometric Modeling | | | Y |
| GV/Advanced Rendering | | | Y |
| GV/Computer Animation | | | Y |
| GV/Visualization | | | Y |

## GV/Fundamental Concepts

### *[2 Core-Tier1 and 1 Core-Tier2 hours]*

For nearly every computer scientist and software developer, an understanding of how humans interact with machines is essential. While these topics may be covered in a standard undergraduate graphics course, they may also be covered in introductory computer science and programming courses. Part of our motivation for including immediate and retained modes is that these modes are analogous to polling vs. event driven programming. This is a fundamental question in computer science: Is there a button object, or is there just the display of a button on the screen? Note that most of the outcomes in this section are at the knowledge level, and many of these topics are revisited in greater depth in later sections.

***Topics:***

[Core-Tier1]

- Applications of computer graphics: including user interfaces, game engines, cad, visualization, virtual reality.
- Digitization of analog data and the limits of human perception, e.g., pixels for visual display, dots for laser printers, and samples for audio (HCI Foundations)
- Use of standard graphics APIs for the construction of UIs and display of standard image formats (see HCI GUI construction).
- Standard image formats, including lossless and lossy formats

[Core-Tier2]

- Additive and subtractive color models (CMYK and RGB) and why these provide a range of colors
- Tradeoffs between storing data and re-computing data as embodied by vector and raster representations of images
- Animation as a sequence of still images
- Double buffering.

***Learning Outcomes:***

[Core-Tier1]

1. Identify common uses of computer graphics. [Familiarity]
2. Explain in general terms how analog signals can be reasonably represented by discrete samples, for example, how images can be represented by pixels. [Familiarity]
3. Construct a simple user interface using a standard graphics API. [Usage]
4. Describe the differences between lossy and lossless image compression techniques, for example as reflected in common graphics image file formats such as JPG, PNG, and GIF. [Familiarity]

[Core-Tier2]

5. Describe color models and their use in graphics display devices. [Familiarity]
6. Describe the tradeoffs between storing information vs. storing enough information to reproduce the information, as in the difference between vector and raster rendering. [Familiarity]
7. Describe the basic process of producing continuous motion from a sequence of discrete frames (sometimes called “flicker fusion”). [Familiarity]
8. Describe how double-buffering can remove flicker from animation. [Familiarity]

## GV/Basic Rendering

## *[Elective]*

This section describes basic rendering and fundamental graphics techniques that nearly every undergraduate course in graphics will cover and that is essential for further study in graphics. Sampling and anti-aliasing is related to the effect of digitization and appears in other areas of computing, for example, in audio sampling.

***Topics:***

- Rendering in nature, i.e., the emission and scattering of light and its relation to numerical integration.
- Forward and backward rendering (i.e., ray-casting and rasterization).
- Polygonal representation.
- Basic radiometry, similar triangles, and projection model.
- Affine and coordinate system transformations.
- Ray tracing.
- Visibility and occlusion, including solutions to this problem such as depth buffering, Painter's algorithm, and ray tracing.
- The forward and backward rendering equation.
- Simple triangle rasterization.
- Rendering with a shader-based API.
- Texture mapping, including minification and magnification (e.g., trilinear MIP-mapping).
- Application of spatial data structures to rendering.
- Sampling and anti-aliasing.
- Scene graphs and the graphics pipeline.

***Learning Outcomes:***

1. Discuss the light transport problem and its relation to numerical integration i.e., light is emitted, scatters around the scene, and is measured by the eye; the form is an integral equation without analytic solution, but we can approach it as numerical integration. [Familiarity]
2. Describe the basic graphics pipeline and how forward and backward rendering factor in this. [Familiarity]
3. Model simple graphics images. [Usage]
4. Derive linear perspective from similar triangles by converting points (x, y, z) to points (x/z, y/z, 1). [Usage]
5. Obtain 2-dimensional and 3-dimensional points by applying affine transformations. [Usage]
6. Apply 3-dimensional coordinate system and the changes required to extend 2D transformation operations to handle transformations in 3D. [Usage]
7. Contrast forward and backward rendering. [Assessment]
8. Explain the concept and applications of texture mapping, sampling, and anti-aliasing. [Familiarity]
9. Explain the ray tracing – rasterization duality for the visibility problem. [Familiarity]
10. Implement simple procedures that perform transformation and clipping operations on simple 2-dimensional images. [Usage]
11. Implement a simple real-time renderer using a rasterization API (e.g., OpenGL) using vertex buffers and shaders. [Usage]
12. Compare and contrast the different rendering techniques. [Assessment]
13. Compute space requirements based on resolution and color coding. [Assessment]
14. Compute time requirements based on refresh rates, rasterization techniques. [Assessment]

# GV/Geometric Modeling

## *[Elective]*

***Topics:***

- Basic geometric operations such as intersection calculation and proximity tests
- Volumes, voxels, and point-based representations.
- Parametric polynomial curves and surfaces.
- Implicit representation of curves and surfaces.
- Approximation techniques such as polynomial curves, Bezier curves, spline curves and surfaces, and non-uniform rational basis (NURB) spines, and level set method.
- Surface representation techniques including tessellation, mesh representation, mesh fairing, and mesh generation techniques such as Delaunay triangulation, marching cubes, .
- Spatial subdivision techniques.
- Procedural models such as fractals, generative modeling, and L-systems.
- Graftals, cross referenced with programming languages (grammars to generated pictures).
- Elastically deformable and freeform deformable models.
- Subdivision surfaces.
- Multiresolution modeling.
- Reconstruction.
- Constructive Solid Geometry (CSG) representation.

***Learning Outcomes:***

1. Represent curves and surfaces using both implicit and parametric forms. [Usage]
2. Create simple polyhedral models by surface tessellation. [Usage]
3. Implement such algorithms as
4. Generate a mesh representation from an implicit surface. [Usage]
5. Generate a fractal model or terrain using a procedural method. [Usage]
6. Generate a mesh from data points acquired with a laser scanner. [Usage]
7. Construct CSG models from simple primitives, such as cubes and quadric surfaces. [Usage]
8. Contrast modeling approaches with respect to space and time complexity and quality of image. [Assessment]

# GV/Advanced Rendering

## *[Elective]*

***Topics:***

- Solutions and approximations to the rendering equation, for example:
    - Distribution ray tracing and path tracing
    - Photon mapping
    - Bidirectional path tracing
    - Reyes (micropolygon) rendering
    - Metropolis light transport
- Considering the dimensions of time (motion blur), lens position (focus), and continuous frequency (color).
- Shadow mapping.
- Occlusion culling.
- Bidirectional Scattering Distribution function (BSDF) theory and microfacets.
- Subsurface scattering.
- Area light sources.
- Hierarchical depth buffering.

- The Light Field, image-based rendering.
- Non-photorealistic rendering.
- GPU architecture.
- Human visual systems including adaptation to light, sensitivity to noise, and flicker fusion.

***Learning Outcomes:***

1. Demonstrate how an algorithm estimates a solution to the rendering equation. [Assessment]
2. Prove the properties of a rendering algorithm, e.g., complete, consistent, and/or unbiased. [Assessment]
3. Analyze the bandwidth and computation demands of a simple algorithm. [Assessment]
4. Implement a non-trivial shading algorithm (e.g., toon shading, cascaded shadow maps) under a rasterization API. [Usage]
5. Discuss how a particular artistic technique might be implemented in a renderer. [Familiarity]
6. Explain how to recognize the graphics techniques used to create a particular image. [Familiarity]
7. Implement any of the specified graphics techniques using a primitive graphics system at the individual pixel level. [Usage]
8. Implement a ray tracer for scenes using a simple (e.g., Phong's) BRDF plus reflection and refraction. [Usage]

## GV/Computer Animation

## *[Elective]*

***Topics:***

- Forward and inverse kinematics.
- Collision detection and response
- Procedural animation using noise, rules (boids/crowds), and particle systems.
- Skinning algorithms.
- Physics based motions including rigid body dynamics, physical particle systems, mass-spring networks for cloth and flesh and hair.
- Key-frame animation.
- Splines.
- Data structures for rotations, such as quaternions.
- Camera animation.
- Motion capture.

***Learning Outcomes:***

1. Compute the location and orientation of model parts using an forward kinematic approach. [Usage]
2. Compute the orientation of articulated parts of a model from a location and orientation using an inverse kinematic approach. [Usage]
3. Describe the tradeoffs in different representations of rotations. [Assessment]
4. Implement the spline interpolation method for producing in-between positions and orientations. [Usage]
5. Implement algorithms for physical modeling of particle dynamics using simple Newtonian mechanics, for example Witkin & Kass, snakes and worms, symplectic Euler, Stormer/Verlet, or midpoint Euler methods. [Usage]
6. Describe the tradeoffs in different approaches to ODE integration for particle modeling. [Assessment]
7. Discuss the basic ideas behind some methods for fluid dynamics for modeling ballistic trajectories, for example for splashes, dust, fire, or smoke. [Familiarity]
8. Use common animation software to construct simple organic forms using metaball and skeleton. [Usage]

## GV/Visualization

*[Elective]*

Visualization has strong ties to Human Computer Interaction as well as Computational Science. Readers should refer to the HCI and CN KAs for additional topics related to user population and interface evaluations.

***Topics:***

- Visualization of 2D/3D scalar fields: color mapping, isosurfaces.
- Direct volume data rendering: ray-casting, transfer functions, segmentation.
- Visualization of:
  - Vector fields and flow data
  - Time-varying data
  - High-dimensional data: dimension reduction, parallel coordinates,
  - Non-spatial data: multi-variate, tree/graph structured, text
- Perceptual and cognitive foundations that drive visual abstractions.
- Visualization design.
- Evaluation of visualization methods.
- Applications of visualization.

***Learning Outcomes:***

1. Describe the basic algorithms for scalar and vector visualization. [Familiarity]
2. Describe the tradeoffs of algorithms in terms of accuracy and performance. [Assessment]
3. Propose a suitable visualization design for a particular combination of data characteristics and application tasks. [Assessment]
4. Discuss the effectiveness of a given visualization for a particular task. [Assessment]
5. Design a process to evaluate the utility of a visualization algorithm or system. [Assessment]
6. Recognize a variety of applications of visualization including representations of scientific, medical, and mathematical data; flow visualization; and spatial analysis. [Familiarity]

# Human-Computer Interaction (HCI)

Human–computer interaction (HCI) is concerned with designing interactions between human activities and the computational systems that support them, with constructing interfaces to afford those interactions, and with the study of major phenomena surrounding them.

Interaction between users and computational artefacts occurs at an interface which includes both software and hardware. Thus interface design impacts the software life-cycle in that it should occur early; the design and implementation of core functionality can influence the user interface – for better or worse.

Because it deals with people as well as computational systems, as a knowledge area HCI demands the consideration of cultural, social, organizational, cognitive and perceptual issues. Consequently it draws on a variety of disciplinary traditions, including psychology, ergonomics, computer science, graphic and product design, anthropology and engineering.

## HCI: Human Computer Interaction (4 Core-Tier1 hours, 4 Core-Tier2 hours)

| | Core-Tier1 hours | Core-Tier2 hours | Includes Electives |
|---|---|---|---|
| HCI/Foundations | 4 | | N |
| HCI/Designing Interaction | | 4 | N |
| HCI/Programming Interactive Systems | | | Y |
| HCI/User-Centered Design & Testing | | | Y |
| HCI/New Interactive Technologies | | | Y |
| HCI/Collaboration & Communication | | | Y |
| HCI/Statistical Methods for HCI | | | Y |
| HCI/Human Factors & Security | | | Y |
| HCI/Design-Oriented HCI | | | Y |
| HCI/Mixed, Augmented and Virtual Reality | | | Y |

## HCI/Foundations

### *[4 Core-Tier1 hours]*

***Motivation:*** For end-users, the interface *is* the system. So design in this domain must be interaction-focused and human-centered. Students need a different repertoire of techniques to address this than is provided elsewhere in the curriculum.

***Topics***:

- Contexts for HCI (anything with a user interface: webpage, business applications, mobile applications, games, etc.)
- Processes for user-centered development: early focus on users, empirical testing, iterative design.
- Different measures for evaluation: utility, efficiency, learnability, user satisfaction.
- Usability heuristics and the principles of usability testing.
- Physical capabilities that inform interaction design: colour perception, ergonomics
- Cognitive models that inform interaction design: attention, perception and recognition, movement, and memory. Gulfs of expectation and execution.
- Social models that inform interaction design: culture, communication, networks and organizations.
- Principles of good design and good designers; engineering tradeoffs
- Accessibility: interfaces for differently-abled populations (e.g blind, motion-impaired)
- Interfaces for differently-aged population groups (e.g. children, 80+)

***Learning Outcomes:***

Students should be able to:

1. Discuss why human-centered software development is important [Familiarity]
2. Summarize the basic precepts of psychological and social interaction [Familiarity]
3. Develop and use a conceptual vocabulary for analyzing human interaction with software: affordance, conceptual model, feedback, and so forth [Usage]
4. Define a user-centered design process that explicitly recognizes that the user is not like the developer or her acquaintances [Usage]
5. Create and conduct a simple usability test for an existing software application [Assessment]

## HCI/Designing Interaction

### *[4 Core-Tier2 hours]*

***Motivation:*** CS students need a minimal set of well-established methods and tools to bring to interface construction.

***Topics:***

- Principles of graphical user interfaces (GUIs)
- Elements of visual design (layout, color, fonts, labeling)
- Task analysis, including qualitative aspects of generating task analytic models
- Low-fidelity (paper) prototyping
- Keystroke-level evaluation
- Help and documentation
- Handling human/system failure
- User interface standards

***Learning Outcomes:***

Students should be able to apply the principles of HCI foundations to:

1. Create a simple application, together with help and documentation, that supports a graphical user interface [Usage]
2. Conduct a quantitative evaluation and discuss/report the results [Usage]
3. Discuss at least one national or international user interface design standard [Assessment]

## HCI/Programming Interactive Systems

## *[Elective]*

***Motivation:*** To take a user-experience-centered view of software development and then cover approaches and technologies to make that happen.

***Topics:***

- Software Architecture Patterns: Model-View controller; command objects, online, offline, [*cross reference SE/Software Design*]
- Interaction Design Patterns: visual hierarchy, navigational distance
- Event management and user interaction
- Geometry management [cross reference GV/Geometric Modelling]
- Choosing interaction styles and interaction techniques
- Presenting information: navigation, representation, manipulation
- Interface animation techniques (scene graphs, etc)
- Widget classes and libraries
- Modern GUI libraries (e.g. iOS, Android, JavaFX) GUI builders and UI programming environments [*cross reference to PBD/Mobile Platforms*]
- Declarative Interface Specification: Stylesheets and DOMs
- Data-driven applications (database-backed web pages)
- Cross-platform design
- Design for resource-constrained devices (e.g. small, mobile devices)

***Learning Outcomes:***

Students should be able to apply the principles of HCI foundations to:

1. Understand there are common approaches to design problems, and be able to explain the importance of Model-View controller to interface programming [Familiarity]
2. Create an application with a modern graphical user interface [Usage]
3. Identify commonalities and differences in UIs across different platforms [Usage]
4. Explain and use GUI programming concepts: event handling, constraint-based layout management, etc [Assessment]

## HCI/User-Centered Design and Testing

## *[Elective]*

***Motivation:*** An exploration of techniques to ensure that end-users are fully considered at all stages of the design process, from inception to implementation.

***Topics:***

- Approaches to, and characteristics of, the design process
- Functionality and usability requirements [*cross reference to Software Engineering*]
- Techniques for gathering requirements: interviews, surveys, ethnographic & contextual enquiry [*cross reference SE-Software Engineering*]
- Techniques and tools for analysis & presentation of requirements: reports, personas
- Prototyping techniques and tools: sketching, storyboards, low-fidelity prototyping, wireframes
- Evaluation without users, using both qualitative and quantitative techniques: walkthroughs, GOMS, expert-based analysis, heuristics, guidelines, and standards
- Evaluation with users: observation, think-aloud, interview, survey, experiment.
- Challenges to effective evaluation: sampling, generalization.
- Reporting the results of evaluations
- Internationalisation, designing for users from other cultures, cross-cultural evaluation [*cross reference SE-Software Engineering*]

***Learning Outcomes:***

Students should be able to apply the principles of HCI foundations to:

1. Understand how user-centred design complements other software process models [Familiarity]
2. Use lo-fi (low fidelity) prototyping techniques to gather, and report, user responses [Usage]
3. Choose appropriate methods to support the development of a specific UI [Assessment]
4. Use a variety of techniques to evaluate a given UI [Assessment]
5. Describe the constraints and benefits of different evaluative methods [Assessment]

## HCI/New Interactive Technologies

## *[Elective]*

***Motivation:*** As technologies evolve, new interaction styles are made possible. This knowledge unit should be considered extensible, to track emergent technology.

***Topics:***

- Choosing interaction styles and interaction techniques
- Representing information to users: navigation, representation, manipulation
- Approaches to design, implementation and evaluation of non-mouse interaction
  - Touch and multi-touch interfaces
  - Shared, embodied, and large interfaces
  - New input modalities (such as sensor and location data)
  - New Windows (iPhone, Android)
  - Speech recognition and natural language processing [*cross reference IS-Intelligent Systems*]
  - Wearable and tangible interfaces
  - Persuasive interaction and emotion
  - Ubiquitous and context-aware (Ubicomp)

- Bayesian inference (e.g. predictive text, guided pointing)
- Ambient/peripheral display and interaction

***Learning Outcomes:***

Students should be able to apply the principles of HCI foundations to:

1. Describe when non-mouse interfaces are appropriate [Familiarity]
2. Understand the interaction possibilities beyond mouse-and-pointer interfaces [Usage]
3. Discuss the advantages (and disadvantages) of non-mouse interfaces [Assessment]

## HCI/Collaboration and Communication

## *[Elective]*

***Motivation:*** Computer interfaces not only support users in achieving their individual goals but also in their interaction with others, whether that is task-focussed (work or gaming) or task-unfocussed (social networking).

***Topics:***

- Asynchronous group communication: e-mail, forums, social networks (e.g., facebook)
- Synchronous group communication: chat rooms, conferencing, online games
- Social media, social computing, and social network analysis
- Online collaboration, 'smart' spaces, and social coordination aspects of workflow technologies
- Online communities
- Software characters and intelligent agents, virtual worlds and avatars (cross-reference IS/Agents)
- Social psychology

***Learning Outcomes:***

Students should be able to apply the principles of HCI foundations to:

1. Describe the difference between synchronous and asynchronous communication [Familiarity]
2. Compare the HCI issues in individual interaction with group interaction [Usage]
3. Discuss several issues of social concern raised by collaborative software [Assessment]
4. Discuss the HCI issues in software that embodies human intention [Assessment]

## HCI/Statistical Methods for HCI

## *[Elective]*

***Motivation:*** Much HCI work depends on the proper use, understanding and application of statistics. This knowledge is often held by students who join the field from psychology, but less common in students with a CS background.

***Topics:***

- t-tests
- ANOVA
- randomization (non-parametric) testing, within v. between-subjects design

- calculating effect size
- exploratory data analysis
- presenting statistical data
- using statistical data
- using qualitative and quantitative results together

***Learning Outcomes:***

Students should be able to apply the principles of HCI foundations to:

1. Explain basic statistical concepts and their areas of application [Familiarity]
2. Extract and articulate the statistical arguments used in papers which report [Usage]

## HCI/Human Factors and Security

## *[Elective]*

***Motivation:*** Effective interface design requires basic knowledge of security psychology. Many attacks do not have a technological basis, but exploit human propensities and vulnerabilities. "Only amateurs attack machines; professionals target people" (Bruce Schneier)

***Topics:***

- Applied psychology and security policies
- Security economics
- Regulatory environments – responsibility, liability and self-determination
- Organizational vulnerabilities and threats
- Usability design and security (cross reference to IAS-Information Assurance and Security)
- Pretext, impersonation and fraud. Phishing and spear phishing (cross reference to IAS-Information Assurance and Security)
- Trust, privacy and deception
- Biometric authentication (camera, voice)
- Identity management

***Learning Outcomes:***

Students should be able to apply the principles of HCI foundations to:

1. Explain the concepts of phishing and spear phishing, and how to recognize them [Familiarity]
2. Explain the concept of identity management and its importance [Familiarity]
3. Describe the issues of trust in interface design with an example of a high and low trust system [Usage]
4. Design a user interface for a security mechanism [Assessment]
5. Analyze a security policy and/or procedures to show where they consider, or fail to consider, human factors [Assessment]

## HCI/Design-Oriented HCI

### *[Elective]*

***Motivation:*** Some curricula will want to emphasize an understanding of the norms and values of HCI work itself as emerging from, and deployed within specific historical, disciplinary and cultural contexts.

***Topics:***

- Intellectual styles and perspectives to technology and its interfaces
- Consideration of HCI as a design discipline:
    - Sketching
    - Participatory design
- Critically reflective HCI
    - Critical technical practice
    - Technologies for political activism
    - Philosophy of user experience
    - Ethnography and ethnomethodology
- Indicative domains of application
    - Sustainability
    - Arts-informed computing

***Learning Objectives***

Students should be able to apply the principles of HCI foundations to:

1. Detail the processes of design appropriate to specific design orientations [Familiarity]
2. Apply a variety of design methods to a given problem [Usage]
3. Understand HCI as a design-oriented discipline. [Assessment]

## HCI/Mixed, Augmented and Virtual Reality

### *[Elective]*

***Motivation:*** A detailed consideration of the interface components required for the creation and development of immersive environments, especially games.

***Topics:***

- Output
    - Sound
    - Stereoscopic display
    - Force feedback simulation, haptic devices
- User input
    - Viewer and object tracking
    - Pose and gesture recognition
    - Accelerometers
    - Fiducial markers
    - User interface issues
- Physical modelling and rendering
    - Physical simulation: collision detection & response, animation
    - Visibility computation
    - Time-critical rendering, multiple levels of details (LOD)

- System architectures
    - Game engines
    - Mobile augmented reality
    - Flight simulators
    - CAVEs
    - Medical imaging
- Networking
    - p2p, client-server, dead reckoning, encryption, synchronization
    - Distributed collaboration

***Learning Objectives:***

1. Describe the optical model realized by a computer graphics system to synthesize stereoscopic view [Familiarity]
2. Describe the principles of different viewer tracking technologies [Familiarity]
3. Describe the differences between geometry- and image-based virtual reality [Familiarity]
4. Describe the issues of user action synchronization and data consistency in a networked environment [Familiarity]
5. Determine the basic requirements on interface, hardware, and software configurations of a VR system for a specified application [Usage]
6. To be aware of the range of possibilities for games engines, including their potential and their limitations [Assessment]

# Information Assurance and Security (IAS)

In CS2013, the Information Assurance and Security KA is added to the Body of Knowledge in recognition of the world's reliance on information technology and its critical role in computer science education. Information assurance and security as a domain is the set of controls and processes both technical and policy intended to protect and defend information and information systems by ensuring their availability, integrity, authentication, and confidentiality and providing for non-repudiation. The concept of assurance also carries an attestation that current and past processes and data are valid. Both assurance and security concepts are needed to ensure a complete perspective. Information assurance and security education, then, includes all efforts to prepare a workforce with the needed knowledge, skills, and abilities to protect our information systems and attest to the assurance of the past and current state of processes and data. The Information Assurance and Security KA is unique among the set of KA's presented here given the manner in which the topics are pervasive throughout other Knowledge Areas. The topics germane to only IAS are presented in depth in the IAS section; other topics are noted and cross referenced in the IAS KA, with the details presented in the KA in which they are tightly integrated.

The importance of security concepts and topics has emerged as a core requirement in the Computer Science discipline, much like the importance of performance concepts has been for many years. The emerging nature of security is reflected in the challenge of Computer Science programs to inculcate the security concepts in the breadth of courses given the past lack of attention. The table titled, "IAS. Information Assurance and Security (distributed)" reflects natural implied or specified topics with a strong role in security concepts and topics.

The importance of security concepts and topics has emerged as a core requirement in the Computer Science discipline, much like the importance of performance concepts has been for many years. The emerging nature of security is reflected in the challenge of Computer Science programs and faculty with security experience to inculcate the security concepts in the breadth of courses. The table titled, "IAS. Information Assurance and Security (distributed)" reflects topics with an implied or specified emphasis on security.

The IAS KA is shown in two groups; (1) concepts that are unique to Information Assurance and Security and (2) IAS topics that are integrated into other KA's. For completeness, the total distribution of hours is summarized in the table below.

| | Core-Tier1 hours | Core-Tier2 hours | Elective Topics |
|---|---|---|---|
| IAS | 3 | 6 | Y |
| IAS distributed in other KA's | 32 | 31.5 | Y |

## IAS. Information Assurance and Security (3 Core-Tier1 hours, 6 Core-Tier2 hours)

| | Core-Tier1 hours | Core-Tier2 hours | Includes Electives |
|---|---|---|---|
| IAS/Foundational Concepts in Security | 1 | | N |
| IAS/Principles of Secure Design | 1 | 1 | N |
| IAS/Defensive Programming | 1 | 1 | Y |
| IAS/Threats and Attacks | | 1 | N |
| IAS/Network Security | | 2 | Y |
| IAS/Cryptography | | 1 | N |
| IAS/Web Security | | | Y |
| IAS/Platform Security | | | Y |
| IAS/Security Policy and Governance | | | Y |
| IAS / Digital Forensics | | | Y |
| IAS/Secure Software Engineering | | | Y |

The following table shows the distribution of hours throughout all other KA's in CS2013 where security is appropriately addressed either as fundamental to the KU topics (for example, OS/Security or Protection or SE/Software Construction, etc.) or as a supportive use case for the topic (for example, HCI/Foundations or NC/Routing and Forwarding, etc). The hours represent the set of hours in that KA/KU where the topics are particularly relevant to Information Assurance and Security.

## IAS. Information Assurance and Security (distributed) (32 Core-Tier1 hours, 31.5 Core-Tier2 hours)

| Knowledge Area and Topic | Core-Tier1 hours | Core-Tier2 hours | Includes Electives |
|---|---|---|---|
| AR/Assembly level machine organization | | 1 | |
| AR/Memory system organization and architecture | | 0.5 | |
| AR/Multiprocessing and alternative architectures | | | Y |
| HCI/Foundations | 1 | | |
| HCI/Human Factors and Security | | | Y |
| IM/Information Management Concepts | 0.5 | 0.5 | |
| IM/Transaction Processing | | | Y |
| IM/Distributed Databases | | | Y |
| IS/Reasoning Under Uncertainty | | | Y |
| NC/Introduction | 1 | | |
| NC/Networked Applications | 0.5 | | |
| NC/Reliable Data Delivery | | 1.5 | |
| NC/Routing and Forwarding | | 1 | |
| NC/Local Area Networks | | 1 | |
| NC/Resource Allocation | | 0.5 | |

| | | | |
|---|---|---|---|
| NC/Mobility | | 1 | |
| OS/ Overview of OS | 2 | | |
| OS/OS Principles | 1 | | |
| OS/Concurrency | | 1.5 | |
| OS/Scheduling and Dispatch | | 2 | |
| OS/Memory Management | | 2 | |
| OS/Security and Protection | | 2 | |
| OS/Virtual Machines | | | Y |
| OS/Device Management | | | Y |
| OS/File Systems | | | Y |
| OS/Real Time and Embedded Systems | | | Y |
| OS/Fault Tolerance | | | Y |
| OS/System Performance Evaluation | | | Y |
| PBD/Web Platforms | | | Y |
| PBD/Mobile Platforms | | | Y |
| PBD/Industrial Platforms | | | Y |
| PD/Parallelism Fundamentals | 1 | | |
| PD/Parallel Decomposition | 0.5 | | |
| PD/Communication and Coordination | 1 | 1 | Y |
| PD/Parallel Architecture | 0.5 | | Y |
| PD/Distributed Systems | | | Y |
| PD/Cloud Computing | | | Y |
| PL/Object-Oriented Programming | 1 | 3 | |

| | | | |
|---|---|---|---|
| **PL/Functional Programming** | **1** | | |
| **PL/Basic Type Systems** | **0.5** | **2** | |
| **PL/Language Translation and Execution** | | **1** | |
| **PL/Runtime Systems** | | | **Y** |
| **PL/Static Analysis** | | | **Y** |
| **PL/Concurrency and Parallelism** | | | **Y** |
| **PL/Type Systems** | | | **Y** |
| **SDF/Fundamental Programming Concepts** | **1** | | |
| **SDF/Development Methods** | **8** | | |
| **SE/Software Processes** | **1** | | |
| **SE/Software Project Management** | | **1** | **Y** |
| **SE/Tools and Environments** | | **1** | |
| **SE/Software Construction** | | **2** | **Y** |
| **SE/Software Verification and Validation** | | **1** | **Y** |
| **SE/Software Evolution** | | **1.5** | |
| **SE/Software Reliability** | **1** | | |
| **SF/Cross-Layer Communications** | **3** | | |
| **SF/Parallelism** | **1** | | |
| **SF/Resource Allocation and Scheduling** | **0.5** | | |
| **SF/Virtualization and Isolation** | | **1** | |
| **SF/Reliability through Redundancy** | | **2** | |

| | | | |
|---|---|---|---|
| **SP/Social Context** | **0.5** | | |
| **SP/Analytical Tools** | **1** | | |
| **SP/Professional Ethics** | **1** | **0.5** | |
| **SP/Intellectual Property** | **2** | | **Y** |
| **SP/ Privacy and Civil Liberties** | **0.5** | | |
| **SP/Security Policies, Laws and Computer Crimes** | | | **Y** |

## IAS/Foundational Concepts in Security

*[1 Core-Tier1 hour]*

***Topics:***

[Core-Tier1]

- CIA (Confidentiality, Integrity, Availability)
- Concepts of risk, threats, vulnerabilities, and attack vectors (cross reference SE/Software Project Management/Risk)
- Authentication and authorization, access control (mandatory vs. discretionary)
- Concept of trust and trustworthiness
- Ethics (responsible disclosure) [cross-reference SP/Professional Ethics/Accountability, responsibility and liability]

***Learning outcomes:***

1. Understand the tradeoffs and balancing of key security properties (Confidentiality, Integrity, Availability) [Usage]
2. Understand the concepts of risk, threats, vulnerabilities and attack vectors (including the fact that there is no such thing as perfect security) [Familiarity]
3. Understand the concept of authentication, authorization, access control [Familiarity]
4. Understand the concept of trust and trustworthiness [Familiarity]
5. Be able to recognize that there are important ethical issues to consider in computer security, including ethical issues associated with fixing or not fixing vulnerabilities and disclosing or not disclosing vulnerabilities [Familiarity]

## IAS/Principles of Secure Design

### *[1 Core-Tier1 hour, 1 Core-Tier2 hour]*

***Topics:***

[Core-Tier1]

- Least privilege and isolation (cross-reference OS/Security and Protection/Policy/mechanism separation and SF/Virtualization and Isolation/Rationale for protection and predictable performance and PL/Language Translation and Execution/Memory management)
- Fail-safe defaults (cross-reference SE/Software Construction/ Coding practices: techniques, idioms/patterns, mechanisms for building quality programs and SDF/Development Methods/Programming correctness)
- Open design (cross-reference SE/Software Evolution/ Software development in the context of large, pre-existing code bases)
- End-to-end security (cross reference SF/Reliability through Redundancy/ How errors increase the longer the distance between the communicating entities; the end-to-end principle)
- Defense in depth
- Security by design (cross reference SE/Software Design/System design principles)
- Tensions between security and other design goals

[Core-Tier 2]

- Complete mediation
- Use of vetted security components
- Economy of mechanism (reducing trusted computing base, minimize attack surface) (cross reference SE/Software Design/System design principles and SE/Software Construction/Development context: "green field" vs. existing code base)
- Usable security (cross reference HCI/Foundations/Cognitive models that inform interaction design)
- Security composability
- Prevention, detection, and deterrence (cross reference SF/Reliability through Redundancy/Distinction between bugs and faults and NC/Reliable Data Delivery/Error control and NC/Reliable Data Delivery/Flow control)

***Learning outcomes:***

[Core-Tier1]

1. Describe the principle of least privilege and isolation and apply to system design [Usage]
2. Understand the principle of fail-safe and deny-by-default [Familiarity]
3. Understand not to rely on the secrecy of design for security (but also that open design alone does not imply security) [Familiarity]
4. Understand the goals of end-to-end data security [Familiarity]
5. Understand the benefits of having multiple layers of defenses [Familiarity]
6. Understand that security has to be a consideration from the point of initial design and throughout the lifecycle of a product [Familiarity]
7. Understanding that security imposes costs and tradeoffs [Familiarity]

[Core-Tier2]

8. Describe the concept of mediation and the principle of complete mediation [Usage]
9. Know to use standard components for security operations, instead of re-inventing fundamentals operations [Familiarity]

10. Understand the concept of trusted computing including trusted computing base and attack surface and the principle of minimizing trusted computing base [Usage]
11. Understand the importance of usability in security mechanism design [Familiarity]
12. Understand that security does not compose by default; security issues can arise at boundaries between multiple components [Familiarity]
13. Understand the different roles of prevention mechanisms and detection/deterrence mechanisms [Familiarity]

## IAS/Defensive Programming

### *[1 Core-Tier1 hour, 1 Core-Tier 2 hour]*

***Topics:***

[Core-Tier1]

- Input validation and data sanitization (cross reference SDF/Development Methods/Program Correctness)
- Choice of programming language and type-safe languages
- Examples of input validation and data sanitization errors (cross reference SDF/Development Methods/Program Correctness and SE/Software Construction/Coding Practices)
    - Buffer overflows
    - Integer errors
    - SQL injection
    - XSS vulnerability
- Race conditions (cross reference SF/Parallelism/Parallel programming and PD/Parallel Architecture/Shared vs. distributed memory and PD/Communication and Coordination/Shared Memory and PD/Parallelism Fundamentals/Programming errors not found in sequential programming)
- Correct handling of exceptions and unexpected behaviors (cross reference SDF/Development Methods/program correctness)

[Core-Tier2]

- Correctly generating randomness for security purposes
- Correct usage of third-party components (cross reference SDF/Development Methods/program correctness and Operating System Principles/Concepts of application program interfaces (APIs))
- Security updates (cross reference OS/Security and Protection/Security methods and devices)

[Electives]

- Information flow control
- Mechanisms for detecting and mitigating input and data sanitization errors
- Fuzzing
- Static analysis and dynamic analysis
- Program verification
- Operating system support (e.g., address space randomization, canaries)
- Hardware support (e.g., DEP, TPM)

***Learning outcomes***

[Core-Tier1]

1. Understand that an adversary controls the input channel and understand the importance of input validation and data sanitization [Familiarity]

2. Explain why you might choose to develop a program in a type-safe language like Java, in contrast to an unsafe programming language like C/C++ [Familiarity]
3. Understand common classes of input validation errors, and be able to write correct input validation code [Usage]
4. Demonstrate using a high-level programming language how to prevent a race condition from occurring and how to handle an exception [Usage]
5. Demonstrate the identification and graceful handling of error conditions. [Usage]

[Core-Tier2]

6. Understand the role of random numbers in security, beyond just cryptography (e.g., password generation, randomized algorithms to avoid algorithmic denial of service attacks) [Familiarity]
7. Understand the risks with misusing interfaces with third-party code and how to correctly use third-party code [Familiarity]
8. Understand the need for the ability to update software to fix security vulnerabilities [Familiarity]

[Elective]

9. Give examples of direct and indirect information flows [Familiarity]
10. Understand different types of mechanisms for detecting and mitigating data sanitization errors [Familiarity]
11. Demonstrate how programs are tested for input handling errors [Usage]
12. Use static and dynamic tools to identify programming faults [Usage]
13. Describe how memory architecture is used to protect runtime attacks [Familiarity]

## IAS/Threats and Attacks

### *[1 Core-Tier2 hour]*

***Topics:***

[Core-Tier2]

- Attacker goals, capabilities, and motivations (including underground economy, digital espionage, cyberwarfare, insider threats, hacktivism, advanced persistent threats)
- Malware: viruses, worms, spyware, botnets, Trojan horses, rootkits
- Social engineering (e.g., phishing, pharming, email spam, link spam) (cross reference SP/Social Context/Social implications of computing in a networked world and HCI/Designing Interaction/Handling human/system failure)
- Side channels, covert channels
- Denial of service and Distributed Denial of Service
- Attacks on privacy and anonymity (cross reference HCI/Foundations/Social models that inform interaction design: culture, communication, networks and organizations)

***Learning outcomes:***

[Core-Tier2]

1. Describe likely attacker types against a particular system [Usage]
2. Understand malware species and the virus and limitations of malware countermeasures (e.g., signature-based detection, behavioral detection) [Usage]
3. Identify instances of social engineering attacks and Denial of Service attacks [Familiarity]
4. Understand the concepts of side channels and covert channels and their differences [Familiarity]
5. Discuss the manner in which Denial of Service attacks can be identified and mitigated. [Familiarity]
6. Describe risks to privacy and anonymity in commonly used applications [Familiarity]

## IAS/Network Security

*[2 Core-Tier2 hours]*

Discussion of network security relies on previous understanding on fundamental concepts of networking, including protocols, such as TCP/IP, and network architecture/organization (cross-reference NC/Network Communication).

***Topics:***

[Core-Tier2]

- Network specific threats and attack types (e.g., denial of service, spoofing, sniffing and traffic redirection, man-in-the-middle, message integrity attacks, routing attacks, and traffic analysis)
- Use of cryptography for data and network security.
- Architectures for secure networks (e.g., secure channels, secure routing protocols, secure DNS, VPNs, anonymous communication protocols, isolation)
- Defense mechanisms and countermeasures (e.g., network monitoring, intrusion detection, firewalls, spoofing and DoS protection, honeypots, tracebacks)
- Security for wireless, cellular networks (cross reference (cross reference/Principles of cellular networks)
-

[Elective]

- Other non-wired networks (e.g., ad hoc, sensor, and vehicular networks)
- Censorship resistance
- Operational network security management (e.g., configure network access control)

***Learning outcomes:***

[Core-Tier2]

1. Describe the different categories of network threats and attacks [Familiarity]
2. Describe the architecture for public and private key cryptography and how PKI supports network security. [Usage]
3. Describe virtues and limitations of security technologies at each layer of the network stack [Familiarity]
4. Identify the appropriate defense mechanism(s) and its limitations given a network threat [Usage]
5. Understand security properties and limitations of other non-wired networks [Familiarity]

[Elective]

6. Understand the additional threats faced by non-wired networks [Familiarity]
7. Describe threats that can and cannot be protected against using secure communication channels [Familiarity]
8. Understand defenses against network censorship [Familiarity]
9. Configure a network for security [Usage]

## IAS/ Cryptography

*[1 Core-Tier2 hour]*

***Topics:***

[Core-Tier2]

- The Basic Cryptography Terminology covers notions pertaining to the different (communication) partners, secure/unsecure channel, attackers and their capabilities, encryption, decryption, keys and their characteristics, signatures, etc.
- Cipher types: Caesar cipher, affine cipher, etc. together with typical attack methods such as frequency analysis, etc.
- Overview of Mathematical Preliminaries where essential for Cryptography; includes topics in linear algebra, number theory, probability theory, and statistics.
- Public Key Infrastructure support for digital signature and encryption and its challenges.

[Elective]

- Cryptographic primitives:
  - pseudo-random generators and stream ciphers
  - block ciphers (pseudo-random permutations), e.g., AES
  - pseudo-random functions
  - hash functions, e.g., SHA2, collision resistance
  - message authentication codes
  - key derivations functions
- Symmetric key cryptography
  - Perfect secrecy and the one time pad
  - Modes of operation for semantic security and authenticated encryption (e.g., encrypt-then-MAC, OCB, GCM)
  - Message integrity (e.g., CMAC, HMAC)
- Public key cryptography:
  - Trapdoor permutation, e.g., RSA
  - Public key encryption, e.g., RSA encryption, EI Gamal encryption
  - Digital signatures
  - Public-key infrastructure (PKI) and certificates
  - Hardness assumptions, e.g., Diffie-Hellman, integer factoring
- Authenticated key exchange protocols, e.g., TLS
- Cryptographic protocols: challenge-response authentication, zero-knowledge protocols, commitment, oblivious transfer, secure 2-party or multi-party computation, secret sharing, and applications
- Motivate concepts using real-world applications, e.g., electronic cash, secure channels between clients and servers, secure electronic mail, entity authentication, device pairing, voting systems.
- Security definitions and attacks on cryptographic primitives:
  - Goals: indistinguishability, unforgeability, collision-resistance
  - Attacker capabilities: chosen-message attack (for signatures), birthday attacks, side channel attacks, fault injection attacks.
- Cryptographic standards and references implementations
- Quantum cryptography
- 

***Learning outcomes:***

[Core-Tier2]

1. Describe the purpose of Cryptography and list ways it is used in data communications. [Familiarity]
2. Define the following terms: Cipher, Cryptanalysis, Cryptographic Algorithm, and Cryptology and describe the two basic methods (ciphers) for transforming plain text in cipher text. [Familiarity]

3. Discuss the importance of prime numbers in cryptography and explain their use in cryptographic algorithms. [Familiarity]
4. Understand how to measure entropy and how to generate cryptographic randomness. [Usage]
5. Demonstrate how Public Key Infrastructure supports digital signing and encryption and discuss the limitations/vulnerabilities. [Usage]

[Elective]

6. Understand the cryptographic primitives and their basic properties. [Usage]
7. Students should be able to identify appropriate use of cryptography techniques for a given scenario. [Usage]
8. Understand public-key primitives and their applications. [Usage]
9. Understand how key exchange protocols work and how they fail. [Familiarity]
10. Understand cryptographic protocols and their properties. [Familiarity]
11. Describe real-world applications of cryptographic primitives and protocols. [Familiarity]
12. Understand precise security definitions, attacker capabilities and goals.[Familiarity]
13. Learn not to invent or implement your own cryptography [Usage]
14. Be aware of quantum cryptography and the impact of quantum computing on cryptographic algorithms. [Familiarity]

## IAS/Web Security

## *[Elective]*

***Topics:***

- Web security model
  - Browser security model including same-origin policy
  - Client-server trust boundaries, e.g., cannot rely on secure execution in the client
- Session management, authentication
  - Single sign-on
  - HTTPS and certificates
- Application vulnerabilities and defenses
  - SQL injection
  - XSS
  - CSRF
- Client-side security
  - Cookies security policy
  - HTTP security extensions, e.g. HSTS
  - Plugins, extensions, and web apps
  - Web user tracking
- Server-side security tools, e.g. Web Application Firewalls (WAFs) and fuzzers

***Learning outcomes:***

1. Understand the browser security model including same-origin policy and threat models in web security [Familiarity]
2. Understand the concept of web sessions, secure communication channels such as TLS and importance of secure certificates, authentication including single sign-on such as OAuth and SAML [Familiarity]
3. Understand common types of vulnerabilities and attacks in web applications and defenses against them. [Usage]
4. Understand how to use client-side security capabilities [Usage]
5. Understand how to use server-side security tools. [Usage]

## IAS/Platform Security

## *[Elective]*

***Topics:***

- Code integrity and code signing
- Secure boot, measured boot, and root of trust
- Attestation
- TPM and secure co-processors
- Security threats from peripherals, e.g., DMA, IOMMU
- Physical attacks: hardware Trojans, memory probes, cold boot attacks
- Security of embedded devices, e.g., medical devices, cars
- Trusted path

***Learning outcomes:***

1. Understand the concept of code integrity and code signing and the scope it applies to [Familiarity]
2. Understand the concept of root of trust and the process of secure boot and secure loading [Familiarity]
3. Understand the mechanism of remote attestation of system integrity [Familiarity]
4. Understand the goals and key primitives of TPM [Familiarity]
5. Understand the threats of plugging peripherals into a device [Familiarity]
6. Understand the physical attacks and countermeasures [Familiarity]
7. Understand attacks on non-PC hardware platforms [Familiarity]
8. Understand the concept and importance of trusted path [Familiarity]

## IAS/Security Policy and Governance

## *[Elective]*

***Topics:***

- Privacy policy
- Inference controls/statistical disclosure limitation
- Backup policy, password refresh policy
- Breach disclosure policy
- Data collection and retention policies
- Supply chain policy
- Cloud security: tradeoffs

***Learning outcomes:***

1. Describe the concept of privacy including personally private information, potential violations of privacy due to security mechanisms, and describe how privacy protection mechanisms run in conflict with security mechanisms [Familiarity]
2. Give an example of how an attacker can infer a secret by interacting with a database [Familiarity]
3. Understand how to set a data backup policy or password refresh policy [Familiarity]
4. Understand how to set a breach disclosure policy [Familiarity]
5. Understand the consequences of data retention policies [Familiarity]
6. Understand the risks of relying on outsourced manufacturing [Familiarity]
7. Understand the risks and benefits of outsourcing to the cloud [Familiarity]

## IAS/ Digital Forensics

### [Elective]

***Topics:***

- Basic Principles and methodologies for digital forensics.
- Design support for forensics
- Rules of Evidence – general concepts and differences between jurisdictions and Chain of Custody.
- Search and Seizure of evidence, e.g., computers, including search warrant issues.
- Digital Evidence methods and standards.
- Techniques and standards for Preservation of Data.
- Legal and Reporting Issues including working as an expert witness
- OS/File System Forensics
- Application Forensics
- Web Forensics
- Network Forensics
- Mobile Device Forensics
- Computer/network/system attacks
- Attack detection and investigation
- Anti-forensics

***Learning outcomes:***

1. Describe what is a Digital Investigation is, the sources of digital evidence, and the limitations of forensics. [Familiarity]
2. Understand how to design software to support forensics [Familiarity]
3. Describe the legal requirements for use if seized data. [Familiarity]
4. Describe the process of evidence seizure from the time when the requirement was identified to the disposition of the data. [Familiarity]
5. Describe how data collection is accomplished and the proper storage of the original and forensics copy. [Familiarity]
6. Conduct a data collection on a hard drive. [Usage]
7. Describe a person's responsibility and liability while testifying as a forensics examiner. [Familiarity]
8. Describe the file system structure for a given device (NTFS, MFS, iNode, HFS…) and recover data based on a given search term from an imaged system. [Usage]
9. Reconstruct application history from application artifacts [Usage]
10. Reconstruct web browsing history from web artifacts [Usage]
11. Capture and interpret network traffic [Usage]
12. Discuss the challenges associated with mobile device forensics. [Familiarity]
13. Evaluate a system (network, computer, or application) for the presence of malware or malicious activity. [Usage]
14. Apply forensics tools to investigate security breaches [Usage]
15. Defeat forensics tools [Usage]

## IAS/Secure Software Engineering

### [Elective]

Fundamentals of secure coding practices covered in other knowledge areas, including SDF/SE. SE/Software Construction; Software Verification and Validation

***Topics:***

- Building security into the Software Development Lifecycle (cross-reference SE/ Software Processes)
- Secure Design Principles and Patterns (Saltzer and Schroeder, etc)

- Secure Software Specification and Requirements deals with specifying what the program should and should not do, which can be done either using a requirements document or using formal methods.
- Secure Coding techniques to minimize vulnerabilities in code, such as data validation, memory handling, and crypto implementation (cross-reference SE/Software Construction)
- Secure Testing is the process of testing that security requirements are met (including Static and Dynamic analysis).
- Software Quality Assurance and benchmarking measurements

***Learning outcomes:***

1. Describe the requirements for integrating security into the SDL. [Familiarity]
2. Apply the concepts of the Design Principles for Protection Mechanisms (e.g. Saltzer and Schroeder ), the Principles for Software Security (Viega and McGraw), and the Principles for Secure Design (Morrie Gasser) on a software development project [Usage]
3. Develop specifications for a software development effort that fully specify functional requirements and identifies the expected execution paths. [Usage]
4. Describe software development best practices for minimizing vulnerabilities in programming code. [Familiarity]
5. Conduct a security verification and assessment (static and dynamic) of a software application [Usage]

# Information Management (IM)

Information Management (IM) is primarily concerned with the capture, digitization, representation, organization, transformation, and presentation of information; algorithms for efficient and effective access and updating of stored information, data modeling and abstraction, and physical file storage techniques. The student needs to be able to develop conceptual and physical data models, determine what IM methods and techniques are appropriate for a given problem, and be able to select and implement an appropriate IM solution that addresses relevant design concerns including scalability, accessibility and usability.

We also note that IM is related to fundamental information security concepts that are described in the Information Assurance and Security (IAS) topic area, *IAS/Fundamental Concepts*.

## IM. Information Management (1 Core-Tier1 hour; 9 Core-Tier2 hours)

| | Core-Tier1 hours | Core-Tier2 hours | Includes Electives |
|---|---|---|---|
| IM/Information Management Concepts | 1 | 2 | N |
| IM/Database Systems | | 3 | Y |
| IM/Data Modeling | | 4 | N |
| IM/Indexing | | | Y |
| IM/Relational Databases | | | Y |
| IM/Query Languages | | | Y |
| IM/Transaction Processing | | | Y |
| IM/Distributed Databases | | | Y |
| IM/Physical Database Design | | | Y |
| IM/Data Mining | | | Y |
| IM/Information Storage And Retrieval | | | Y |
| IM/MultiMedia Systems | | | Y |

## IM. Information Management-related topics (distributed) (1 Core-Tier1 hour, 2 Core-Tier2 hours)

| | Core-Tier1 hours | Core-Tier2 hours | Includes Electives |
|---|---|---|---|
| IAS/Fundamental Concepts* | 1 | 2 | N |

* See Information Assurance and Security Knowledge Area for a description of this topic area.

## IM/Information Management Concepts

### *[1 Core-Tier1 hour; 2 Core-Tier2 hours]*

***Topics:***

[Core-Tier1]

- Information systems as socio-technical systems
- Basic information storage and retrieval (IS&R) concepts
- Information capture and representation
- Supporting human needs: Searching, retrieving, linking, browsing, navigating
-

[Core-Tier2]

- Information management applications
- Declarative and navigational queries, use of links
- Analysis and indexing
- Quality issues: Reliability, scalability, efficiency, and effectiveness

***Learning Outcomes:***

[Core-Tier1]

1. Describe how humans gain access to information and data to support their needs [Familiarity]
2. Understand advantages and disadvantages of central organizational control over data [Assessment]
3. Identify the careers/roles associated with information management (e.g., database administrator, data modeler, application developer, end-user) [Familiarity]
4. Compare and contrast information with data and knowledge [Assessment]
5. Demonstrate uses of explicitly stored metadata/schema associated with data [Usage]
6. Identify issues of data persistence for an organization [Familiarity]

[Core-Tier2]

7. Critique/defend a small- to medium-size information application with regard to its satisfying real user information needs [Assessment]
8. Explain uses of declarative queries [Familiarity]
9. Give a declarative version for a navigational query [Familiarity]
10. Describe several technical solutions to the problems related to information privacy, integrity, security, and preservation [Familiarity]
11. Explain measures of efficiency (throughput, response time) and effectiveness (recall, precision) [Familiarity]
12. Describe approaches that scale up to globally networked systems [Familiarity]
13. Identify vulnerabilities and failure scenarios in common forms of information systems [Usage]

## IM/Database Systems

### *[3 Core-Tier2 hours]*

***Topics:***

[Core-Tier2]

- Approaches to and evolution of database systems
- Components of database systems

- Design of core DBMS functions (e.g., query mechanisms, transaction management, buffer management, access methods)
- Database architecture and data independence
- Use of a declarative query language
- Systems supporting structured and/or stream content

[Elective]

- Approaches for managing large volumes of data (e.g., noSQL database systems, use of MapReduce).

***Learning Outcomes:***

[Core-Tier2]

1. Explain the characteristics that distinguish the database approach from the traditional approach of programming with data files [Familiarity]
2. Understand the most common designs for core database system components including the query optimizer, query executor, storage manager, access methods, and transaction processor. [Familiarity]
3. Cite the basic goals, functions, models, components, applications, and social impact of database systems [Familiarity]
4. Describe the components of a database system and give examples of their use [Familiarity]
5. Identify major DBMS functions and describe their role in a database system [Familiarity]
6. Explain the concept of data independence and its importance in a database system [Familiarity]
7. Use a declarative query language to elicit information from a database [Usage]
8. Describe how various types of content cover the notions of structure and/or of stream (sequence), e.g., documents, multimedia, tables [Familiarity]

[Elective]

9. Describe major approaches to storing and processing large volumes of data [Familiarity]

## IM/Data Modeling

## *[4 Core-Tier2 hours]*

***Topics:***

[Core-Tier2]

- Data modeling
- Conceptual models (e.g., entity-relationship, UML diagrams)
- Spreadsheet models
- Relational data models
- Object-oriented models
- Semi-structured data model (expressed using DTD or XML Schema, for example)

***Learning Outcomes:***

[Core-Tier2]

1. Categorize data models based on the types of concepts that they provide to describe the database structure and their usage, for example, use of conceptual, spreadsheet, physical, and representational data models [Assessment]
2. Describe the modeling concepts and notation of widely used modeling notation (e.g., ERD notation, and UML), including their use in data modeling [Familiarity]
3. Define the fundamental terminology used in the relational data model [Familiarity]
4. Describe the basic principles of the relational data model [Familiarity]
5. Apply the modeling concepts and notation of the relational data model [Usage]
6. Describe the main concepts of the OO model such as object identity, type constructors, encapsulation, inheritance, polymorphism, and versioning [Familiarity]
7. Describe the differences between relational and semi-structured data models [Assessment]
8. Give a semi-structured equivalent (e.g., in DTD or XML Schema) for a given relational schema [Usage]

## IM/Indexing

## *[Elective]*

***Topics:***

- The impact of indices on query performance
- The basic structure of an index
- Keeping a buffer of data in memory
- Creating indexes with SQL
- Indexing text
- Indexing the web (how search engines work)

***Learning Outcomes:***

1. Generate an index file for a collection of resources [Usage]
2. Explain the role of an inverted index in locating a document in a collection [Familiarity]
3. Explain how stemming and stop words affect indexing [Familiarity]
4. Identify appropriate indices for given relational schema and query set [Usage]
5. Estimate time to retrieve information, when indices are used compared to when they are not used [Usage]

## IM/Relational Databases

## *[Elective]*

***Topics:***

Elective

- Mapping conceptual schema to a relational schema
- Entity and referential integrity
- Relational algebra and relational calculus
- Relational Database design
- Functional dependency
- Decomposition of a schema; lossless-join and dependency-preservation properties of a decomposition

- Candidate keys, superkeys, and closure of a set of attributes
- Normal forms (BCNF)
- Multi-valued dependency (4NF)
- Join dependency (PJNF, 5NF)
- Representation theory

***Learning Outcomes:***

1. Prepare a relational schema from a conceptual model developed using the entity- relationship model [Usage]
2. Explain and demonstrate the concepts of entity integrity constraint and referential integrity constraint (including definition of the concept of a foreign key) [Usage]
3. Demonstrate use of the relational algebra operations from mathematical set theory (union, intersection, difference, and Cartesian product) and the relational algebra operations developed specifically for relational databases (select (restrict), project, join, and division) [Usage]
4. Demonstrate queries in the relational algebra [Usage]
5. Demonstrate queries in the tuple relational calculus [Usage]
6. Determine the functional dependency between two or more attributes that are a subset of a relation [Assessment]
7. Connect constraints expressed as primary key and foreign key, with functional dependencies [Usage]
8. Compute the closure of a set of attributes under given functional dependencies [Usage]
9. Determine whether or not a set of attributes form a superkey and/or candidate key for a relation with given functional dependencies [Assessment]
10. Evaluate a proposed decomposition, to say whether or not it has lossless-join and dependency-preservation [Assessment]
11. Describe what is meant by BCNF, PJNF, 5NF [Familiarity]
12. Explain the impact of normalization on the efficiency of database operations especially query optimization [Familiarity]
13. Describe what is a multi-valued dependency and what type of constraints it specifies [Familiarity]

## IM/Query Languages

## *[Elective]*

***Topics:***

- Overview of database languages
- SQL (data definition, query formulation, update sublanguage, constraints, integrity)
- Selections
- Projections
- Select-project-join
- Aggregates and  group-by
- Subqueries
- QBE and 4th-generation environments
- Different ways to invoke non-procedural queries in conventional languages
- Introduction to other major query languages (e.g., XPATH, SPARQL)
- Stored procedures

***Learning Outcomes:***

1. Create a relational database schema in SQL that incorporates key, entity integrity, and referential integrity constraints [Usage]
2. Demonstrate data definition in SQL and retrieving information from a database using the SQL SELECT statement [Usage]
3. Evaluate a set of query processing strategies and select the optimal strategy [Assessment]
4. Create a non-procedural query by filling in templates of relations to construct an example of the desired query result [Usage]
5. Embed object-oriented queries into a stand-alone language such as C++ or Java (e.g., SELECT Col.Method() FROM Object) [Usage]
6. Write a stored procedure that deals with parameters and has some control flow, to provide a given functionality [Usage]

## IM/Transaction Processing

## *[Elective]*

***Topics:***

- Transactions
- Failure and recovery
- Concurrency control
- Interaction of transaction management with storage, especially buffering

***Learning Outcomes:***

1. Create a transaction by embedding SQL into an application program [Usage]
2. Explain the concept of implicit commits [Familiarity]
3. Describe the issues specific to efficient transaction execution [Familiarity]
4. Explain when and why rollback is needed and how logging assures proper rollback [Assessment]
5. Explain the effect of different isolation levels on the concurrency control mechanisms [Assessment]
6. Choose the proper isolation level for implementing a specified transaction protocol [Assessment]
7. Identify appropriate transaction boundaries in application programs [Assessment]

## IM/Distributed Databases

## *[Elective]*

***Topics:***

- Distributed DBMS
  - Distributed data storage
  - Distributed query processing
  - Distributed transaction model
  - Homogeneous and heterogeneous solutions
  - Client-server distributed databases (cross-reference SF/Computational Paradigms)
- Parallel DBMS
  - Parallel DBMS architectures: shared memory, shared disk, shared nothing;
  - Speedup and scale-up, e.g., use of the MapReduce processing model (cross-reference CN/Processing, PD/Parallel Decomposition)
  - Data replication and weak consistency models

***Learning Outcomes:***

1. Explain the techniques used for data fragmentation, replication, and allocation during the distributed database design process [Familiarity]
2. Evaluate simple strategies for executing a distributed query to select the strategy that minimizes the amount of data transfer [Assessment]
3. Explain how the two-phase commit protocol is used to deal with committing a transaction that accesses databases stored on multiple nodes [Familiarity]
4. Describe distributed concurrency control based on the distinguished copy techniques and the voting method [Familiarity]
5. Describe the three levels of software in the client-server model [Familiarity]

## IM/Physical Database Design

## *[Elective]*

***Topics:***

- Storage and file structure
- Indexed files
- Hashed files
- Signature files
- B-trees
- Files with dense index
- Files with variable length records
- Database efficiency and tuning

***Learning Outcomes:***

1. Explain the concepts of records, record types, and files, as well as the different techniques for placing file records on disk [Familiarity]
2. Give examples of the application of primary, secondary, and clustering indexes [Familiarity]
3. Distinguish between a non-dense index and a dense index [Assessment]
4. Implement dynamic multilevel indexes using B-trees [Usage]
5. Explain the theory and application of internal and external hashing techniques [Familiarity]
6. Use hashing to facilitate dynamic file expansion [Usage]
7. Describe the relationships among hashing, compression, and efficient database searches [Familiarity]
8. Evaluate costs and benefits of various hashing schemes [Assessment]
9. Explain how physical database design affects database transaction efficiency [Familiarity]

## IM/Data Mining

## *[Elective]*

***Topics:***

- The usefulness of data mining
- Data mining algorithms
- Associative and sequential patterns
- Data clustering
- Market basket analysis
- Data cleaning
- Data visualization

***Learning Outcomes:***

1. Compare and contrast different conceptions of data mining as evidenced in both research and application [Assessment]
2. Explain the role of finding associations in commercial market basket data [Familiarity]
3. Characterize the kinds of patterns that can be discovered by association rule mining [Assessment]
4. Describe how to extend a relational system to find patterns using association rules [Familiarity]
5. Evaluate methodological issues underlying the effective application of data mining [Assessment]
6. Identify and characterize sources of noise, redundancy, and outliers in presented data [Assessment]
7. Identify mechanisms (on-line aggregation, anytime behavior, interactive visualization) to close the loop in the data mining process [Familiarity]
8. Describe why the various close-the-loop processes improve the effectiveness of data mining [Familiarity]

## IM/Information Storage and Retrieval

## *[Elective]*

***Topics:***

- Characters, strings, coding, text
- Documents, electronic publishing, markup, and markup languages
- Tries, inverted files, PAT trees, signature files, indexing
- Morphological analysis, stemming, phrases, stop lists
- Term frequency distributions, uncertainty, fuzziness, weighting
- Vector space, probabilistic, logical, and advanced models
- Information needs, relevance, evaluation, effectiveness
- Thesauri, ontologies, classification and categorization, metadata
- Bibliographic information, bibliometrics, citations
- Routing and (community) filtering
- Search and search strategy, multimedia search, information seeking behavior, user modeling, feedback
- Information summarization and visualization
- Integration of citation, keyword, classification scheme, and other terms
- Protocols and systems (including Z39.50, OPACs, WWW engines, research systems)
- Digital libraries
- Digitization, storage, interchange, digital objects, composites, and packages
- Metadata, cataloging, author submission
- Naming, repositories, archives
- Spaces (conceptual, geographical, 2/3D, VR)
- Architectures (agents, buses, wrappers/mediators), interoperability
- Services (searching, linking, browsing, and so forth)
- Intellectual property rights management, privacy, and protection (watermarking)
- Archiving and preservation, integrity

***Learning Outcomes:***

1. Explain basic information storage and retrieval concepts [Familiarity]
2. Describe what issues are specific to efficient information retrieval [Familiarity]
3. Give applications of alternative search strategies and explain why the particular search strategy is appropriate for the application [Assessment]
4. Perform Internet-based research [Usage]
5. Design and implement a small to medium size information storage and retrieval system, or digital library [Usage]
6. Describe some of the technical solutions to the problems related to archiving and preserving information in a digital library [Familiarity]

## IM/Multimedia Systems

### *[Elective]*

***Topics:***

- Input and output devices (scanners, digital camera, touch-screens, voice-activated, MIDI keyboards, synthesizers), device drivers, control signals and protocols, DSPs
- Standards (audio, music, graphics, image, telephony, video, TV), including storage standards (Magneto Optical disk, CD-ROM, DVD)
- Applications, media editors, authoring systems, and authoring
- Streams/structures, capture/represent/transform, spaces/domains, compression/coding
- Content-based analysis, indexing, and retrieval of audio, images, animation, and video
- Presentation, rendering, synchronization, multi-modal integration/interfaces
- Real-time delivery, quality of service (including performance), capacity planning, audio/video conferencing, video-on-demand

***Learning Outcomes:***

1. Describe the media and supporting devices commonly associated with multimedia information and systems [Familiarity]
2. Explain basic multimedia presentation concepts [Familiarity]
3. Demonstrate the use of content-based information analysis in a multimedia information system [Usage]
4. Critique multimedia presentations in terms of their appropriate use of audio, video, graphics, color, and other information presentation concepts [Assessment]
5. Implement a multimedia application using a commercial authoring system [Usage]
6. For each of several media or multimedia standards, describe in non-technical language what the standard calls for, and explain how aspects of human perception might be sensitive to the limitations of that standard [Familiarity]
7. Describe the characteristics of a computer system (including identification of support tools and appropriate standards) that has to host the implementation of one of a range of possible multimedia applications [Familiarity]

# Intelligent Systems (IS)

Artificial intelligence (AI) is the study of solutions for problems that are difficult or impractical to solve with traditional methods. It is used pervasively in support of everyday applications such as email, word-processing and search, as well as in the design and analysis of autonomous agents that perceive their environment and interact rationally with the environment.

The solutions rely on a broad set of general and specialized knowledge representation schemes, problem solving mechanisms and learning techniques. They deal with sensing (e.g., speech recognition, natural language understanding, computer vision), problem-solving (e.g., search, planning), and acting (e.g., robotics) and the architectures needed to support them (e.g,. agents, multi-agents). The study of Artificial Intelligence prepares the student to determine when an AI approach is appropriate for a given problem, identify the appropriate representation and reasoning mechanism, and implement and evaluate it.

## IS. Intelligent Systems (10 Core-Tier2 hours)

| | **Core-Tier1 hours** | **Core-Tier2 hours** | **Includes Electives** |
|---|---|---|---|
| **IS/Fundamental Issues** | | **1** | **Y** |
| **IS/Basic Search Strategies** | | **4** | **N** |
| **IS/Basic Knowledge Representation and Reasoning** | | **3** | **N** |
| **IS/Basic Machine Learning** | | **2** | **N** |
| **IS/Advanced Search** | | | **Y** |
| **IS/Advanced Representation and Reasoning** | | | **Y** |
| **IS/Reasoning Under Uncertainty** | | | **Y** |
| **IS/Agents** | | | **Y** |
| **IS/Natural Language Processing** | | | **Y** |
| **IS/Advanced Machine Learning** | | | **Y** |
| **IS/Robotics** | | | **Y** |
| **IS/Perception and Computer Vision** | | | **Y** |

## IS/Fundamental Issues

*[1 Core-Tier2 hours]*

***Topics:***

- Overview of AI problems, Examples of successful recent AI applications
- What is intelligent behavior?
    - The Turing test
    - Rational versus non-rational reasoning
- Nature of environments
    - Fully versus partially observable
    - Single versus multi-agent
    - Deterministic versus stochastic
    - Static versus dynamic
    - Discrete versus continuous
- Nature of agents
    - Autonomous versus semi-autonomous
    - Reflexive, goal-based, and utility-based
    - The importance of perception and environmental interactions
- Philosophical and ethical issues [elective]

***Learning Outcomes:***

1. Describe Turing test and the "Chinese Room" thought experiment. [Familiarity]
2. Differentiate between the concepts of optimal reasoning/behavior and human-like reasoning/behavior. [Familiarity]
3. Describe a given problem domain using the characteristics of the environments in which intelligent systems must function. [Assessment]

## IS/Basic Search Strategies

*[4 Core-Tier2 hours]*

(Cross-reference AL/Basic Analysis, AL/Algorithmic Strategies, AL/Fundamental Data Structures and Algorithms)

***Topics:***

- Problem spaces (states, goals and operators), problem solving by search
- Factored representation (factoring state into variables)
- Uninformed search (breadth-first, depth-first, depth-first with iterative deepening)
- Heuristics and informed search (hill-climbing, generic best-first, A*)
- Space and time efficiency of search
- Two-player games (Introduction to minimax search)
- Constraint satisfaction (backtracking and local search methods)

***Learning Outcomes:***

1. Formulate an efficient problem space for a problem expressed in natural language (e.g., English) in terms of initial and goal states, and operators. [Usage]
2. Describe the role of heuristics and describe the trade-offs among completeness, optimality, time complexity, and space complexity. [Familiarity]
3. Describe the problem of combinatorial explosion of search space and its consequences. [Familiarity]
4. Select and implement an appropriate uninformed search algorithm for a problem, and characterize its time and space complexities. [Assessment, Usage]
5. Select and implement an appropriate informed search algorithm for a problem by designing the necessary heuristic evaluation function. [Assessment, Usage]
6. Evaluate whether a heuristic for a given problem is admissible/can guarantee optimal solution. [Assessment]
7. Formulate a problem specified in natural language (e.g., English) as a constraint satisfaction problem and implement it using a chronological backtracking algorithm or stochastic local search. [Usage]
8. Compare and contrast basic search issues with game playing issues [Familiarity]

## IS/Basic Knowledge Representation and Reasoning

### *[3 Core-Tier2 hours]*

***Topics:***

- Review of propositional and predicate logic (cross-reference DS/Basic Logic)
- Resolution and theorem proving (propositional logic only)
- Forward chaining, backward chaining
- Review of probabilistic reasoning, Bayes theorem (cross-reference with DS/Discrete Probability)

***Learning Outcomes:***

1. Translate a natural language (e.g., English) sentence into predicate logic statement. [Usage]
2. Convert a logic statement into clause form. [Usage]
3. Apply resolution to a set of logic statements to answer a query. [Usage]
4. Apply Bayes theorem to determine conditional probabilities in a problem. [Usage]

## IS/Basic Machine Learning

### *[2 Core-Tier2 hours]*

***Topics:***

- Definition and examples of broad variety of machine learning tasks, including classification
- Inductive learning
- Simple statistical-based learning such as Naive Bayesian Classifier, Decision trees
- Define over-fitting problem
- Measuring classifier accuracy

***Learning Outcomes:***

1. List the differences among the three main styles of learning: supervised, reinforcement, and unsupervised. [Familiarity]
2. Identify examples of classification tasks, including the available input features and output to be predicted. [Familiarity]
3. Explain the difference between inductive and deductive learning. [Familiarity]

4. Apply the simple statistical learning algorithm such as Naive Bayesian Classifier to a classification task and measure the classifier's accuracy. [Usage]

## IS/Advanced Search

## *[Elective]*

***Topics:***

- Constructing search trees, dynamic search space, combinatorial explosion of search space
- Stochastic search
  - Simulated annealing
  - Genetic algorithms
  - Monte-Carlo tree search
- Implementation of A* search, Beam search
- Minimax Search, Alpha-beta pruning
- Expectimax search (MDP-solving) and chance nodes

***Learning Outcomes:***

1. Design and implement a genetic algorithm solution to a problem. [Usage]
2. Design and implement a simulated annealing schedule to avoid local minima in a problem. [Usage]
3. Design and implement A*/beam search to solve a problem. [Usage]
4. Apply minimax search with alpha-beta pruning to prune search space in a two-player game. [Usage]
5. Compare and contrast genetic algorithms with classic search techniques. [Assessment]
6. Compare and contrast various heuristic searches vis-a-vis applicability to a given problem. [Assessment]

## IS/Advanced Representation and Reasoning

## *[Elective]*

***Topics:***

- Knowledge representation issues
  - Description logics
  - Ontology engineering
- Non-monotonic reasoning (e.g., non-classical logics, default reasoning, etc.)
- Argumentation
- Reasoning about action and change (e.g., situation and event calculus)
- Temporal and spatial reasoning
- Rule-based Expert Systems
- Model-based and Case-based reasoning
- Planning:
  - Partial and totally ordered planning
  - Plan graphs
  - Hierarchical planning
  - Planning and execution including conditional planning and continuous planning
  - Mobile agent/Multi-agent planning

***Learning Outcomes:***

1. Compare and contrast the most common models used for structured knowledge representation, highlighting their strengths and weaknesses. [Assessment]
2. Identify the components of non-monotonic reasoning and its usefulness as a representational mechanisms for belief systems. [Familiarity]
3. Compare and contrast the basic techniques for representing uncertainty. [Familiarity, Assessment]
4. Compare and contrast the basic techniques for qualitative representation. [Familiarity, Assessment]
5. Apply situation and event calculus to problems of action and change. [Usage]
6. Explain the distinction between temporal and spatial reasoning, and how they interrelate. [Familiarity, Assessment]
7. Explain the difference between rule-based, case-based and model-based reasoning techniques. [Familiarity, Assessment]
8. Define the concept of a planning system and how they differ from classical search techniques. [Familiarity, Assessment]
9. Describe the differences between planning as search, operator-based planning, and propositional planning, providing examples of domains where each is most applicable. [Familiarity, Assessment]
10. Explain the distinction between monotonic and non-monotonic inference. [Familiarity]

## IS/Reasoning Under Uncertainty

### *[Elective]*

***Topics:***

- Review of basic probability (cross-reference DS/Discrete Probability)
- Random variables and probability distributions
  - Axioms of probability
  - Probabilistic inference
  - Bayes' Rule
- Conditional Independence
- Knowledge representations
  - Bayesian Networks
    - Exact inference and its complexity
    - Randomized sampling (Monte Carlo) methods (e.g. Gibbs sampling)
  - Markov Networks
  - Relational probability models
  - Hidden Markov Models
- Decision Theory
  - Preferences and utility functions
  - Maximizing expected utility

***Learning Outcomes:***

1. Apply Bayes' rule to determine the probability of a hypothesis given evidence. [Usage]
2. Explain how conditional independence assertions allow for greater efficiency of probabilistic systems. [Assessment]
3. Identify examples of knowledge representations for reasoning under uncertainty. [Familiarity]
4. State the complexity of exact inference. Identify methods for approximate inference. [Familiarity]
5. Design and implement at least one knowledge representation for reasoning under uncertainty. [Usage]
6. Describe the complexities of temporal probabilistic reasoning. [Familiarity]
7. Explain the complexities of temporal probabilistic reasoning. [Assessment]
8. Design and implement an HMM as one example of a temporal probabilistic system. [Usage]
9. Describe the relationship between preferences and utility functions. [Familiarity]

10. Explain how utility functions and probabilistic reasoning can be combined to make rational decisions. [Assessment]

## IS/Agents

### *[Elective]*

(Cross-reference HCI/Collaboration and Communication)

***Topics:***

- Definitions of agents
- Agent architectures (e.g., reactive, layered, cognitive, etc.)
- Agent theory
- Rationality, Game Theory
  - Decision-theoretic agents
  - Markov decision processes (MDP)
- Software agents, personal assistants, and information access
  - Collaborative agents
  - Information-gathering agents
  - Believable agents (synthetic characters, modeling emotions in agents)
- Learning agents
- Multi-agent systems
  - Collaborating agents
  - Agent teams
  - Competitive agents (e.g., auctions, voting)
  - Swarm systems and biologically inspired models

***Learning Outcomes:***

1. List the defining characteristics of an intelligent agent. [Familiarity]
2. Characterize and contrast the standard agent architectures. [Assessment]
3. Describe the applications of agent theory to domains such as software agents, personal assistants, and believable agents. [Familiarity]
4. Describe the primary paradigms used by learning agents. [Familiarity]
5. Demonstrate using appropriate examples how multi-agent systems support agent interaction. [Usage]

# IS/Natural Language Processing

## *[Elective]*

(Cross-reference HCI/Design for Non-Mouse Interfaces)

***Topics:***

- Deterministic and stochastic grammars
- Parsing algorithms
  - CFGs and chart parsers (e.g. CYK)
  - Probabilistic CFGs and weighted CYK
- Representing meaning / Semantics
  - Logic-based knowledge representations
  - Semantic roles
  - Temporal representations
  - Beliefs, desires, and intentions
- Corpus-based methods
- N-grams and HMMs
- Smoothing and backoff
- Examples of use: POS tagging and morphology
- Information retrieval (Cross-reference IM/Information Storage and Retrieval)
  - Vector space model
    - TF & IDF
  - Precision and recall
- Information extraction
- Language translation
- Text classification, categorization
  - Bag of words model

***Learning Outcomes:***

1. Define and contrast deterministic and stochastic grammars, providing examples to show the adequacy of each. [Assessment]
2. Simulate, apply, or implement classic and stochastic algorithms for parsing natural language. [Usage]
3. Identify the challenges of representing meaning. [Familiarity]
4. List the advantages of using standard corpora. Identify examples of current corpora for a variety of NLP tasks. [Familiarity]
5. Identify techniques for information retrieval, language translation, and text classification. [Familiarity]

## IS/Advanced Machine Learning

## *[Elective]*

***Topics:***

- Definition and examples of broad variety of machine learning tasks
- General statistical-based learning, parameter estimation (maximum likelihood)
- Inductive logic programming (ILP)
- Supervised learning
  - Learning decision trees
  - Learning neural networks
  - Support vector machines (SVMs)
- Ensembles
- Nearest-neighbor algorithms
- Unsupervised Learning and clustering
  - EM
  - K-means
  - Self-organizing maps
- Semi-supervised learning
- Learning graphical models (Cross-reference IS/Reasoning under Uncertainty)
- Performance evaluation (such as cross-validation, area under ROC curve)
- Learning theory
- The problem of overfitting, the curse of dimensionality
- Reinforcement learning
  - Exploration vs. exploitation trade-off
  - Markov decision processes
  - Value and policy iteration
- Application of Machine Learning algorithms to Data Mining (Cross-reference IM/Data Mining)

***Learning Outcomes:***

1. Explain the differences among the three main styles of learning: supervised, reinforcement, and unsupervised. [Familiarity]
2. Implement simple algorithms for supervised learning, reinforcement learning, and unsupervised learning. [Usage]
3. Determine which of the three learning styles is appropriate to a particular problem domain. [Usage]
4. Compare and contrast each of the following techniques, providing examples of when each strategy is superior: decision trees, neural networks, and belief networks. [Assessment]
5. Evaluate the performance of a simple learning system on a real-world dataset. [Assessment]
6. Characterize the state of the art in learning theory, including its achievements and its shortcomings. [Familiarity]
7. Explain the problem of overfitting, along with techniques for detecting and managing the problem. [Usage]

## IS/Robotics

*[Elective]*

***Topics:***

- Overview: problems and progress
  - State-of-the-art robot systems, including their sensors and an overview of their sensor processing
  - Robot control architectures, e.g., deliberative vs. reactive control and Braitenberg vehicles
  - World modeling and world models
  - Inherent uncertainty in sensing and in control
- Configuration space and environmental maps
- Interpreting uncertain sensor data
- Localizing and mapping
- Navigation and control
- Motion planning
- Multiple-robot coordination

***Learning Outcomes:***

1. List capabilities and limitations of today's state-of-the-art robot systems, including their sensors and the crucial sensor processing that informs those systems. [Familiarity]
2. Integrate sensors, actuators, and software into a robot designed to undertake some task. [Usage]
3. Program a robot to accomplish simple tasks using deliberative, reactive, and/or hybrid control architectures. [Usage]
4. Implement fundamental motion planning algorithms within a robot configuration space. [Usage]
5. Characterize the uncertainties associated with common robot sensors and actuators; articulate strategies for mitigating these uncertainties. [Familiarity]
6. List the differences among robots' representations of their external environment, including their strengths and shortcomings. [Familiarity]
7. Compare and contrast at least three strategies for robot navigation within known and/or unknown environments, including their strengths and shortcomings. [Assessment]
8. Describe at least one approach for coordinating the actions and sensing of several robots to accomplish a single task. [Familiarity]

## IS/Perception and Computer Vision

*[Elective]*

***Topics:***

- Computer vision
  - Image acquisition, representation, processing and properties
  - Shape representation, object recognition and segmentation
  - Motion analysis
- Audio and speech recognition
- Modularity in recognition
- Approaches to pattern recognition [overlapping with machine learning]
  - Classification algorithms and measures of classification quality
  - Statistical techniques

***Learning Outcomes:***

1. Summarize the importance of image and object recognition in AI and indicate several significant applications of this technology. [Familiarity]
2. List at least three image-segmentation approaches, such as thresholding, edge-based and region-based algorithms, along with their defining characteristics, strengths, and weaknesses. [Familiarity]
3. Implement 2d object recognition based on contour- and/or region-based shape representations. [Usage]
4. Distinguish the goals of sound-recognition, speech-recognition, and speaker-recognition and identify how the raw audio signal will be handled differently in each of these cases. [Familiarity]
5. Provide at least two examples of a transformation of a data source from one sensory domain to another, e.g., tactile data interpreted as single-band 2d images. [Familiarity]
6. Implement a feature-extraction algorithm on real data, e.g., an edge or corner detector for images or vectors of Fourier coefficients describing a short slice of audio signal. [Usage]
7. Implement an algorithm combining features into higher-level percepts, e.g., a contour or polygon from visual primitives or phoneme hypotheses from an audio signal. [Usage]
8. Implement a classification algorithm that segments input percepts into output categories and quantitatively evaluates the resulting classification. [Usage]
9. Evaluate the performance of the underlying feature-extraction, relative to at least one alternative possible approach (whether implemented or not) in its contribution to the classification task (8), above. [Assessment]
10. Describe at least three classification approaches, their prerequisites for applicability, their strengths, and their shortcomings. [Familiarity]

# Networking and Communication (NC)

The Internet and computer networks are now ubiquitous and a growing number of computing activities strongly depend on the correct operation of the underlying network. Networks, both fixed and mobile, are a key part of today's and tomorrow's computing environment. Many computing applications that are used today would not be possible without networks. This dependency on the underlying network is likely to increase in the future.

The high-level learning objective of this module can be summarized as follows:

- Thinking in a networked world. The world is more and more interconnected and the use of networks will continue to increase. Students must understand how the network behaves and the key principles behind the organization and the operation of the computer networks.
- Continued study. The networking domain is rapidly evolving and a first networking course should be a starting point to other more advanced courses on network design, network management, sensor networks, etc.
- Principles and practice interact. Networking is real and many of the design choices that involve networks also depend on practical constraints. Students should be exposed to these practical constraints by experimenting with networking, using tools, and writing networked software.

There are different ways of organizing a networking course. Some educators prefer a top-down approach, i.e. the course starts from the applications and then explains reliable delivery, routing and forwarding, etc. Other educators prefer a bottom-up approach where the students start with the lower layers and build their understanding of the network, transport and application layers later.

## NC. Networking and Communication (3 Core-Tier1 hours, 7 Core-Tier2 hours)

| | Core-Tier1 hours | Core-Tier2 hours | Includes Electives |
|---|---|---|---|
| NC/Introduction | 1.5 | | N |
| NC/Networked Applications | 1.5 | | N |
| NC/Reliable Data Delivery | | 2 | N |
| NC/Routing And Forwarding | | 1.5 | N |
| NC/Local Area Networks | | 1.5 | N |
| NC/Resource Allocation | | 1 | N |
| NC/Mobility | | 1 | N |
| NC/Social Networking | | | Y |

## NC/Introduction

*[1.5 Core-Tier1 hours]*

(Cross-reference IAS/Network Security, which discusses network security and its applications.)

***Topics:***

[Core-Tier1]

- Organization of the Internet (Internet Service Providers, Content Providers, etc.)
- Switching techniques (Circuit, packet, etc.)
- Physical pieces of a network (hosts, routers, switches, ISPs, wireless, LAN, access point, firewalls, etc.)
- Layering principles (encapsulation, multiplexing)
- Roles of the different layers (application, transport, network, datalink, physical)

***Learning Outcomes:***

[Core-Tier1]

1. Articulate the organization of the Internet [Familiarity]
2. List and define the appropriate network terminology [Familiarity]
3. Describe the layered structure of a typical networked architecture [Familiarity]
4. Identify the different levels of complexity in a network (edges, core, etc.) [Familiarity]

# NC/Networked Applications

## *[1.5 Core-Tier1 hours]*

***Topics:***

[Core-Tier1]

- Naming and address schemes (DNS, IP addresses, Uniform Resource Identifiers, etc.)
- Distributed applications (client/server, peer-to-peer, cloud, etc.)
- HTTP as an application layer protocol
- Multiplexing with TCP and UDP
- Socket APIs

***Learning Outcomes:***

[Core-Tier1]

1. List the differences and the relations between names and addresses in a network [Familiarity]
2. Define the principles behind naming schemes and resource location [Familiarity]
3. Implement a simple client-server socket-based application [Usage]

# NC/Reliable Data Delivery

## *[2 Core-Tier2 hours]*

This Knowledge Unit is related to SF-Systems Fundamentals. (Cross-reference SF/State-State Transition and SF/Reliability through Redundancy.)

***Topics:***

[Core-Tier2]

- Error control (retransmission techniques, timers)
- Flow control (acknowledgements, sliding window)
- Performance issues (pipelining)
- TCP

***Learning Outcomes:***

[Core-Tier2]

1. Describe the operation of reliable delivery protocols [Familiarity]
2. List the factors that affect the performance of reliable delivery protocols [Familiarity]
3. Design and implement a simple reliable protocol [Usage]

# NC/Routing and Forwarding

## *[1.5 Core-Tier2 hours]*

***Topics:***

[Core-Tier2]

- Routing versus forwarding
- Static routing

- Internet Protocol (IP)
- Scalability issues (hierarchical addressing)

***Learning Outcomes:***

[Core-Tier2]

1. Describe the organization of the network layer [Familiarity]
2. Describe how packets are forwarded in an IP networks [Familiarity]
3. List the scalability benefits of hierarchical addressing [Familiarity]

## NC/Local Area Networks

### *[1.5 Core-Tier2 hours]*

***Topics:***

[Core-Tier2]

- Multiple Access Problem
- Common approaches to multiple access (exponential-backoff, time division multiplexing, etc)
- Local Area Networks
- Ethernet
- Switching

***Learning Outcomes:***

[Core-Tier2]

1. Describe how frames are forwarded in an Ethernet network [Familiarity]
2. Identify the differences between IP and Ethernet [Familiarity]
3. Describe the steps used in one common approach to the multiple access problem [Familiarity]
4. Describe the interrelations between IP and Ethernet [Familiarity]

## NC/Resource Allocation

### *[1 Core-Tier2 hours]*

***Topics:***

[Core-Tier2]

- Need for resource allocation
- Fixed allocation (TDM, FDM, WDM) versus dynamic allocation
- End-to-end versus network assisted approaches
- Fairness
- Principles of congestion control
- Approaches to Congestion (Content Distribution Networks, etc)

***Learning Outcomes:***

[Core-Tier2]

1. Describe how resources can be allocated in a network [Familiarity]
2. Describe the congestion problem in a large network [Familiarity]
3. Compare and contrast the fixed and dynamic allocation techniques [Assessment]

4. Compare and contrast current approaches to congestion [Assessment]

## NC/Mobility

### *[1 Core-Tier2 hours]*

***Topics:***

[Core-Tier2]

- Principles of cellular networks
- 802.11 networks
- Issues in supporting mobile nodes (home agents)

***Learning Outcomes:***

[Core-Tier2]

1. Describe the organization of a wireless network [Familiarity]
2. Describe how wireless networks support mobile users [Familiarity]

## NC/Social Networking

### *[Elective]*

***Topics:***

[Elective]

- Social Networks Overview
- Example Social Network Platforms
- Structure of social network graphs
- Social Network Analysis

***Learning Outcomes:***

[Elective]

1. Discuss the key principles of social networking [Familiarity]
2. Describe how existing social networks operate [Familiarity]
3. Construct a social network graph from network data [Usage]
4. Analyze a social network to determine who the key people are [Usage]
5. Evaluate a given interpretation of a social network question with associated data [Assessment]

# Operating Systems (OS)

An operating system defines an abstraction of hardware and manages resource sharing among the computer's users. The topics in this area explain the most basic knowledge of operating systems in the sense of interfacing an operating system to networks, teaching the difference between the kernel and user modes, and developing key approaches to operating system design and implementation. This knowledge area is structured to be complementary to Systems Fundamentals, Networks, Information Assurance, and the Parallel and Distributed Computing knowledge areas. The Systems Fundamentals and Information Assurance knowledge areas are the new ones to include contemporary issues. For example, the Systems Fundamentals includes topics such as performance, virtualization and isolation, and resource allocation and scheduling; Parallel and Distributed Systems knowledge area includes parallelism fundamentals; and Information Assurance includes forensics and security issues in depth.  Many courses in Operating Systems will draw material from across these Knowledge Areas.

## OS. Operating Systems (4 Core-Tier1 hours; 11 Core Tier2 hours)

| | Core-Tier1 hours | Core-Tier2 hours | Includes Electives |
|---|---|---|---|
| OS/Overview of Operating Systems | 2 | | N |
| OS/Operating System Principles | 2 | | N |
| OS/Concurrency | | 3 | N |
| OS/Scheduling and Dispatch | | 3 | N |
| OS/Memory Management | | 3 | N |
| OS/Security and Protection | | 2 | N |
| OS/Virtual Machines | | | Y |
| OS/Device Management | | | Y |
| OS/File Systems | | | Y |
| OS/Real Time and Embedded Systems | | | Y |
| OS/Fault Tolerance | | | Y |
| OS/System Performance Evaluation | | | Y |

## OS/Overview of Operating Systems

### *[2 Core-Tier1 hours]*

***Topics:***

- Role and purpose of the operating system
- Functionality of a typical operating system
- Mechanisms to support client-server models, hand-held devices
- Design issues (efficiency, robustness, flexibility, portability, security, compatibility)
- Influences of security, networking, multimedia, windows

***Learning Objectives:***

1. Explain the objectives and functions of modern operating systems [Familiarity].
2. Analyze the tradeoffs inherent in operating system design [Usage].
3. Describe the functions of a contemporary operating system with respect to convenience, efficiency, and the ability to evolve [Familiarity].
4. Discuss networked, client-server, distributed operating systems and how they differ from single user operating systems [Familiarity].
5. Identify potential threats to operating systems and the security features design to guard against them [Familiarity].

## OS/Operating System Principles

### *[2 Core-Tier1 hours]*

***Topics:***

- Structuring methods (monolithic, layered, modular, micro-kernel models)
- Abstractions, processes, and resources
- Concepts of application program interfaces (APIs)
- Application needs and the evolution of hardware/software techniques
- Device organization
- Interrupts: methods and implementations
- Concept of user/system state and protection, transition to kernel mode

***Learning Objectives:***

1. Explain the concept of a logical layer [Familiarity].
2. Explain the benefits of building abstract layers in hierarchical fashion [Familiarity].
3. Defend the need for APIs and middleware [Assessment].
4. Describe how computing resources are used by application software and managed by system software [Familiarity].
5. Contrast kernel and user mode in an operating system [Usage].
6. Discuss the advantages and disadvantages of using interrupt processing [Familiarity].
7. Explain the use of a device list and driver I/O queue [Familiarity].

## OS/Concurrency

*[3 Core-Tier2 hours]*

***Topics:***

- States and state diagrams (cross reference SF/State-State Transition-State Machines)
- Structures (ready list, process control blocks, and so forth)
- Dispatching and context switching
- The role of interrupts
- Managing atomic access to OS objects
- Implementing synchronization primitives
- Multiprocessor issues (spin-locks, reentrancy) (cross reference SF/Parallelism)

***Learning Objectives:***

1. Describe the need for concurrency within the framework of an operating system [Familiarity].
2. Demonstrate the potential run-time problems arising from the concurrent operation of many separate tasks [Usage].
3. Summarize the range of mechanisms that can be employed at the operating system level to realize concurrent systems and describe the benefits of each [Familiarity].
4. Explain the different states that a task may pass through and the data structures needed to support the management of many tasks [Familiarity].
5. Summarize techniques for achieving synchronization in an operating system (e.g., describe how to implement a semaphore using OS primitives) [Familiarity].
6. Describe reasons for using interrupts, dispatching, and context switching to support concurrency in an operating system [Familiarity].
7. Create state and transition diagrams for simple problem domains [Usage].

## OS/Scheduling and Dispatch

*[3 Core-Tier2 hours]*

***Topics:***

- Preemptive and non-preemptive scheduling (cross reference SF/Resource Allocation and Scheduling, PD/Parallel Performance)
- Schedulers and policies (cross reference SF/Resource Allocation and Scheduling, PD/Parallel Performance)
- Processes and threads (cross reference SF/computational paradigms)
- Deadlines and real-time issues

***Learning Objectives:***

1. Compare and contrast the common algorithms used for both preemptive and non-preemptive scheduling of tasks in operating systems, such as priority, performance comparison, and fair-share schemes [Usage].
2. Describe relationships between scheduling algorithms and application domains [Familiarity].
3. Discuss the types of processor scheduling such as short-term, medium-term, long-term, and I/O [Familiarity].
4. Describe the difference between processes and threads [Usage].
5. Compare and contrast static and dynamic approaches to real-time scheduling [Usage].
6. Discuss the need for preemption and deadline scheduling [Familiarity].
7. Identify ways that the logic embodied in scheduling algorithms are applicable to other domains, such as disk I/O, network scheduling, project scheduling, and problems beyond computing [Usage].

## OS/Memory Management

*[3 Core-Tier2 hours]*

***Topics:***

- Review of physical memory and memory management hardware
- Working sets and thrashing
- Caching

***Learning Objectives:***

1. Explain memory hierarchy and cost-performance trade-offs [Familiarity].
2. Summarize the principles of virtual memory as applied to caching and paging [Familiarity].
3. Evaluate the trade-offs in terms of memory size (main memory, cache memory, auxiliary memory) and processor speed [Assessment].
4. Defend the different ways of allocating memory to tasks, citing the relative merits of each [Assessment].
5. Describe the reason for and use of cache memory (performance and proximity, different dimension of how caches complicate isolation and VM abstraction) [Familiarity].
6. Discuss the concept of thrashing, both in terms of the reasons it occurs and the techniques used to recognize and manage the problem [Familiarity].

## OS/Security and Protection

*[2 Core-Tier2 hours]*

***Topics:***

- Overview of system security
- Policy/mechanism separation
- Security methods and devices
- Protection, access control, and authentication
- Backups

***Learning Objectives:***

1. Defend the need for protection and security in an OS (cross reference IAS/Security Architecture and Systems Administration/Investigating Operating Systems Security for various systems) [Assessment].
2. Summarize the features and limitations of an operating system used to provide protection and security (cross reference IAS/Security Architecture and Systems Administration) [Familiarity].
3. Explain the mechanisms available in an OS to control access to resources (cross reference IAS/Security Architecture and Systems Administration/Access Control/Configuring systems to operate securely as an IT system) [Familiarity].
4. Carry out simple system administration tasks according to a security policy, for example creating accounts, setting permissions, applying patches, and arranging for regular backups (cross reference IAS/Security Architecture and Systems Administration ) [Usage].

## OS/Virtual Machines

*[Elective]*

***Topics:***

- Types of virtualization (Hardware/Software, OS, Server, Service, Network, etc.)
- Paging and virtual memory
- Virtual file systems
- Virtual file
- Hypervisors
- Portable virtualization; emulation vs. isolation
- Cost of virtualization

***Learning Objectives:***

1. Explain the concept of virtual memory and how it is realized in hardware and software [Familiarity].
2. Differentiate emulation and isolation [Familiarity].
3. Evaluate virtualization trade-offs [Assessment].
4. Discuss hypervisors and the need for them in conjunction with different types of hypervisors [Usage].

## OS/Device Management

*[Elective]*

***Topics:***

- Characteristics of serial and parallel devices
- Abstracting device differences
- Buffering strategies
- Direct memory access
- Recovery from failures

***Learning Objectives:***

1. Explain the key difference between serial and parallel devices and identify the conditions in which each is appropriate [Familiarity].
2. Identify the relationship between the physical hardware and the virtual devices maintained by the operating system [Usage].
3. Explain buffering and describe strategies for implementing it [Familiarity].
4. Differentiate the mechanisms used in interfacing a range of devices (including hand-held devices, networks, multimedia) to a computer and explain the implications of these for the design of an operating system [Usage].
5. Describe the advantages and disadvantages of direct memory access and discuss the circumstances in which its use is warranted [Usage].
6. Identify the requirements for failure recovery [Familiarity].
7. Implement a simple device driver for a range of possible devices [Usage].

## OS/File Systems

### *[Elective]*

***Topics:***

- Files: data, metadata, operations, organization, buffering, sequential, nonsequential
- Directories: contents and structure
- File systems: partitioning, mount/unmount, virtual file systems
- Standard implementation techniques
- Memory-mapped files
- Special-purpose file systems
- Naming, searching, access, backups
- Journaling and log-structured file systems

***Learning Objectives:***

1. Summarize the full range of considerations in the design of file systems [Familiarity].
2. Compare and contrast different approaches to file organization, recognizing the strengths and weaknesses of each [Usage].
3. Summarize how hardware developments have led to changes in the priorities for the design and the management of file systems [Familiarity].
4. Summarize the use of journaling and how log-structured file systems enhance fault tolerance [Familiarity].

## OS/Real Time and Embedded Systems

### *[Elective]*

***Topics:***

- Process and task scheduling
- Memory/disk management requirements in a real-time environment
- Failures, risks, and recovery
- Special concerns in real-time systems

***Learning Objectives:***

1. Describe what makes a system a real-time system [Familiarity].
2. Explain the presence of and describe the characteristics of latency in real-time systems [Familiarity].
3. Summarize special concerns that real-time systems present and how these concerns are addressed [Familiarity].

## OS/Fault Tolerance

### *[Elective]*

***Topics:***

- Fundamental concepts: reliable and available systems (cross reference SF/Reliability through Redundancy)
- Spatial and temporal redundancy (cross reference SF/Reliability through Redundancy)
- Methods used to implement fault tolerance
- Examples of OS mechanisms for detection, recovery, restart to implement fault tolerance, use of these techniques for the OS's own services

***Learning Objectives:***

1. Explain the relevance of the terms fault tolerance, reliability, and availability [Familiarity].
2. Outline the range of methods for implementing fault tolerance in an operating system [Familiarity].
3. Explain how an operating system can continue functioning after a fault occurs [Familiarity].

## OS/System Performance Evaluation

## *[Elective]*

***Topics:***

- Why system performance needs to be evaluated (cross reference SF/Performance/Figures of performance merit)
- What is to be evaluated (cross reference SF/Performance/Figures of performance merit)
- Policies for caching, paging, scheduling, memory management, security, and so forth
- Evaluation models: deterministic, analytic, simulation, or implementation-specific
- How to collect evaluation data (profiling and tracing mechanisms)

***Learning Objectives:***

1. Describe the performance measurements used to determine how a system performs [Familiarity].
2. Explain the main evaluation models used to evaluate a system [Familiarity].

# Platform-Based Development (PBD)

Platform-based development is concerned with the design and development of software applications that reside on specific software platforms. In contrast to general purpose programming, platform-based development takes into account platform-specific constraints. For instance web programming, multimedia development, mobile computing, app development, and robotics are examples of relevant platforms which provide specific services/APIs/hardware which constrain development. Such platforms are characterized by the use of specialized APIs, distinct delivery/update mechanisms, and being abstracted away from the machine level. Platform-based development may be applied over a wide breadth of ecosystems.

While we recognize that some platforms (e.g., web development) are prominent, we are also cognizant of the fact that no particular platform should be specified as a requirement in the CS2013 curricular guidelines. Consequently, this Knowledge Area highlights many of the platforms which have become popular, without including any such platform in the core curriculum. We note that the general skill of developing with respect to an API or a constrained environment is covered in other Knowledge Areas, such as SDF-Software Development Fundamentals. Platform-based development further emphasizes such general skills within the context of particular platforms.

## PBD. Platform-Based Development (Elective)

| | Core-Tier1 hours | Core-Tier2 hours | Includes Electives |
|---|---|---|---|
| PBD/Introduction | | | Y |
| PBD/Web Platforms | | | Y |
| PBD/Mobile Platforms | | | Y |
| PBD/Industrial Platforms | | | Y |
| PBD/Game Platforms | | | Y |

## PBD/Introduction

*[Elective]*

This unit describes the fundamental differences that Platform-Based Development has over traditional software development.

***Topics:***

- Overview of platforms (Web, Mobile, Game, Industrial etc)
- Programming via platform-specific APIs
- Overview of Platform Languages (Objective C, HTML5, etc)
- Programming under platform constraints

***Learning Outcomes:***

[Elective]

1. Describe how platform-based development differs from general purpose programming [Familiarity]
2. List characteristics of platform languages [Familiarity]
3. Write and execute a simple platform-based program [Usage]
4. List the advantages and disadvantages of programming with platform constraints [Familiarity]

## PBD/Web Platforms

*[Elective]*

***Topics:***

- Web programming languages (HTML5, Java Script, PHP, CSS, etc.)
- Web platform constraints
- Software as a Service (SaaS)
- Web standards

***Learning Outcomes:***

[Elective]

1. Design and Implement a simple web application [Usage]
2. Describe the constraints that the web puts on developers [Familiarity]
3. Compare and contrast web programming with general purpose programming [Assessment]
4. Describe the differences between Software-as-a-Service and traditional software products [Familiarity]
5. Discuss how web standards impact software development [Familiarity]
6. Review an existing web application against a current web standard [Assessment]

## PBD/Mobile Platforms

*[Elective]*

***Topics:***

- Mobile Programming Languages (Objective C, Java Script, Java, etc.)
- Challenges with mobility and wireless communication
- Location-aware applications

- Performance / power tradeoffs
- Mobile platform constraints
- Emerging Technologies

***Learning Outcomes:***

[Elective]

1. Design and implement a mobile application for a given mobile platform. [Usage]
2. Discuss the constraints that mobile platforms put on developers [Familiarity]
3. Discuss the performance vs. power tradeoff [Familiarity]
4. Compare and Contrast mobile programming with general purpose programming [Assessment]

## PBD/Industrial Platforms

## *[Elective]*

This knowledge unit is related to IS/Robotics.

***Topics:***

- Types of Industrial Platforms (Mathematic, Robotics, Industrial Controls, etc.)
- Robotic Software and its Architecture
- Domain Specific Languages
- Industrial Platform Constraints

***Learning Outcomes:***

[Elective]

1. Design and implement an industrial application on a given platform (Lego Mindstorms, Matlab, etc.) [Usage]
2. Compare and contrast domain specific languages with general purpose programming languages. [Assessment]
3. Discuss the constraints that a given industrial platforms impose on developers [Familiarity]

## PBD/Game Platforms

## *[Elective]*

***Topics:***

- Types of Game Platforms (XBox, Wii, PlayStation, etc)
- Game Platform Languages (C++, Java, Lua, Python, etc)
- Game Platform Constraints

***Learning Outcomes:***

[Elective]

1. Design and Implement a simple application on a game platform. [Usage]
2. Describe the constraints that game platforms impose on developers. [Familiarity]
3. Compare and contrast game programming with general purpose programming [Assessment]

# Parallel and Distributed Computing (PD)

The past decade has brought explosive growth in multiprocessor computing, including multi-core processors and distributed data centers.  As a result, parallel and distributed computing has moved from a largely elective topic to become more of a core component of undergraduate computing curricula.  Both parallel and distributed computing entail the logically simultaneous execution of multiple processes, whose operations have the potential to interleave in complex ways.  Parallel and distributed computing builds on foundations in many areas, including an understanding of fundamental systems concepts such as concurrency and parallel execution, consistency in state/memory manipulation, and latency.  Communication and coordination among processes is rooted in the message-passing and shared-memory models of computing and such algorithmic concepts as atomicity, consensus, and conditional waiting. Achieving speedup in practice requires an understanding of parallel algorithms, strategies for problem decomposition, system architecture, detailed implementation strategies, and performance analysis and tuning. Distributed systems highlight the problems of security and fault tolerance, emphasize the maintenance of replicated state, and introduce additional issues that bridge to computer networking.

Because parallelism interacts with so many areas of computing, including at least algorithms, languages, systems, networking, and hardware, many curricula will put different parts of the knowledge area in different courses, rather than in a dedicated course.  While we acknowledge that computer science is moving in this direction and may reach that point, in 2013 this process is still in flux and we feel it provides more useful guidance to curriculum designers to aggregate the fundamental parallelism topics in one place.  Note, however, that the fundamentals of concurrency and mutual exclusion appear in Systems Fundamentals.  Many curricula may choose to introduce parallelism and concurrency in the same course (see below for the distinction intended by these terms).  Further, we note that the topics and learning outcomes listed below include only brief mentions of purely elective coverage.  At the present time, there is too much diversity in topics that share little in common (including for example, parallel scientific computing, process calculi, and non-blocking data structures) to recommend particular topics be covered in elective courses.

Because the terminology of parallel and distributed computing varies among communities, we provide here brief descriptions of the intended senses of a few terms. This list is not exhaustive or definitive, but is provided for the sake of clarity:

- *Parallelism:* Using additional computational resources simultaneously, usually for speedup.
- *Concurrency:* Efficiently and correctly managing concurrent access to resources.
- *Activity*: A computation that may proceed concurrently with others; for example a program, process, thread, or active parallel hardware component.
- *Atomicity*: Rules and properties governing whether an action is observationally indivisible; for example setting all of the bits in a word, transmitting a single packet, or completing a transaction.
- *Consensus*: Agreement among two or more activities about a given predicate; for example the value of a counter, the owner of a lock, or the termination of a thread.
- *Consistency*: Rules and properties governing agreement about the values of variables written, or messages produced, by some activities and used by others (thus possibly exhibiting a *data race*); for example, *sequential consistency*, stating that the values of all variables in a shared memory parallel program are equivalent to that of a single program performing some interleaving of the memory accesses of these activities.
- *Multicast*: A message sent to possibly many recipients, generally without any constraints about whether some recipients receive the message before others. An *event* is a multicast message sent to a designated set of *listeners* or *subscribers*.

As multi-processor computing continues to grow in the coming years, so too will the role of parallel and distributed computing in undergraduate computing curricula. In addition to the guidelines presented here, we also direct the interested reader to the document entitled "NSF/TCPP Curriculum Initiative on Parallel and Distributed Computing - Core Topics for Undergraduates", available from the website: http://www.cs.gsu.edu/~tcpp/curriculum/.

***General cross-referencing note:*** Systems Fundamentals also contains an introduction to parallelism (SF/Computational Paradigms, SF/System Support for Parallelism, SF/Performance).

The introduction to parallelism in SF complements the one here and there is no ordering constraint between them. In SF, the idea is to provide a unified view of the system support for simultaneous execution at multiple levels of abstraction (parallelism is inherent in gates, processors, operating systems, servers, etc.), whereas here the focus is on a preliminary understanding of parallelism as a computing primitive and the complications that arise in parallel and concurrent programming. Given these different perspectives, the hours assigned to each are not redundant: the layered systems view and the high-level computing concepts are accounted for separately in terms of the core hours.

## PD. Parallel and Distributed Computing (5 Core-Tier1 hours, 9 Core-Tier2 hours)

| | Core-Tier1 hours | Core-Tier2 hours | Includes Electives |
|---|---|---|---|
| PD/Parallelism Fundamentals | 2 | | N |
| PD/Parallel Decomposition | 1 | 3 | N |
| PD/Communication and Coordination | 1 | 3 | Y |
| PD/Parallel Algorithms, Analysis, and Programming | | 3 | Y |
| PD/Parallel Architecture | 1 | 1 | Y |
| PD/Parallel Performance | | | Y |
| PD/Distributed Systems | | | Y |
| PD/Cloud Computing | | | Y |
| PD/Formal Models and Semantics | | | Y |

## PD/Parallelism Fundamentals

### *[2 Core-Tier1 hours]*

Build upon students' familiarity with the notion of basic parallel execution--a concept addressed in Systems Fundamentals--to delve into the complicating issues that stem from this notion, such as race conditions and liveness.

(Cross-reference SF/Computational Paradigms and SF/System Support for Parallelism)

***Topics:***

[Core-Tier1]

- Multiple simultaneous computations
- Goals of parallelism (e.g., throughput) versus concurrency (e.g., controlling access to shared resources)
- Parallelism, communication, and coordination
    - Programming constructs for coordinating multiple simultaneous computatations
    - Need for synchronization
- Programming errors not found in sequential programming
    - Data races (simultaneous read/write or write/write of shared state)
    - Higher-level races (interleavings violating program intention, undesired non-determinism)
    - Lack of liveness/progress (deadlock, starvation)

***Learning outcomes:***

[Core-Tier1]

1. Distinguish using computational resources for a faster answer from managing efficient access to a shared resource [Familiarity]
2. Distinguish multiple sufficient programming constructs for synchronization that may be inter-implementable but have complementary advantages [Familiarity]
3. Distinguish data races from higher level races [Familiarity]

## PD/Parallel Decomposition

### *[1 Core-Tier1 hour, 3 Core-Tier2 hours]*

(Cross-reference SF/System Support for Parallelism)

***Topics:***

[Core-Tier1]

- Need for communication and coordination/synchronization
- Independence and partitioning

[Core-Tier2]

- Basic knowledge of parallel decomposition concepts (cross-reference SF/System Support for Parallelism)
- Task-based decomposition
    - Implementation strategies such as threads
- Data-parallel decomposition

        - Strategies such as SIMD and MapReduce
- Actors and reactive processes (e.g., request handlers)

***Learning outcomes:***

[Core-Tier1]

1. Explain why synchronization is necessary in a specific parallel program [Usage]

[Core-Tier2]

2. Write a correct and scalable parallel algorithm [Usage]
3. Parallelize an algorithm by applying task-based decomposition [Usage]
4. Parallelize an algorithm by applying data-parallel decomposition [Usage]

## PD/Communication and Coordination

## *[1 Core-Tier1 hour, 3 Core-Tier2 hours]*

(Cross-reference OS/Concurrency for mechanism implementation issues.)

***Topics:***

[Core-Tier1]

- Shared Memory
    - Consistency, and its role in programming language guarantees for data-race-free programs

[Core-Tier2]

- Consistency in shared memory models
- Message passing
    - Point-to-point versus multicast (or event-based) messages
    - Blocking versus non-blocking styles for sending and receiving messages
    - Message buffering (cross-reference PF/Fundamental Data Structures/Queues)
- Atomicity
    - Specifying and testing atomicity and safety requirements
    - Granularity of atomic accesses and updates, and the use of constructs such as critical sections or transactions to describe them
    - Mutual Exclusion using locks, semaphores, monitors, or related constructs
        - Potential for liveness failures and deadlock (causes, conditions, prevention)
    - Composition
        - Composing larger granularity atomic actions using synchronization
        - Transactions, including optimistic and conservative approaches

[Elective]

- Consensus
    - (Cyclic) barriers, counters, or related constructs
- Conditional actions
    - Conditional waiting (e.g., using condition variables)

***Learning outcomes:***

[Core-Tier1]

1. Use mutual exclusion to avoid a given race condition [Usage]

[Core-Tier2]

2. Give an example of an ordering of accesses among concurrent activities that is not sequentially consistent [Familiarity]
3. Give an example of a scenario in which blocking message sends can deadlock [Usage]
4. Explain when and why multicast or event-based messaging can be preferable to alternatives [Familiarity]
5. Write a program that correctly terminates when all of a set of concurrent tasks have completed [Usage]
6. Use a properly synchronized queue to buffer data passed among activities [Usage]
7. Explain why checks for preconditions, and actions based on these checks, must share the same unit of atomicity to be effective [Familiarity]
8. Write a test program that can reveal a concurrent programming error; for example, missing an update when two activities both try to increment a variable [Usage]
9. Describe at least one design technique for avoiding liveness failures in programs using multiple locks or semaphores [Familiarity]
10. Describe the relative merits of optimistic versus conservative concurrency control under different rates of contention among updates [Familiarity]
11. Give an example of a scenario in which an attempted optimistic update may never complete [Familiarity]

[Elective]

12. Use semaphores or condition variables to block threads until a necessary precondition holds [Usage]

## PD/Parallel Algorithms, Analysis, and Programming

### *[3 Core-Tier2 hours]*

***Topics:***

[Core-Tier2]

- Critical paths, work and span, and the relation to Amdahl's law (cross-reference SF/Performance)
- Speed-up and scalability
- Naturally (embarassingly) parallel algorithms
- Parallel algorithmic patterns (divide-and-conquer, map and reduce, master-workers, others)
    - Specific algorithms (e.g., parallel MergeSort)

[Elective]

- Parallel graph algorithms (e.g., parallel shortest path, parallel spanning tree) (cross-reference AL/Algorithmic Strategies/Divide-and-conquer)
- Parallel matrix computations
- Producer-consumer and pipelined algorithms

***Learning outcomes:***

[Core-Tier2]

1. Define “critical path”, “work”, and “span” [Familiarity]
2. Compute the work and span, and determine the critical path with respect to a parallel execution diagram [Usage]
3. Define “speed-up” and explain the notion of an algorithm’s scalability in this regard [Familiarity]
4. Identify independent tasks in a program that may be parallelized [Usage]
5. Characterize features of a workload that allow or prevent it from being naturally parallelized [Familiarity]
6. Implement a parallel divide-and-conquer (and/or graph algorithm) and empirically measure its performance relative to its sequential analog [Usage]
7. Decompose a problem (e.g., counting the number of occurrences of some word in a document) via map and reduce operations [Usage]

[Elective]

8. Provide an example of a problem that fits the producer-consumer paradigm [Familiarity]
9. Give examples of problems where pipelining would be an effective means of parallelization [Familiarity]
10. Identify issues that arise in producer-consumer algorithms and mechanisms that may be used for addressing them [Familiarity]

## PD/Parallel Architecture

### *[1 Core-Tier1 hour, 1 Core-Tier2 hour]*

The topics listed here are related to knowledge units in the Architecture and Organization area (AR/Assembly Level Machine Organization and AR/Multiprocessing and Alternative Architectures). Here, we focus on parallel architecture from the standpoint of applications, whereas the Architecture and Organization area presents the topic from the hardware perspective.

[Core-Tier1]

- Multicore processors
- Shared vs. distributed memory

[Core-Tier2]

- Symmetric multiprocessing (SMP)
- SIMD, vector processing

[Elective]

- GPU, co-processing
- Flynn’s taxonomy
- Instruction level support for parallel programming
    - Atomic instructions such as Compare and Set
- Memory issues
    - Multiprocessor caches and cache coherence
    - Non-uniform memory access (NUMA)
- Topologies
    - Interconnects
    - Clusters
    - Resource sharing (e.g., buses and interconnects)

***Learning outcomes:***

[Core-Tier1]

1. Explain the differences between shared and distributed memory [Familiarity]

[Core-Tier2]

2. Describe the SMP architecture and note its key features [Familiarity]
3. Characterize the kinds of tasks that are a natural match for SIMD machines [Familiarity]

[Elective]

4. Explain the features of each classification in Flynn's taxonomy [Familiarity]
5. Describe the challenges in maintaining cache coherence [Familiarity]
6. Describe the key features of different distributed system topologies [Familiarity]

## PD/Parallel Performance

## *[Elective]*

***Topics:***

- Load balancing
- Performance measurement
- Scheduling and contention (cross-reference OS/Scheduling and Dispatch)
- Evaluating communication overhead
- Data management
  - Non-uniform communication costs due to proximity (cross-reference SF/Proximity)
  - Cache effects (e.g., false sharing)
  - Maintaining spatial locality
- Impact of composing multiple concurrent components
- Power usage and management

***Learning outcomes:***

[Elective]

1. Calculate the implications of Amdahl's law for a particular parallel algorithm [Usage]
2. Describe how data distribution/layout can affect an algorithm's communication costs [Familiarity]
3. Detect and correct a load imbalance [Usage]
4. Detect and correct an instance of false sharing [Usage]
5. Explain the impact of scheduling on parallel performance [Familiarity]
6. Explain performance impacts of data locality [Familiarity]
7. Explain the impact and trade-off related to power usage on parallel performance [Familiarity]

# PD/Distributed Systems

## *[Elective]*

***Topics:***

- Faults (cross-reference OS/Fault Tolerance)
    - Network-based (including partitions) and node-based failures
    - Impact on system-wide guarantees (e.g., availability)
- Distributed message sending
    - Data conversion and transmission
    - Sockets
    - Message sequencing
    - Buffering, retrying, and dropping messages
- Distributed system design tradeoffs
    - Latency versus throughput
    - Consistency, availability, partition tolerance
- Distributed service design
    - Stateful versus stateless protocols and services
    - Session (connection-based) designs
    - Reactive (IO-triggered) and multithreaded designs
- Core distributed algorithms
    - Election, discovery

***Learning outcomes:***

[Elective]

1. Distinguish network faults from other kinds of failures [Familiarity]
2. Explain why synchronization constructs such as simple locks are not useful in the presence of distributed faults [Familiarity]
3. Give examples of problems for which consensus algorithms such as leader election are required [Usage]
4. Write a program that performs any required marshalling and conversion into message units, such as packets, to communicate interesting data between two hosts [Usage]
5. Measure the observed throughput and response latency across hosts in a given network [Usage]
6. Explain why no distributed system can be simultaneously consistent, available, and partition tolerant [Familiarity]
7. Implement a simple server -- for example, a spell checking service [Usage]
8. Explain the tradeoffs among overhead, scalability, and fault tolerance when choosing a stateful v. stateless design for a given service [Familiarity]
9. Describe the scalability challenges associated with a service growing to accommodate many clients, as well as those associated with a service only transiently having many clients [Familiarity]

## PD/Cloud Computing

### *[Elective]*

***Topics:***

- Internet-Scale computing
  - Task partitioning (cross-reference PD/Parallel Algorithms, Analysis, and Programming)
  - Data access
  - Clusters, grids, and meshes
- Cloud services
  - Infrastructure as a service
    - Elasticity of resources
    - Platform APIs
  - Software as a service
  - Security
  - Cost management
- Virtualization (cross-reference SF/Virtualization and Isolation and OS/Virtual Machines)
  - Shared resource management
  - Migration of processes
- Cloud-based data storage
  - Shared access to weakly consistent data stores
  - Data synchronization
  - Data partitioning
  - Distributed file systems (cross-reference IM/Distributed Databases)
  - Replication

***Learning outcomes:***

[Elective]

1. Discuss the importance of elasticity and resource management in cloud computing. [Familiarity]
2. Explain strategies to synchronize a common view of shared data across a collection of devices [Familiarity]
3. Explain the advantages and disadvantages of using virtualized infrastructure [Familiarity]
4. Deploy an application that uses cloud infrastructure for computing and/or data resources [Usage]
5. Appropriately partition an application between a client and resources [Usage]

## PD/Formal Models and Semantics

### *[Elective]*

***Topics:***

- Formal models of processes and message passing, including algebras such as Communicating Sequential Processes (CSP) and pi-calculus
- Formal models of parallel computation, including the Parallel Random Access Machine (PRAM) and alternatives such as Bulk Synchronous Parallel (BSP)
- Formal models of computational dependencies
- Models of (relaxed) shared memory consistency and their relation to programming language specifications
- Algorithmic correctness criteria including linearizability
- Models of algorithmic progress, including non-blocking guarantees and fairness
- Techniques for specifying and checking correctness properties such as atomicity and freedom from data races

***Learning outcomes:***

[Elective]

1. Model a concurrent process using a formal model, such as pi-calculus [Usage]
2. Explain the characteristics of a particular formal parallel model [Familiarity]
3. Formally model a shared memory system to show if it is consistent [Usage]
4. Use a model to show progress guarantees in a parallel algorithm [Usage]
5. Use formal techniques to show that a parallel algorithm is correct with respect to a safety or liveness property [Usage]
6. Decide if a specific execution is linearizable or not [Usage]

# Programming Languages (PL)

Programming languages are the medium through which programmers precisely describe concepts, formulate algorithms, and reason about solutions. In the course of a career, a computer scientist will work with many different languages, separately or together. Software developers must understand the programming models underlying different languages, and make informed design choices in languages supporting multiple complementary approaches. Computer scientists will often need to learn new languages and programming constructs, and must understand the principles underlying how programming language features are defined, composed, and implemented. The effective use of programming languages, and appreciation of their limitations, also requires a basic knowledge of programming language translation and static program analysis, as well as run-time components such as memory management.

## PL. Programming Languages (8 Core-Tier1 hours, 20 Core-Tier2 hours)

| | Core-Tier1 hours | Core-Tier2 hours | Includes Electives |
|---|---|---|---|
| PL/Object-Oriented Programming | 4 | 6 | N |
| PL/Functional Programming | 3 | 4 | N |
| PL/Event-Driven and Reactive Programming | | 2 | N |
| PL/Basic Type Systems | 1 | 4 | N |
| PL/Program Representation | | 1 | N |
| PL/Language Translation and Execution | | 3 | N |
| PL/Syntax Analysis | | | Y |
| PL/Compiler Semantic Analysis | | | Y |
| PL/Code Generation | | | Y |
| PL/Runtime Systems | | | Y |
| PL/Static Analysis | | | Y |
| PL/Advanced Programming Constructs | | | Y |
| PL/Concurrency and Parallelism | | | Y |
| PL/Type Systems | | | Y |
| PL/Formal Semantics | | | Y |
| PL/Language Pragmatics | | | Y |
| PL/Logic Programming | | | Y |

Note:

- Some topics from one or more of the first three Knowledge Units (*Object-Oriented Programming*, *Functional Programming*, *Event-Driven and Reactive Programming*) are likely to be integrated with topics in the Software Development Fundamentals Knowledge Area in a curriculum's introductory courses. Curricula will differ on which topics are integrated in this fashion and which are delayed until later courses on software development and programming languages.
- Some of the most important core learning outcomes are relevant to object-oriented programming, functional programming, and, in fact, all programming. These learning outcomes are *repeated* in the *Object-Oriented Programming* and *Functional Programming* Knowledge Units, with a note to this effect. We do not intend that a

curriculum necessarily needs to cover them multiple times, though some will. We repeat them only because they do not naturally fit in only one Knowledge Unit.

## PL/Object-Oriented Programming

### *[4 Core-Tier1 hours, 6 Core-Tier2 hours]*

***Topics:***

[Core-Tier1]

- Object-oriented design
  - Decomposition into objects carrying state and having behavior
  - Class-hierarchy design for modeling
- Definition of classes: fields, methods, and constructors
- Subclasses, inheritance, and method overriding
- Dynamic dispatch: definition of method-call

[Core-Tier2]

- Subtyping (cross-reference PL/Type Systems)
  - Subtype polymorphism; implicit upcasts in typed languages
  - Notion of behavioral replacement: subtypes acting like supertypes
  - Relationship between subtyping and inheritance
- Object-oriented idioms for encapsulation
  - Privacy and visibility of class members
  - Interfaces revealing only method signatures
  - Abstract base classes
- Using collection classes, iterators, and other common library components

***Learning outcomes:***

[Core-Tier1]

1. Compare and contrast (1) the procedural/functional approach—defining a function for each operation with the function body providing a case for each data variant—and (2) the object-oriented approach—defining a class for each data variant with the class definition providing a method for each operation. Understand both as defining a matrix of operations and variants. [Assessment] *This outcome also appears in PL/Functional Programming.*
2. Use subclassing to design simple class hierarchies that allow code to be reused for distinct subclasses. [Usage]
3. Correctly reason about control flow in a program using dynamic dispatch. [Usage]

[Core-Tier2]

4. Explain the relationship between object-oriented inheritance (code-sharing and overriding) and subtyping (the idea of a subtype being usable in a context that expects the supertype). [Familiarity]
5. Use multiple encapsulation mechanisms, such as function closures, object-oriented interfaces, and support for abstract datatypes, in multiple programming languages. [Usage] *This outcome also appears in PL/Functional Programming.*

6. Define and use iterators and other operations on aggregates, including operations that take functions as arguments, in multiple programming languages, selecting the most natural idioms for each language. [Usage] *This outcome also appears in PL/Functional Programming.*

## PL/Functional Programming

### *[3 Core-Tier1 hours, 4 Core-Tier2 hours]*

***Topics:***

[Core-Tier1]

- Effect-free programming
  - Function calls have no side effects, facilitating compositional reasoning
  - Variables are immutable, preventing unexpected changes to program data by other code
  - Data can be freely aliased or copied without introducing unintended effects from mutation
- Processing structured data (e.g., trees) via functions with cases for each data variant
  - Associated language constructs such as discriminated unions and pattern-matching over them
  - Functions defined over compound data in terms of functions applied to the constituent pieces
- First-class functions (taking, returning, and storing functions)

[Core-Tier2]

- Function closures (functions using variables in the enclosing lexical environment)
  - Basic meaning and definition -- creating closures at run-time by capturing the environment
  - Canonical idioms: call-backs, arguments to iterators, reusable code via function arguments
  - Using a closure to encapsulate data in its environment
  - Currying and partial application
- Defining higher-order operations on aggregates, especially map, reduce/fold, and filter

***Learning outcomes:***

[Core-Tier1]

1. Compare and contrast (1) the procedural/functional approach—defining a function for each operation with the function body providing a case for each data variant—and (2) the object-oriented approach—defining a class for each data variant with the class definition providing a method for each operation. Understand both as defining a matrix of operations and variants. [Assessment] *This outcome also appears in PL/Object-Oriented Programming.*
2. Write basic algorithms that avoid assigning to mutable state or considering reference equality. [Usage]
3. Write useful functions that take and return other functions. [Usage]

[Core-Tier2]

4. Correctly reason about variables and lexical scope in a program using function closures. [Usage]
5. Use multiple encapsulation mechanisms, such as function closures, object-oriented interfaces, and support for abstract datatypes, in multiple programming languages. [Usage] *This outcome also appears in PL/Object-Oriented Programming.*
6. Define and use iterators and other operations on aggregates, including operations that take functions as arguments, in multiple programming languages, selecting the most natural idioms for each language. [Usage] *This outcome also appears in PL/Object-Oriented Programming.*

## PL/Event-Driven and Reactive Programming

### *[2 Core-Tier2 hours]*

This material can stand alone or be integrated with other knowledge units on concurrency, asynchrony, and threading to allow contrasting events with threads.

***Topics:***

- Events and event handlers
- Canonical uses such as GUIs, mobile devices, robots, servers
- Using a reactive framework
    - Defining event handlers/listeners
    - Main event loop not under event-handler-writer's control
- Externally-generated events and program-generated events
- Separation of model, view, and controller

***Learning outcomes:***

1. Write event handlers for use in reactive systems, such as GUIs. [Usage]
2. Explain why an event-driven programming style is natural in domains where programs react to external events. [Familiarity]

## PL/Basic Type Systems

### *[1 Core-Tier1 hour, 4 Core-Tier2 hours]*

The core-tier2 hours would be profitably spent both on the core-tier2 topics and on a less shallow treatment of the core-tier1 topics and learning outcomes.

***Topics:***

[Core-Tier1]

- A type as a set of values together with a set of operations
    - Primitive types (e.g., numbers, Booleans)
    - Compound types built from other types (e.g., records, unions, arrays, lists, functions, references)
- Association of types to variables, arguments, results, and fields
- Type safety and errors caused by using values inconsistently with their intended types
- Goals and limitations of static typing
    - Eliminating some classes of errors without running the program
    - Undecidability means static analysis must conservatively approximate program behavior

[Core-Tier2]

- Generic types (parametric polymorphism)
    - Definition
    - Use for generic libraries such as collections
    - Comparison with ad hoc polymorphism (overloading) and subtype polymorphism
- Complementary benefits of static and dynamic typing
    - Errors early vs. errors late/avoided
    - Enforce invariants during code development and code maintenance vs. postpone typing decisions while prototyping and conveniently allow flexible coding patterns such as heterogeneous collections

- Avoid misuse of code vs. allow more code reuse
- Detect incomplete programs vs. allow incomplete programs to run

***Learning outcomes:***

[Core-Tier1]

1. For multiple programming languages, identify program properties checked statically and program properties checked dynamically. Use this knowledge when writing and debugging programs. [Usage]

[Core-Tier2]

2. Define and use program pieces (such as functions, classes, methods) that use generic types. [Usage]
3. Explain benefits and limitations of static typing. [Familiarity]

## PL/Program Representation

### *[1 Core-Tier2 hour]*

***Topics:***

- Programs that take (other) programs as input such as interpreters, compilers, type-checkers, documentation generators, etc.
- Abstract syntax trees; contrast with concrete syntax
- Data structures to represent code for execution, translation, or transmission

***Learning outcomes:***

1. Write a program to process some representation of code for some purpose, such as an interpreter, an expression optimizer, a documentation generator, etc. [Usage]

## PL/Language Translation and Execution

### *[3 Core-Tier2 hours]*

***Topics:***

- Interpretation vs. compilation to native code vs. compilation to portable intermediate representation
- Language translation pipeline: parsing, optional type-checking, translation, linking, execution
    - Execution as native code or within a virtual machine
    - Alternatives like dynamic loading and dynamic (or "just-in-time") code generation
- Run-time representation of core language constructs such as objects (method tables) and first-class functions (closures)
- Run-time layout of memory: call-stack, heap, static data
    - Implementing loops, recursion, and tail calls
- Memory management
    - Manual memory management: allocating, deallocating, and reusing heap memory
    - Automated memory management: garbage collection as an automated technique using the notion of reachability

***Learning outcomes:***

1. Distinguish syntax and parsing from semantics and evaluation. [Familiarity]
2. Distinguish a language definition (what constructs mean) from a particular language implementation (compiler vs. interpreter, run-time representation of data objects, etc.). [Familiarity]
3. Explain how programming language implementations typically organize memory into global data, text, heap, and stack sections and how features such as recursion and memory management map to this memory model. [Familiarity]
4. Reason about memory leaks, dangling-pointer dereferences, and the benefits and limitations of garbage collection. [Usage]

## PL/Syntax Analysis

## *[Elective]*

***Topics:***

- Scanning (lexical analysis) using regular expressions
- Parsing strategies including top-down (e.g., recursive descent, Earley parsing, or LL) and bottom-up (e.g., backtracking or LR) techniques; role of context-free grammars
- Generating scanners and parsers from declarative specifications

***Learning outcomes:***

1. Use formal grammars to specify the syntax of languages. [Usage]
2. Use declarative tools to generate parsers and scanners. [Usage]
3. Identify key issues in syntax definitions: ambiguity, associativity, precedence. [Familiarity]

## PL/Compiler Semantic Analysis

## *[Elective]*

***Topics:***

- High-level program representations such as abstract syntax trees
- Scope and binding resolution
- Type checking
- Declarative specifications such as attribute grammars

***Learning outcomes:***

1. Implement context-sensitive, source-level static analyses such as type-checkers or resolving identifiers to identify their binding occurrences. [Usage]

## PL/Code Generation

*[Elective]*

***Topics:***

- Instruction selection
- Procedure calls and method dispatching
- Register allocation
- Separate compilation; linking
- Instruction scheduling
- Peephole optimization

***Learning outcomes:***

1. Identify all essential steps for automatically converting source code into assembly or other low-level languages. [Familiarity]
2. Generate the low-level code for calling functions/methods in modern languages. [Usage]
3. Discuss opportunities for optimization introduced by naive translation and approaches for achieving optimization. [Familiarity]

## PL/Runtime Systems

*[Elective]*

***Topics:***

- Target-platform characteristics such as registers, instructions, bytecodes
- Dynamic memory management approaches and techniques: malloc/free, garbage collection (mark-sweep, copying, reference counting), regions (also known as arenas or zones)
- Data layout for objects and activation records
- Just-in-time compilation and dynamic recompilation
- Other features such as class loading, threads, security, etc.

***Learning outcomes:***

1. Compare the benefits of different memory-management schemes, using concepts such as fragmentation, locality, and memory overhead. [Familiarity]
2. Discuss benefits and limitations of automatic memory management. [Familiarity]
3. Identify the services provided by modern language run-time systems. [Familiarity]
4. Discuss advantages, disadvantages, and difficulties of dynamic recompilation. [Familiarity]

# PL/Static Analysis

## *[Elective]*

***Topics:***

- Relevant program representations, such as basic blocks, control-flow graphs, def-use chains, static single assignment, etc.
- Undecidability and consequences for program analysis
- Flow-insensitive analyses, such as type-checking and scalable pointer and alias analyses
- Flow-sensitive analyses, such as forward and backward dataflow analyses
- Path-sensitive analyses, such as software model checking
- Tools and frameworks for defining analyses
- Role of static analysis in program optimization
- Role of static analysis in (partial) verification and bug-finding

***Learning outcomes:***

1. Define useful static analyses in terms of a conceptual framework such as dataflow analysis. [Usage]
2. Communicate why an analysis is correct (sound and terminating). [Usage]
3. Explain why non-trivial sound static analyses must be approximate. [Familiarity]
4. Distinguish “may” and “must” analyses. [Familiarity]
5. Explain why potential aliasing limits sound program analysis and how alias analysis can help. [Familiarity]
6. Use the results of a static analysis for program optimization and/or partial program correctness. [Usage]

# PL/Advanced Programming Constructs

## *[Elective]*

***Topics:***

- Lazy evaluation and infinite streams
- Control Abstractions: Exception Handling, Continuations, Monads
- Object-oriented abstractions: Multiple inheritance, Mixins, Traits, Multimethods
- Metaprogramming: Macros, Generative programming, Model-based development
- Module systems
- String manipulation via pattern-matching (regular expressions)
- Dynamic code evaluation (“eval”)
- Language support for checking assertions, invariants, and pre/post-conditions

***Learning outcomes:***

1. Use various advanced programming constructs and idioms correctly. [Usage]
2. Discuss how various advanced programming constructs aim to improve program structure, software quality, and programmer productivity. [Familiarity]
3. Discuss how various advanced programming constructs interact with the definition and implementation of other language features. [Familiarity]

## PL/Concurrency and Parallelism

### *[Elective]*

Support for concurrency is a fundamental programming-languages issue with rich material in programming language design, language implementation, and language theory. Due to coverage in other Knowledge Areas, this elective Knowledge Unit aims only to complement the material included elsewhere in the body of knowledge. Courses on programming languages are an excellent place to include a general treatment of concurrency including this other material.

(Cross-reference: PD-Parallel and Distributed Computing)

***Topics:***

- Constructs for thread-shared variables and shared-memory synchronization
- Actor models
- Futures
- Language support for data parallelism
- Models for passing messages between sequential processes
- Effect of memory-consistency models on language semantics and correct code generation

***Learning outcomes:***

1. Write correct concurrent programs using multiple programming models. [Usage]
2. Explain why programming languages do not guarantee sequential consistency in the presence of data races and what programmers must do as a result. [Familiarity]

## PL/Type Systems

### *[Elective]*

***Topics:***

- Compositional type constructors, such as product types (for aggregates), sum types (for unions), function types, quantified types, and recursive types
- Type checking
- Type safety as preservation plus progress
- Type inference
- Static overloading

***Learning outcomes:***

1. Define a type system precisely and compositionally. [Usage]
2. For various foundational type constructors, identify the values they describe and the invariants they enforce. [Familiarity]
3. Precisely specify the invariants preserved by a sound type system. [Familiarity]

## PL/Formal Semantics

*[Elective]*

***Topics:***

- Syntax vs. semantics
- Lambda Calculus
- Approaches to semantics: Operational, Denotational, Axiomatic
- Proofs by induction over language semantics
- Formal definitions and proofs for type systems
- Parametricity

***Learning outcomes:***

1. Give a formal semantics for a small language. [Usage]
2. Use induction to prove properties of all (or a well-defined subset of) programs in a language. [Usage]
3. Use language-based techniques to build a formal model of a software system. [Usage]

## PL/Language Pragmatics

*[Elective]*

***Topics:***

- Principles of language design such as orthogonality
- Evaluation order, precedence, and associativity
- Eager vs. delayed evaluation
- Defining control and iteration constructs
- External calls and system libraries

***Learning outcomes:***

1. Discuss the role of concepts such as orthogonality and well-chosen defaults in language design. [Familiarity]
2. Use crisp and objective criteria for evaluating language-design decisions. [Usage]

## PL/Logic Programming

*[Elective]*

***Topics:***

- Clausal representation of data structures and algorithms
- Unification
- Backtracking and search

***Learning outcomes:***

1. Use a logic language to implement conventional algorithms. [Usage]
2. Use a logic language to implement algorithms employing implicit search using clauses and relations. [Usage]

# Software Development Fundamentals (SDF)

Fluency in the process of software development is a prerequisite to the study of most of computer science. In order to use computers to solve problems effectively, students must be competent at reading and writing programs in multiple programming languages. Beyond programming skills, however, they must be able to design and analyze algorithms, select appropriate paradigms, and utilize modern development and testing tools. This knowledge area brings together those fundamental concepts and skills related to the software development process. As such, it provides a foundation for other software-oriented knowledge areas, most notably Programming Languages, Algorithms and Complexity, and Software Engineering.

It is important to note that this knowledge area is distinct from the old Programming Fundamentals knowledge area from CC2001. Whereas that knowledge area focused exclusively on the programming skills required in an introductory computer science course, this new knowledge area is intended to fill a much broader purpose. It focuses on the entire software development process, identifying those concepts and skills that should be mastered in the first year of a computer science program. This includes the design and simple analysis of algorithms, fundamental programming concepts and data structures, and basic software development methods and tools. As a result of its broader purpose, the Software Development Fundamentals knowledge area includes fundamental concepts and skills that could naturally be listed in other software-oriented knowledge areas (e.g., programming constructs from Programming Languages, simple algorithm analysis from Algorithms & Complexity, simple development methodologies from Software Engineering). Likewise, each of these knowledge areas will contain more advanced material that builds upon the fundamental concepts and skills listed here.

While broader in scope than the old Programming Fundamentals, this knowledge area still allows for considerable flexibility in the design of first-year curricula. For example, the Fundamental Programming Concepts unit identifies only those concepts that are common to all programming paradigms. It is expected that an instructor would select one or more programming paradigms (e.g., object-oriented programming, functional programming, scripting) to illustrate these programming concepts, and would pull paradigm-specific content from the Programming Languages knowledge area to fill out a course. Likewise, an instructor could choose to emphasize formal analysis (e.g., Big-Oh, computability) or design methodologies (e.g., team projects, software life cycle) early, thus integrating hours from the Programming Languages, Algorithms and Complexity, and/or Software Engineering knowledge areas. Thus, the 43 hours of material in this knowledge area will typically be augmented with core material from one or more of these knowledge areas to form a complete and coherent first-year experience.

When considering the hours allocated to each knowledge unit, it should be noted that these hours reflect the minimal amount of classroom coverage needed to introduce the material. Many software development topics will reappear and be reinforced by later topics (e.g., applying iteration constructs when processing lists). In addition, the mastery of concepts and skills from this knowledge area requires a significant amount of software development experience outside of class.

## SDF. Software Development Fundamentals (43 Core-Tier1 hours)

| | Core-Tier1 hours | Core-Tier2 hours | Includes Electives |
|---|---|---|---|
| SDF/Algorithms and Design | 11 | | N |
| SDF/Fundamental Programming Concepts | 10 | | N |
| SDF/Fundamental Data Structures | 12 | | N |
| SDF/Development Methods | 10 | | N |

## SDF/Algorithms and Design

### *[11 Core-Tier1 hours]*

This unit builds the foundation for core concepts in the Algorithms & Complexity knowledge area, most notably in the Basic Analysis and Algorithmic Strategies units.

***Topics:***

- The concept and properties of algorithms
  - Informal comparison of algorithm efficiency (e.g., operation counts)
- The role of algorithms in the problem-solving process
- Problem-solving strategies
  - Iterative and recursive mathematical functions
  - Iterative and recursive traversal of data structures
  - Divide-and-conquer strategies
- Fundamental design concepts and principles
  - Abstraction
  - Program decomposition
  - Encapsulation and information hiding
  - Separation of behavior and implementation

***Learning Outcomes:***

1. Discuss the importance of algorithms in the problem-solving process. [Familiarity]
2. Discuss how a problem may be solved by multiple algorithms, each with different properties. [Familiarity]
3. Create algorithms for solving simple problems. [Usage]
4. Use a programming language to implement, test, and debug algorithms for solving simple problems. [Usage]
5. Implement, test, and debug simple recursive functions and procedures. [Usage]
6. Determine whether a recursive or iterative solution is most appropriate for a problem. [Assessment]
7. Implement a divide-and-conquer algorithm for solving a problem. [Usage]
8. Apply the techniques of decomposition to break a program into smaller pieces. [Usage]
9. Identify the data components and behaviors of multiple abstract data types. [Usage]
10. Implement a coherent abstract data type, with loose coupling between components and behaviors. [Usage]
11. Identify the relative strengths and weaknesses among multiple designs or implementations for a problem. [Assessment]

## SDF/Fundamental Programming Concepts

### *[10 Core-Tier1 hours]*

This unit builds the foundation for core concepts in the Programming Languages knowledge area, most notably in the paradigm-specific units: Object-Oriented Programming, Functional Programming, and Event-Driven & Reactive Programming.

***Topics:***

- Basic syntax and semantics of a higher-level language
- Variables and primitive data types (e.g., numbers, characters, Booleans)
- Expressions and assignments
- Simple I/O including file I/O
- Conditional and iterative control structures
- Functions and parameter passing
- The concept of recursion

***Learning Outcomes:***

1. Analyze and explain the behavior of simple programs involving the fundamental programming constructs covered by this unit. [Assessment]
2. Identify and describe uses of primitive data types. [Familiarity]
3. Write programs that use primitive data types. [Usage]
4. Modify and expand short programs that use standard conditional and iterative control structures and functions. [Usage]
5. Design, implement, test, and debug a program that uses each of the following fundamental programming constructs: basic computation, simple I/O, standard conditional and iterative structures, the definition of functions, and parameter passing. [Usage]
6. Write a program that uses file I/O to provide persistence across multiple executions. [Usage]
7. Choose appropriate conditional and iteration constructs for a given programming task. [Assessment]
8. Describe the concept of recursion and give examples of its use. [Familiarity]
9. Identify the base case and the general case of a recursively-defined problem. [Assessment]

## SDF/Fundamental Data Structures

### *[12 Core-Tier1 hours]*

This unit builds the foundation for core concepts in the Algorithms & Complexity knowledge area, most notably in the Fundamental Data Structures & Algorithms and Basic Computability & Complexity units.

***Topics:***

- Arrays
- Records/structs (heterogeneous aggregates)
- Strings and string processing
- Abstract data types and their implementation
  - Stacks
  - Queues
  - Priority queues
  - Sets
  - Maps
- References and aliasing
- Linked lists
- Strategies for choosing the appropriate data structure

***Learning Outcomes:***

1. Discuss the appropriate use of built-in data structures. [Familiarity]
2. Describe common applications for each data structure in the topic list. [Familiarity]
3. Write programs that use each of the following data structures: arrays, strings, linked lists, stacks, queues, sets, and maps. [Usage]
4. Compare alternative implementations of data structures with respect to performance. [Assessment]
5. Compare and contrast the costs and benefits of dynamic and static data structure implementations. [Assessment]
6. Choose the appropriate data structure for modeling a given problem. [Assessment]

## SDF/Development Methods

*[10 Core-Tier1 hours]*

This unit builds the foundation for core concepts in the Software Engineering knowledge area, most notably in the Software Processes, Software Design and Software Evolution units.

***Topics:***

- Program comprehension
- Program correctness
    - Types or errors (syntax, logic, run-time)
    - The concept of a specification
    - Defensive programming (e.g. secure coding, exception handling)
    - Code reviews
    - Testing fundamentals and test-case generation
    - Test-driven development
    - The role and the use of contracts, including pre- and post-conditions
    - Unit testing
- Simple refactoring
- Modern programming environments
    - Code search
    - Programming using library components and their APIs
- Debugging strategies
- Documentation and program style

***Learning Outcomes:***

1. Trace the execution of a variety of code segments and write summaries of their computations. [Assessment]
2. Explain why the creation of correct program components is important in the production of high-quality software. [Familiarity]
3. Identify common coding errors that lead to insecure programs (e.g., buffer overflows, memory leaks, malicious code) and apply strategies for avoiding such errors. [Usage]
4. Conduct a personal code review (focused on common coding errors) on a program component using a provided checklist. [Usage]
5. Contribute to a small-team code review focused on component correctness. [Usage]
6. Describe how a contract can be used to specify the behavior of a program component. [Familiarity]
7. Create a unit test plan for a medium-size code segment. [Usage]
8. Refactor a program by identifying opportunities to apply procedural abstraction. [Usage]
9. Apply a variety of strategies to the testing and debugging of simple programs. [Usage]
10. Construct, execute and debug programs using a modern IDE and associated tools such as unit testing tools and visual debuggers. [Usage]
11. Construct and debug programs using the standard libraries available with a chosen programming language. [Usage]
12. Analyze the extent to which another programmer's code meets documentation and programming style standards. [Assessment]
13. Apply consistent documentation and program style standards that contribute to the readability and maintainability of software. [Usage]

# Software Engineering (SE)

In every computing application domain, professionalism, quality, schedule, and cost are critical to producing software systems. Because of this, the elements of software engineering are applicable to developing software in all areas of computing. A wide variety of software engineering practices have been developed and utilized since the need for a discipline of software engineering was first recognized. Many trade-offs between these different practices have also been identified. Practicing software engineers have to select and apply appropriate techniques and practices to a given development effort to maximize value. To learn how to do this, they study the elements of software engineering.

Software engineering is the discipline concerned with the application of theory, knowledge, and practice to effectively and efficiently build reliable software systems that satisfy the requirements of customers and users. This discipline is applicable to small, medium, and large-scale systems. It encompasses all phases of the lifecycle of a software system, including requirements elicitation, analysis and specification; design; construction; verification and validation; deployment; and operation and maintenance. Whether small or large, following a traditional disciplined development process, an agile approach, or some other method, software engineering is concerned with the best way to build good software systems.

Software engineering uses engineering methods, processes, techniques, and measurements. It benefits from the use of tools for managing software development; analyzing and modeling software artifacts; assessing and controlling quality; and for ensuring a disciplined, controlled approach to software evolution and reuse. The software engineering toolbox has evolved over the years. For instance, the use of contracts, with requires and ensure clauses and class invariants, is one good practice that has become more common. Software development, which can involve an individual developer or a team or teams of developers, requires choosing the most appropriate tools, methods, and approaches for a given development environment.

Students and instructors need to understand the impacts of specialization on software engineering approaches. For example, specialized systems include:

- Real time systems
- Client-server systems
- Distributed systems
- Parallel systems
- Web-based systems
- High integrity systems
- Games
- Mobile computing
- Domain specific software (e.g., scientific computing or business applications)

Issues raised by each of these specialized systems demand specific treatments in each phase of software engineering. Students must become aware of the differences between general software engineering techniques and principles and the techniques and principles needed to address issues specific to specialized systems.

An important effect of specialization is that different choices of material may need to be made when teaching applications of software engineering, such as between different process models, different approaches to modeling systems, or different choices of techniques for carrying out any of the key activities. This is reflected in the assignment of core and elective material, with the core topics and learning outcomes focusing on the principles underlying the various choices, and the details of the various alternatives from which the choices have to be made being assigned to the elective material.

Another division of the practices of software engineering is between those concerned with the fundamental need to develop systems that implement correctly the functionality that is required for them, and those concerned with other qualities for systems and the trade-offs needed to balance these qualities. This division too is reflected in the assignment of core and elective material, so that topics and learning outcomes concerned with the basic methods for developing such system are assigned to the core, and those that are concerned with other qualities and trade-offs between them are assigned to the elective material.

In general, students learn best at the application level much of the material defined in the SE KA by participating in a project. Such projects should require students to work on a team to develop a software system through as much of its lifecycle as is possible. Much of software engineering is devoted to effective communication among team members and stakeholders. Utilizing project teams, projects can be sufficiently challenging to require the use of effective software engineering techniques and that students develop and practice their communication skills. While organizing and running effective projects within the academic framework can be challenging, the best way to learn to apply software engineering theory and knowledge is in the practical environment of a project. The minimum hours specified for some knowledge units in this document may appear insufficient to accomplish associated application-level learning outcomes. It should be understood that these outcomes are to be achieved through project experience that may even occur later in the curriculum than when the topics within the knowledge unit are introduced.

Note: The SDF/Development Methods knowledge unit includes 10 Core-Tier1 hours that constitute an introduction to certain aspects of software engineering. The knowledge units, topics and core hour specifications in this document must be understood as assuming previous exposure to the material described in SDF/Development Methods.

## SE. Software Engineering (6 Core-Tier1 hours; 21 Core-Tier2 hours)

| | Core-Tier1 hours | Core-Tier2 hours | Includes Electives |
|---|---|---|---|
| SE/Software Processes | 2 | 1 | Y |
| SE/Software Project Management | | 2 | Y |
| SE/Tools and Environments | | 2 | N |
| SE/Requirements Engineering | 1 | 3 | Y |
| SE/Software Design | 3 | 5 | Y |
| SE/Software Construction | | 2 | Y |
| SE/Software Verification and Validation | | 3 | Y |
| SE/Software Evolution | | 2 | Y |
| SE/Formal Methods | | | Y |
| SE/Software Reliability | | 1 | Y |

## SE/Software Processes

*[2 Core-Tier1 hours; 1 Core-Tier2 hour]*

***Topics:***

[Core-Tier1]

- Systems level considerations, i.e., the interaction of software with its intended environment
- Introduction to software process models (e.g., waterfall, incremental, agile)
    - Phases of software life-cycles
- Programming in the large vs. individual programming

[Core-Tier2]

- Applying software process models

[Elective]

- Software quality concepts
- Process improvement
- Software process capability maturity models
- Software process measurements

***Learning Outcomes:***

[Core-Tier1]

1. Describe how software can interact with and participate in various systems including information management, embedded, process control, and communications systems. [Familiarity]
2. Describe the difference between principles of the waterfall model and models using iterations. [Familiarity]
3. Describe the different practices that are key components of various process model. [Familiarity]
4. Differentiate among the phases of software development. [Familiarity]
5. Describe how programming in the large differs from individual efforts with respect to understanding a large code base, code reading, understanding builds, and understanding context of changes. [Familiarity]

[Core-Tier2]

6. Explain the concept of a software life cycle and provide an example, illustrating its phases including the deliverables that are produced. [Familiarity]
7. Compare several common process models with respect to their value for development of particular classes of software systems taking into account issues such as requirement stability, size, and non-functional characteristics. [Usage]

[Elective]

8. Define software quality and describe the role of quality assurance activities in the software process. [Familiarity]
9. Describe the intent and fundamental similarities among process improvement approaches. [Familiarity]
10. Compare several process improvement models such as CMM, CMMI, CQI, Plan-Do-Check-Act, or ISO9000. [Familiarity]
11. Use a process improvement model such as PSP to assess a development effort and recommend approaches to improvement. [Usage]
12. Explain the role of process maturity models in process improvement. [Familiarity]
13. Describe several process metrics for assessing and controlling a project. [Familiarity]
14. Use project metrics to describe the current state of a project. [Usage]

## SE/Software Project Management

### *[2 Core-Tier2 hours]*

***Topics:***

[Core-Tier2]

- Team participation
  - Team processes including responsibilities for tasks, meeting structure, and work schedule
  - Roles and responsibilities in a software team
  - Team conflict resolution
  - Risks associated with virtual teams (communication, perception, structure)
- Effort Estimation (at the personal level)
- Risk
  - The role of risk in the life cycle
  - Risk categories including security, safety, market, financial, technology, people, quality, structure and process

[Elective]

- Team management
  - Team organization and decision-making
  - Role identification and assignment

- Individual and team performance assessment
- Project management
    - Scheduling and tracking
    - Project management tools
    - Cost/benefit analysis
- Software measurement and estimation techniques
- Software quality assurance and the role of measurements
- Risk
    - Risk identification and management
    - Risk analysis and evaluation
    - Risk tolerance (e.g., risk-adverse, risk-neutral, risk-seeking)
    - Risk planning
- System-wide approach to risk including hazards associated with tools

***Learning Outcomes:***

[Core-Tier2]

1. Identify behaviors that contribute to the effective functioning of a team. [Familiarity]
2. Create and follow an agenda for a team meeting. [Usage]
3. Identify and justify necessary roles in a software development team. [Usage]
4. Understand the sources, hazards, and potential benefits of team conflict. [Usage]
5. Apply a conflict resolution strategy in a team setting. [Usage]
6. Use an *ad hoc* method to estimate software development effort (e.g., time) and compare to actual effort required. [Usage]
7. List several examples of software risks. [Familiarity]
8. Describe the impact of risk in a software development life cycle. [Familiarity]
9. Describe different categories of risk in software systems. [Familiarity]

[Elective]

10. Identify security risks for a software system. [Usage]
11. Demonstrate through involvement in a team project the central elements of team building and team management. [Usage]
12. Identify several possible team organizational structures and team decision-making processes. [Familiarity]
13. Create a team by identifying appropriate roles and assigning roles to team members. [Usage]
14. Assess and provide feedback to teams and individuals on their performance in a team setting. [Usage]
15. Prepare a project plan for a software project that includes estimates of size and effort, a schedule, resource allocation, configuration control, change management, and project risk identification and management. [Usage]
16. Track the progress of a project using appropriate project metrics. [Usage]
17. Compare simple software size and cost estimation techniques. [Usage]
18. Use a project management tool to assist in the assignment and tracking of tasks in a software development project. [Usage]
19. Describe the impact of risk tolerance on the software development process. [Assessment]
20. Identify risks and describe approaches to managing risk (avoidance, acceptance, transference, mitigation), and characterize the strengths and shortcomings of each. [Familiarity]
21. Explain how risk affects decisions in the software development process. [Usage]
22. Demonstrate a systematic approach to the task of identifying hazards and risks in a particular situation. [Usage]
23. Apply the basic principles of risk management in a variety of simple scenarios including a security situation. [Usage]
24. Conduct a cost/benefit analysis for a risk mitigation approach. [Usage]
25. Identify and analyze some of the risks for an entire system that arise from aspects other than the software. [Usage]

## SE/Tools and Environments

### *[2 Core-Tier2 hours]*

***Topics:***

[Core-Tier2]

- Software configuration management and version control; release management
- Requirements analysis and design modeling tools
- Testing tools including static and dynamic analysis tools
- Programming environments that automate parts of program construction processes (e.g., automated builds)
    - Continuous integration
- Tool integration concepts and mechanisms

***Learning Outcomes:***

[Core-Tier2]

1. Describe the difference between centralized and distributed software configuration management. [Familiarity]
2. Identify configuration items and use a source code control tool in a small team-based project. [Usage]
3. Describe the issues that are important in selecting a set of tools for the development of a particular software system, including tools for requirements tracking, design modeling, implementation, build automation, and testing. [Familiarity]
4. Demonstrate the capability to use software tools in support of the development of a software product of medium size. [Usage]

## SE/Requirements Engineering

### *[1 Core-Tier1 hour; 3 Core-Tier2 hours]*

***Topics:***

[Core-Tier1]

- Properties of requirements including consistency, validity, completeness, and feasibility
- Describing functional requirements using, for example, use cases or users stories

[Core-Tier2]

- Software requirements elicitation
- Non-functional requirements and their relationship to software quality
- Describing system data using, for example, class diagrams or entity-relationship diagrams
- Evaluation and use of requirements specifications

[Elective]

- Requirements analysis modeling techniques
- Acceptability of certainty / uncertainty considerations regarding software / system behavior
- Prototyping
- Basic concepts of formal requirements specification
- Requirements specification
- Requirements validation
- Requirements tracing

***Learning Outcomes:***

[Core-Tier1]

1. List the key components of a use case or similar description of some behavior that is required for a system and discuss their role in the requirements engineering process. [Familiarity]
2. Interpret a given requirements model for a simple software system. [Familiarity]
3. Conduct a review of a set of software requirements to determine the quality of the requirements with respect to the characteristics of good requirements. [Usage]

[Core-Tier2]

4. Describe the fundamental challenges of and common techniques used for requirements elicitation. [Familiarity]
5. List the key components of a class diagram or similar description of the data that a system is required to handle. [Familiarity]
6. Identify both functional and non-functional requirements in a given requirements specification for a software system. [Usage]

[Elective]

7. Apply key elements and common methods for elicitation and analysis to produce a set of software requirements for a medium-sized software system. [Usage]
8. Use a common, non-formal method to model and specify (in the form of a requirements specification document) the requirements for a medium-size software system [Usage]
9. Translate into natural language a software requirements specification (e.g., a software component contract) written in a formal specification language. [Usage]
10. Create a prototype of a software system to mitigate risk in requirements. [Usage]
11. Differentiate between forward and backward tracing and explain their roles in the requirements validation process. [Familiarity]

## SE/Software Design

### *[3 Core-Tier1 hours; 5 Core-Tier2 hours]*

***Topics:***

[Core-Tier1]

- Overview of design paradigms
- System design principles: divide and conquer (architectural design and detailed design), separation of concerns, information hiding, coupling and cohesion, re-use of standard structures.
- Appropriate models of software designs, including structure and behavior.

[Core-Tier2]

- Design Paradigms such as structured design (top-down functional decomposition), object-oriented analysis and design, event driven design, component-level design, data-structured centered, aspect oriented, function oriented, service oriented.
- Relationships between requirements and designs: transformation of models, design of contracts, invariants.
- Software architecture concepts and standard architectures (e.g. client-server, n-layer, transform centered, pipes-and-filters, etc).
- Refactoring designs and the use of design patterns.
- The use of components in design: component selection, design, adaptation and assembly of components, components and patterns, components and objects, (for example, build a GUI using a standard widget set).

[Elective]

- Internal design qualities, and models for them: efficiency and performance, redundancy and fault tolerance, traceability of requirements.
- External design qualities, and models for them: functionality, reliability, performance and efficiency, usability, maintainability, portability.
- Measurement and analysis of design quality.
- Tradeoffs between different aspects of quality.
- Application frameworks.
- Middleware: the object-oriented paradigm within middleware, object request brokers and marshalling, transaction processing monitors, workflow systems.

***Learning Outcomes:***

[Core-Tier1]

1. Articulate design principles including separation of concerns, information hiding, coupling and cohesion, and encapsulation. [Familiarity]
2. Use a design paradigm to design a simple software system, and explain how system design principles have been applied in this design. [Usage]
3. Construct models of the design of a simple software system that are appropriate for the paradigm used to design it. [Usage]
4. For the design of a simple software system within the context of a single design paradigm, describe the software architecture of that system. [Familiarity]
5. Within the context of a single design paradigm, describe one or more design patterns that could be applicable to the design of a simple software system. [Familiarity]

[Core-Tier2]

6. For a simple system suitable for a given scenario, discuss and select an appropriate design paradigm. [Usage]
7. Create appropriate models for the structure and behavior of software products from their requirements specifications. [Usage]
8. Explain the relationships between the requirements for a software product and the designed structure and behavior, in terms of the appropriate models and transformations of them. [Assessment]
9. Apply simple examples of patterns in a software design. [Usage]
10. Given a high-level design, identify the software architecture by differentiating among common software architectures such as 3-tier, pipe-and-filter, and client-server. [Familiarity]
11. Investigate the impact of software architectures selection on the design of a simple system. [Assessment]
12. Select suitable components for use in the design of a software product. [Usage]
13. Explain how suitable components might need to be adapted for use in the design of a software product. [Familiarity].
14. Design a contract for a typical small software component for use in a given system. [Usage]

[Elective]

15. Discuss and select appropriate software architecture for a simple system suitable for a given scenario. [Usage]
16. Apply models for internal and external qualities in designing software components to achieve an acceptable tradeoff between conflicting quality aspects. [Usage]
17. Analyze a software design from the perspective of a significant internal quality attribute. [Assessment]
18. Analyze a software design from the perspective of a significant external quality attribute. [Assessment]
19. Explain the role of objects in middleware systems and the relationship with components. [Familiarity]
20. Apply component-oriented approaches to the design of a range of software, such as using components for concurrency and transactions, for reliable communication services, for database interaction including services for remote query and database management, or for secure communication and access. [Usage]

## SE/Software Construction

*[2 Core-Tier2 hours]*

***Topics:***

[Core-Tier2]

- Coding practices: techniques, idioms/patterns, mechanisms for building quality programs
  - Defensive coding practices
  - Secure coding practices
  - Using exception handling mechanisms to make programs more robust, fault-tolerant
- Coding standards
- Integration strategies
- Development context: "green field" vs. existing code base
  - Change impact analysis
  - Change actualization

[Elective]

- Robust and Security Enhanced Programming
  - Defensive programming
  - Principles of secure design and coding:
  - Principle of least privilege
  - Principle of fail-safe defaults
  - Principle of psychological acceptability
- Potential security problems in programs
  - Buffer and other types of overflows
  - Race conditions
  - Improper initialization, including choice of privileges
  - Checking input
  - Assuming success and correctness
  - Validating assumptions
- Documenting security considerations in using a program

***Learning Outcomes:***

[Core-Tier2]

1. Describe techniques, coding idioms and mechanisms for implementing designs to achieve desired properties such as reliability, efficiency, and robustness. [Familiarity]
2. Build robust code using exception handling mechanisms. [Usage]
3. Describe secure coding and defensive coding practices. [Familiarity]
4. Select and use a defined coding standard in a small software project. [Usage]
5. Compare and contrast integration strategies including top-down, bottom-up, and sandwich integration. [Familiarity]
6. Describe the process of analyzing and implementing changes to code base developed for a specific project. [Familiarity]
7. Describe the process of analyzing and implementing changes to a large existing code base. [Familiarity]

[Elective]

8. Rewrite a simple program to remove common vulnerabilities, such as buffer overflows, integer overflows and race conditions [Usage]
9. State and apply the principles of least privilege and fail-safe defaults. [Familiarity]

10. Write a simple library that performs some non-trivial task and will not terminate the calling program regardless of how it is called [Usage]

## SE/Software Verification and Validation

### *[3 Core-Tier2 hours]*

***Topics:***

[Core-Tier2]

- Verification and validation concepts
- Inspections, reviews, audits
- Testing types, including human computer interface, usability, reliability, security, conformance to specification
- Testing fundamentals
    - Unit, integration, validation, and system testing
    - Test plan creation and test case generation
    - Black-box and white-box testing techniques
- Defect tracking
- Testing parallel and distributed systems

[Elective]

- Static approaches and dynamic approaches to verification
- Regression testing
- Test-driven development
- Validation planning; documentation for validation
- Object-oriented testing; systems testing
- Verification and validation of non-code artifacts (documentation, help files, training materials)
- Fault logging, fault tracking and technical support for such activities
- Fault estimation and testing termination including defect seeding

***Learning Outcomes:***

[Core-Tier2]

1. Distinguish between program validation and verification. [Familiarity]
2. Describe the role that tools can play in the validation of software. [Familiarity]
3. Undertake, as part of a team activity, an inspection of a medium-size code segment. [Usage]
4. Describe and distinguish among the different types and levels of testing (unit, integration, systems, and acceptance). [Familiarity]
5. Describe techniques for identifying significant test cases for unit, integration, and system testing. [Familiarity]
6. Use a defect tracking tool to manage software defects in a small software project. [Usage]
7. Describe the issues and approaches to testing parallel and distributed systems. [Familiarity]

[Elective]

8. Create, evaluate, and implement a test plan for a medium-size code segment. [Usage]
9. Compare static and dynamic approaches to verification. [Familiarity]
10. Discuss the issues involving the testing of object-oriented software. [Usage]
11. Describe techniques for the verification and validation of non-code artifacts. [Familiarity]
12. Describe approaches for fault estimation. [Familiarity]
13. Estimate the number of faults in a small software application based on fault density and fault seeding. [Usage]
14. Conduct an inspection or review of software source code for a small or medium sized software project. [Usage]

## SE/Software Evolution

### *[2 Core-Tier2 hour]*

***Topics:***

[Core-Tier2]

- Software development in the context of large, pre-existing code bases
    - Software change
    - Concerns and concern location
    - Refactoring
- Software evolution
- Characteristics of maintainable software
- Reengineering systems
- Software reuse

***Learning Outcomes:***

[Core-Tier2]

1. Identify the principal issues associated with software evolution and explain their impact on the software life cycle. [Familiarity]
2. Estimate the impact of a change request to an existing product of medium size. [Usage]
3. Identify weaknesses in a given simple design, and removed them through refactoring. [Usage]
4. Discuss the challenges of evolving systems in a changing environment. [Familiarity]
5. Outline the process of regression testing and its role in release management. [Usage]
6. Discuss the advantages and disadvantages of software reuse. [Familiarity]

## SE/Formal Methods

### *[Elective]*

The topics listed below have a strong dependency on core material from the Discrete Structures area, particularly knowledge units DS/Functions Relations And Sets, DS/Basic Logic and DS/Proof Techniques.

***Topics*:**

- Role of formal specification and analysis techniques in the software development cycle

- Program assertion languages and analysis approaches (including languages for writing and analyzing pre- and post-conditions, such as OCL, JML)
- Formal approaches to software modeling and analysis
    - Model checkers
    - Model finders
- Tools in support of formal methods

***Learning Outcomes:***

1. Describe the role formal specification and analysis techniques can play in the development of complex software and compare their use as validation and verification techniques with testing. [Familiarity]
2. Apply formal specification and analysis techniques to software designs and programs with low complexity. [Usage]
3. Explain the potential benefits and drawbacks of using formal specification languages. [Familiarity]
4. Create and evaluate program assertions for a variety of behaviors ranging from simple through complex. [Usage]
5. Using a common formal specification language, formulate the specification of a simple software system and derive examples of test cases from the specification. [Usage]

## SE/Software Reliability

### *[1 Core-Tier2]*

***Topics:***

[Core-Tier2]

- Software reliability engineering concepts
- Software reliability, system reliability and failure behavior (cross-reference SF9/Reliability Through Redundancy)
- Fault lifecycle concepts and techniques

[Elective]

- Software reliability models
- Software fault tolerance techniques and models
- Software reliability engineering practices
- Measurement-based analysis of software reliability

***Learning Outcomes:***

[Core-Tier2]

1. Explain the problems that exist in achieving very high levels of reliability. [Familiarity]
2. Describe how software reliability contributes to system reliability [Familiarity]
3. List approaches to minimizing faults that can be applied at each stage of the software lifecycle. [Familiarity]

[Elective]

4. Compare the characteristics of three different reliability modeling approaches. [Familiarity]
5. Demonstrate the ability to apply multiple methods to develop reliability estimates for a software system. [Usage]
6. Identify methods that will lead to the realization of a software architecture that achieves a specified reliability level of reliability. [Usage]

7. Identify ways to apply redundancy to achieve fault tolerance for a medium-sized application. [Usage]

# Systems Fundamentals (SF)

The underlying hardware and software infrastructure upon which applications are constructed is collectively described by the term "computer systems." Computer systems broadly span the sub-disciplines of operating systems, parallel and distributed systems, communications networks, and computer architecture. Traditionally, these areas are taught in a non-integrated way through independent courses. However these sub-disciplines increasingly share important common fundamental concepts within their respective cores. These concepts include computational paradigms, parallelism, cross-layer communications, state and state transition, resource allocation and scheduling, and so on. This knowledge area is designed to present an integrative view of these fundamental concepts in a unified albeit simplified fashion, providing a common foundation for the different specialized mechanisms and policies appropriate to the particular domain area.

## SF. Systems Fundamentals [18 core Tier 1, 9 core Tier 2 hours, 27 total]

| | **Core-Tier 1 hours** | **Core-Tier 2 hours** | **Includes Electives** |
|---|---|---|---|
| **SF/Computational Paradigms** | **3** | | **N** |
| **SF/Cross-Layer Communications** | **3** | | **N** |
| **SF/State-State Transition-State Machines** | **6** | | **N** |
| **SF/Parallelism** | **3** | | **N** |
| **SF/Evaluation** | **3** | | **N** |
| **SF/Resource Allocation and Scheduling** | | **2** | **N** |
| **SF/Proximity** | | **3** | **N** |
| **SF/Virtualization and Isolation** | | **2** | **N** |
| **SF/Reliability through Redundancy** | | **2** | **N** |
| **SF/Quantitative Evaluation** | | | **Y** |

## SF/Computational Paradigms

### *[3 Core-Tier 1 hours]*

[Cross-reference PD/parallelism fundamentals: The view presented here is the multiple representations of a system across layers, from hardware building blocks to application components, and the parallelism available in each representation; PD/parallelism fundamentals focuses on the application structuring concepts for parallelism.]

***Topics***:

- Basic building blocks and components of a computer (gates, flip-flops, registers, interconnections; Datapath + Control + Memory)
- Hardware as a computational paradigm: Fundamental logic building blocks (logic gates, flip-flops, counters, registers, PL); Logic expressions, minimization, sum of product forms
- Application-level sequential processing: single thread
- Simple application-level parallel processing: request level (web services/client-server/distributed), single thread per server, multiple threads with multiple servers
- Basic concept of pipelining, overlapped processing stages
- Basic concept of scaling: going faster vs. handling larger problems

***Learning Outcomes:***

[Core-Tier1]

1. List commonly encountered patterns of how computations are organized. [Familiarity]
2. Describe the basic building blocks of computers and their role in the historical development of computer architecture. [Familiarity]
3. Articulate the differences between single thread vs. multiple thread, single server vs. multiple server models, motivated by real world examples (e.g., cooking recipes, lines for multiple teller machines, couple shopping for food, wash-dry-fold, etc.). [Familiarity]
4. Articulate the concept of strong vs. weak scaling, i.e., how performance is affected by scale of problem vs. scale of resources to solve the problem. This can be motivated by the simple, real-world examples. [Familiarity]
5. Design a simple logic circuit using the fundamental building blocks of logic design. [Usage]
6. Use tools for capture, synthesis, and simulation to evaluate a logic design. [Usage]
7. Write a simple sequential problem and a simple parallel version of the same program. [Usage]
8. Evaluate performance of simple sequential and parallel versions of a program with different problem sizes, and be able to describe the speed-ups achieved. [Assessment]

## SF/Cross-Layer Communications

[Conceptual presentation here, practical experience in programming these abstractions in PD, NC, OS]

### *[3 Core-Tier 1 hours]*

***Topics***:

- Programming abstractions, interfaces, use of libraries
- Distinction between Application and OS services, Remote Procedure Call
- Application-Virtual Machine Interaction
- Reliability

***Learning Outcomes:***

[Core-Tier1]

1. Describe how computing systems are constructed of layers upon layers, based on separation of concerns, with well-defined interfaces, hiding details of low layers from the higher layers. This can be motivated by real-world systems, like how a car works, or libraries. [Familiarity]
2. Recognize that hardware, VM, OS, application are additional layers of interpretation/processing. [Familiarity]
3. Describe the mechanisms of how errors are detected, signaled back, and handled through the layers. [Familiarity]
4. Construct a simple program using methods of layering, error detection and recovery, and reflection of error status across layers. [Usage]
5. Find bugs in a layered program by using tools for program tracing, single stepping, and debugging. [Usage]

## SF/State-State Transition-State Machines

### *[6 Core-Tier 1 hours]*

[Cross-reference AL/Basic Computability and Complexity, OS/State and State diagrams, NC/Protocols]

***Topics:***

- Digital vs. Analog/Discrete vs. Continuous Systems
- Simple logic gates, logical expressions, Boolean logic simplification
- Clocks, State, Sequencing
- Combinational Logic, Sequential Logic, Registers, Memories
- Computers and Network Protocols as examples of State Machines

***Learning Outcomes:***

[Core-Tier1]

1. Describe computations as a system with a known set of configurations, and a byproduct of the computation is to transition from one unique configuration (state) to another (state). [Familiarity]
2. Recognize the distinction between systems whose output is only a function of their input (Combinational) and those with memory/history (Sequential). [Familiarity]
3. Describe a computer as a state machine that interprets machine instructions. [Familiarity]
4. Explain how a program or network protocol can also be expressed as a state machine, and that alternative representations for the same computation can exist. [Familiarity]
5. Develop state machine descriptions for simple problem statement solutions (e.g., traffic light sequencing, pattern recognizers). [Usage]
6. Derive time-series behavior of a state machine from its state machine representation. [Assessment]

## SF/Parallelism

### *[3 Core-Tier1 hours]*

[Cross-reference: PD/Parallelism Fundamentals]

***Topics:***

- Sequential vs. parallel processing

- Parallel programming (e.g., synchronization for producer-consumer for performance improvement) vs. concurrent programming (e.g., mutual exclusion/atomic operations for reactive programs)
- Request parallelism (e.g., web services) vs. Task parallelism (map-reduce processing)
- Client-Server/Web Services, Thread (Fork-Join), Pipelining
- Multicore architectures and hardware support for synchronization

***Learning Outcomes**:*

[Core-Tier1]

1. For a given program, distinguish between its sequential and parallel execution, and the performance implications thereof. [Familiarity]
2. Demonstrate on an execution time line that parallelism events and operations can take place simultaneously (i.e., at the same time). Explain how work can be performed in less elapsed time if this can be exploited. [Familiarity]
3. Explain other uses of parallelism, such as for reliability/redundancy of execution. [Familiarity]
4. Define the differences between the concepts of Instruction Parallelism, Data Parallelism, Thread Parallelism/Multitasking, Task/Request Parallelism. [Familiarity]
5. Write more than one parallel program (e.g., one simple parallel program in more than one parallel programming paradigm; a simple parallel program that manages shared resources through synchronization primitives; a simple parallel program that performs simultaneous operation on partitioned data through task parallel (e.g., parallel search terms; a simple parallel program that performs step-by-step pipeline processing through message passing). [Usage]
6. Use performance tools to measure speed-up achieved by parallel programs in terms of both problem size and number of resources. [Assessment]

## SF/Evaluation

### *[3 Core-Tier 1 hours]*

[Cross-reference PD/Parallel Performance]

***Topics**:*

- Choosing and understanding performance figures of merit (e.g., speed of execution, energy consumption, bandwidth vs. latency, resource cost)
- Choosing and understanding workloads and representative benchmarks (e.g., SPEC, Dhrystone), and methods of collecting and analyzing performance figures of merit
- CPI equation (Execution time = # of instructions * cycles/instruction * time/cycle) as tool for understanding tradeoffs in the design of instruction sets, processor pipelines, and memory system organizations.
- Amdahl's Law: the part of the computation that cannot be sped up limits the effect of the parts that can

***Learning Outcomes**:*

[Core-Tier1]

1. Explain how the components of system architecture contribute to improving its performance. [Familiarity]
2. Describe Amdahl's law and discuss its limitations. [Familiarity]
3. Design and conduct a performance-oriented experiment, e.g., benchmark a parallel program with different data sets in order to iteratively improve its performance. [Usage]
4. Use software tools to profile and measure program performance. [Assessment]

## SF/Resource Allocation and Scheduling

### *[2 Core-Tier 2 hours]*

***Topics:***

- Kinds of resources: processor share, memory, disk, net bandwidth
- Kinds of scheduling: first-come, priority
- Advantages of fair scheduling, preemptive scheduling

***Learning Outcomes:***

[Core-Tier2]

1. Define how finite computer resources (e.g., processor share, memory, storage and network bandwidth) are managed by their careful allocation to existing entities. [Familiarity]
2. Describe the scheduling algorithms by which resources are allocated to competing entities, and the figures of merit by which these algorithms are evaluated, such as fairness. [Familiarity]
3. Implement simple schedule algorithms. [Usage]
4. Measure figures of merit of alternative scheduler implementations. [Assessment]

## SF/Proximity

### *[3 Core-Tier 2 hours]*

[Cross-reference: AR/Memory Management, OS/Virtual Memory]

***Topics:***

- Speed of light and computers (one foot per nanosecond vs. one GHz clocks)
- Latencies in computer systems: memory vs. disk latencies vs. across the network memory
- Caches, spatial and temporal locality, in processors and systems
- Caches, cache coherency in database, operating systems, distributed systems, and computer architecture
- Introduction into the processor memory hierarchy: registers and multi-level caches, and the formula for average memory access time

***Learning Outcomes:***

[Core-Tier2]

1. Explain the importance of locality in determining performance. [Familiarity]
2. Describe why things that are close in space take less time to access. [Familiarity]
3. Calculate average memory access time and describe the tradeoffs in memory hierarchy performance in terms of capacity, miss/hit rate, and access time. [Assessment]

## SF/Virtualization and Isolation

### *[2 Core-Tier 2 hours]*

***Topics:***

- Rationale for protection and predictable performance
- Levels of indirection, illustrated by virtual memory for managing physical memory resources
- Methods for implementing virtual memory and virtual machines

***Learning Outcomes:***

[Core-Tier2]

1. Explain why it is important to isolate and protect the execution of individual programs and environments that share common underlying resources, including the processor, memory, storage, and network access. [Familiarity]
2. Describe how the concept of indirection can create the illusion of a dedicated machine and its resources even when physically shared among multiple programs and environments. [Familiarity]
3. Measure the performance of two application instances running on separate virtual machines, and determine the effect of performance isolation. [Assessment]

## SF/Reliability through Redundancy

## *[2 Core-Tier 2 hours]*

***Topics**:*

- Distinction between bugs and faults
- How errors increase the longer the distance between the communicating entities; the end-to-end principle as it applies to systems and networks
- Redundancy through check and retry
- Redundancy through redundant encoding (error correcting codes, CRC, FEC)
- Duplication/mirroring/replicas
- Other approaches to fault tolerance and availability

***Learning Outcomes**:*

[Core-Tier2]

1. Explain the distinction between program errors, system errors, and hardware faults (e.g., bad memory) and exceptions (e.g., attempt to divide by zero). [Familiarity]
2. Articulate the distinction between detecting, handling, and recovering from faults, and the methods for their implementation. [Familiarity]
3. Describe the role of error correcting codes in providing error checking and correction techniques in memories, storage, and networks. [Familiarity]
4. Apply simple algorithms for exploiting redundant information for the purposes of data correction. [Usage]
5. Compare different error detection and correction methods for their data overhead, implementation complexity, and relative execution time for encoding, detecting, and correcting errors. [Assessment]

## SF/Quantitative Evaluation

## *[Elective]*

***Topics**:*

- Analytical tools to guide quantitative evaluation
- Order of magnitude analysis (Big O notation)
- Analysis of slow and fast paths of a system
- Events on their effect on performance (e.g., instruction stalls, cache misses, page faults)
- Understanding layered systems, workloads, and platforms, their implications for performance, and the challenges they represent for evaluation
- Microbenchmarking pitfalls

***Learning Outcomes***:

[Elective]

1. Explain the circumstances in which a given figure of system performance metric is useful. [Familiarity]
2. Explain the inadequacies of benchmarks as a measure of system performance. [Familiarity]
3. Use limit studies or simple calculations to produce order-of-magnitude estimates for a given performance metric in a given context. [Usage]
4. Conduct a performance experiment on a layered system to determine the effect of a system parameter on figure of system performance. [Assessment]

# Social Issues and Professional Practice (SP)

While technical issues are central to the computing curriculum, they do not constitute a complete educational program in the field. Students must also be exposed to the larger societal context of computing to develop an understanding of the relevant social, ethical, legal and professional issues. This need to incorporate the study of these non-technical issues into the ACM curriculum was formally recognized in 1991, as can be seen from the following excerpt [Tucker91]:

> *Undergraduates also need to understand the basic cultural, social, legal, and ethical issues inherent in the discipline of computing. They should understand where the discipline has been, where it is, and where it is heading. They should also understand their individual roles in this process, as well as appreciate the philosophical questions, technical problems, and aesthetic values that play an important part in the development of the discipline.*
>
> *Students also need to develop the ability to ask serious questions about the social impact of computing and to evaluate proposed answers to those questions. Future practitioners must be able to anticipate the impact of introducing a given product into a given environment. Will that product enhance or degrade the quality of life? What will the impact be upon individuals, groups, and institutions?*
>
> *Finally, students need to be aware of the basic legal rights of software and hardware vendors and users, and they also need to appreciate the ethical values that are the basis for those rights. Future practitioners must understand the responsibility that they will bear, and the possible consequences of failure. They must understand their own limitations as well as the limitations of their tools. All practitioners must make a long-term commitment to remaining current in their chosen specialties and in the discipline of computing as a whole.*

As technological advances continue to significantly impact the way we live and work, the critical importance of these social and professional issues continues to increase; new computer-based products and venues pose ever more challenging problems each year. It is our students who must enter the workforce and academia with intentional regard for the identification and resolution of these problems.

Computer science educators may opt to deliver this core and elective material in stand-alone courses, integrated into traditional technical and theoretical courses, or as special units in capstone and professional practice courses. The material in this familiarity area is best covered through a combination of one required course along with short modules in other courses. On the one hand, some units listed as core tier-1—in particular, Social Context, Analytical Tools, Professional Ethics, and Intellectual Property—do not readily lend themselves to being covered in other traditional courses. Without a standalone course, it is difficult to cover these topics appropriately. On the other hand, if ethical and social considerations are covered only in the standalone course and not "in context," it will reinforce the false notion that technical processes are void of these other relevant issues. Because of this broad relevance, it is important that several traditional courses include modules that analyze the ethical, social and professional considerations in the context of the technical subject matter of the course. Courses in areas such as software engineering, databases, computer networks, computer security, and introduction to computing provide obvious context for analysis of ethical issues. However, an ethics-related module could be developed for almost any course in the curriculum. It would be explicitly against the spirit of the recommendations to have only a standalone course. Running through all of the issues in this area is the need to speak to the computer practitioner's responsibility to proactively address these issues by both moral and technical actions. The ethical issues discussed in any class should be directly related to and arise naturally from the subject matter of that class. Examples include a discussion in the database course of data aggregation or data mining, or a discussion in the software engineering course of the potential conflicts between obligations to the customer and obligations to the user and others affected by their work. Programming assignments built around applications such as controlling the movement of a laser during eye surgery can help to address the professional, ethical and social impacts of computing. Computing faculty who are unfamiliar with the content and/or pedagogy of applied ethics are urged to take advantage of the considerable resources from ACM, IEEE-CS, SIGCAS (special interest group on computers and society), and other organizations.

It should be noted that the application of ethical analysis underlies every subsection of this Social and Professional knowledge area in computing. The ACM Code of Ethics and Professional Conduct - www.acm.org/about/code-of-ethics - provide guidelines that serve as the basis for the conduct of our professional work. The General Moral Imperatives provide an understanding of our commitment to personal responsibility, professional conduct, and our leadership roles.

## SP. Social Issues and Professional Practice [11 Core-Tier1 hours, 5 Core-Tier2 hours]

| | Core-Tier1 hours | Core-Tier2 hours | Includes Electives |
|---|---|---|---|
| SP/Social Context | 1 | 2 | N |
| SP/Analytical Tools | 2 | | N |
| SP/Professional Ethics | 2 | 2 | N |
| SP/Intellectual Property | 2 | | Y |
| SP/Privacy and Civil Liberties | 2 | | Y |
| SP/Professional Communication | 1 | | Y |
| SP/Sustainability | 1 | 1 | Y |
| SP/History | | | Y |
| SP/Economies of Computing | | | Y |
| SP/Security Policies, Laws and Computer Crimes | | | Y |

## SP/Social Context

*[1 Core-Tier1 hour, 2 Core-Tier2 hours]*

Computers and the Internet, perhaps more than any other technology, have transformed society over the past 50 years, with dramatic increases in human productivity; an explosion of options for news, entertainment, and communication; and fundamental breakthroughs in almost every branch of science and engineering.

***Topics:***

[Core-Tier1]

- Social implications of computing in a networked world (cross-reference HCI/Foundations/social models; IAS/Fundamental Concepts/social issues)
- Impact of social media on individualism, collectivism and culture.

[Core-Tier2]

- Growth and control of the Internet (cross-reference NC/Introduction/organization of the Internet)
- Often referred to as the digital divide, differences in access to digital technology resources and its resulting ramifications for gender, class, ethnicity, geography, and/or underdeveloped countries.

- Accessibility issues, including legal requirements
- Context-aware computing (cross-reference HC/Design for non-mouse interfaces/ ubiquitous and context-aware)

***Learning Outcomes:***

[Core-Tier1]

1. Describe positive and negative ways in which computer technology (networks, mobile computing, cloud computing) alters modes of social interaction at the personal level. [Familiarity]
2. Identify developers' assumptions and values embedded in hardware and software design, especially as they pertain to usability for diverse populations including under-represented populations and the disabled. [Familiarity]
3. Interpret the social context of a given design and its implementation. [Familiarity]
4. Evaluate the efficacy of a given design and implementation using empirical data. [Assessment]
5. Investigate the implications of social media on individualism versus collectivism and culture. [Usage]

[Core-Tier2]

6. Discuss how Internet access serves as a liberating force for people living under oppressive forms of government; explain how limits on Internet access are used as tools of political and social repression. [Familiarity]
7. Analyze the pros and cons of reliance on computing in the implementation of democracy (e.g. delivery of social services, electronic voting). [Assessment]
8. Describe the impact of the under-representation of diverse populations in the computing profession (e.g., industry culture, product diversity). [Familiarity]
9. Investigate the implications of context awareness in ubiquitous computing systems. [Usage]

## SP/Analytical Tools

### *[2 Core-Tier1 hours]*

Ethical theories and principles are the foundations of ethical analysis because they are the viewpoints from which guidance can be obtained along the pathway to a decision. Each theory emphasizes different points such as predicting the outcome and following one's duties to others in order to reach an ethically guided decision. However, in order for an ethical theory to be useful, the theory must be directed towards a common set of goals. Ethical principles are the common goals that each theory tries to achieve in order to be successful. These goals include beneficence, least harm, respect for autonomy and justice.

***Topics:***

[Core-Tier1]

- Ethical argumentation
- Ethical theories and decision-making
- Moral assumptions and values

***Learning Outcomes:***

[Core-Tier1]

1. Evaluate stakeholder positions in a given situation. [Assessment]
2. Analyze basic logical fallacies in an argument. [Assessment]
3. Analyze an argument to identify premises and conclusion. [Assessment]
4. Illustrate the use of example and analogy in ethical argument. [Usage]
5. Evaluate ethical/social tradeoffs in technical decisions. [Assessment]

## SP/Professional Ethics

### *[2 Core-Tier1 hours, 2 Core-Tier2 hours]*

Computer ethics is a branch of practical philosophy which deals with how computing professionals should make decisions regarding professional and social conduct. There are three primary influences: 1) The individual's own personal code, 2) Any informal code of ethical behavior existing in the work place, and 3) Exposure to formal codes of ethics.

***Topics:***

[Core-Tier1]

- Community values and the laws by which we live
- The nature of professionalism including care, attention and discipline, fiduciary responsibility, and mentoring
- Keeping up-to-date as a professional in terms of familiarity, tools, skills, legal and professional framework as well as the ability to self-assess and computer fluency
- Professional certification, codes of ethics, conduct, and practice, such as the ACM/IEEE-CS, SE, AITP, IFIP and international societies (cross-reference IAS/Fundamental Concepts/ethical issues)
- Accountability, responsibility and liability (e.g. software correctness, reliability and safety, as well as ethical confidentiality of cybersecurity professionals)

[Core-Tier2]

- The role of the professional in public policy
- Maintaining awareness of consequences
- Ethical dissent and whistle-blowing
- Dealing with harassment and discrimination
- Forms of professional credentialing
- Acceptable use policies for computing in the workplace
- Ergonomics and healthy computing environments
- Time to market and cost considerations versus quality professional standards

***Learning Outcomes:***

[Core-Tier1]

1. Identify ethical issues that arise in software development and determine how to address them technically and ethically. [Familiarity]
2. Recognize the ethical responsibility of ensuring software correctness, reliability and safety. [Familiarity]
3. Describe the mechanisms that typically exist for a professional to keep up-to-date. [Familiarity]
4. Describe the strengths and weaknesses of relevant professional codes as expressions of professionalism and guides to decision-making. [Familiarity]

5. Analyze a global computing issue, observing the role of professionals and government officials in managing this problem. [Assessment]
6. Evaluate the professional codes of ethics from the ACM, the IEEE Computer Society, and other organizations. [Assessment]

[Core-Tier2]

7. Describe ways in which professionals may contribute to public policy. [Familiarity]
8. Describe the consequences of inappropriate professional behavior. [Familiarity]
9. Identify progressive stages in a whistle-blowing incident. [Familiarity]
10. Investigate forms of harassment and discrimination and avenues of assistance [Usage]
11. Examine various forms of professional credentialing [Usage]
12. Identify the social implications of ergonomic devices and the workplace environment to people's health. [Familiarity]
13. Develop a computer usage/acceptable use policy with enforcement measures. [Assessment]
14. Describe issues associated with industries' push to focus on time to market versus enforcing quality professional standards [Familiarity]

## SP/ Intellectual Property

### *[2 Core-Tier1 hours]*

Intellectual property is the foundation of the software industry. The term refers to a range of intangible rights of ownership in an asset such as a software program. Each intellectual property "right" is itself an asset. The law provides different methods for protecting these rights of ownership based on their type. There are essentially four types of intellectual property rights relevant to software: patents, copyrights, trade secrets and trademarks. Each affords a different type of legal protection.

***Topics:***

[Core-Tier1]

- Philosophical foundations of intellectual property
- Intellectual property rights (cross-reference IM/Information Storage and Retrieval/intellectual property and protection)
- Intangible digital intellectual property (IDIP)
- Legal foundations for intellectual property protection
- Digital rights management
- Copyrights, patents, trade secrets, trademarks
- Plagiarism

[Elective]

- Foundations of the open source movement
- Software piracy

***Learning Outcomes:***

[Core-Tier1]

1. Discuss the philosophical bases of intellectual property. [Familiarity]
2. Discuss the rationale for the legal protection of intellectual property. [Familiarity]
3. Describe legislation aimed at digital copyright infringements. [Familiarity]
4. Critique legislation aimed at digital copyright infringements [Assessment]
5. Identify contemporary examples of intangible digital intellectual property [Familiarity]
6. Justify uses of copyrighted materials. [Assessment]
7. Evaluate the ethical issues inherent in various plagiarism detection mechanisms. [Assessment]
8. Interpret the intent and implementation of software licensing. [Familiarity]
9. Discuss the issues involved in securing software patents. [Familiarity]
10. Characterize and contrast the concepts of copyright, patenting and trademarks. [Assessment]

[Elective]

11. Identify the goals of the open source movement. [Familiarity]
12. Identify the global nature of software piracy. [Familiarity]

## SP/ Privacy and Civil Liberties

### *[2 Core-Tier1 hours]*

Electronic information sharing highlights the need to balance privacy protections with information access. The ease of digital access to many types of data makes privacy rights and civil liberties more complex, differing among the variety of cultures worldwide.

***Topics:***

[Core-Tier1]

- Philosophical foundations of privacy rights (cross-reference IS/Fundamental Issues/philosophical issues)
- Legal foundations of privacy protection
- Privacy implications of widespread data collection for transactional databases, data warehouses, surveillance systems, and cloud computing (cross reference IM/Database Systems/data independence; IM/Data Mining/data cleaning)
- Ramifications of differential privacy
- Technology-based solutions for privacy protection (cross-reference IAS/Fundamental Concepts/data protection laws)

[Elective]

- Privacy legislation in areas of practice
- Civil liberties and cultural differences
- Freedom of expression and its limitations

***Learning Outcomes:***

[Core-Tier1]

1. Discuss the philosophical basis for the legal protection of personal privacy. [Familiarity]
2. Evaluate solutions to privacy threats in transactional databases and data warehouses. [Assessment]
3. Recognize the fundamental role of data collection in the implementation of pervasive surveillance systems (e.g., RFID, face recognition, toll collection, mobile computing). [Familiarity]

4. Recognize the ramifications of differential privacy. [Familiarity]
5. Investigate the impact of technological solutions to privacy problems. [Usage]

[Elective]

6. Critique the intent, potential value and implementation of various forms of privacy legislation. [Assessment]
7. Identify strategies to enable appropriate freedom of expression. [Familiarity]

## SP/ Professional Communication

### *[1 Core-Tier1 hour]*

Professional communication conveys technical information to various audiences who may have very different goals and needs for that information. Effective professional communication of technical information is rarely an inherited gift, but rather needs to be taught in context throughout the undergraduate curriculum.

***Topics:***

[Core-Tier1]

- Reading, understanding and summarizing technical material, including source code and documentation
- Writing effective technical documentation and materials
- Dynamics of oral, written, and electronic team and group communication (cross-reference HCI/Collaboration and Communication/group communication; SE/Project Management/team participation)
- Communicating professionally with stakeholders
- Utilizing collaboration tools (cross-reference HCI/ Collaboration and Communication/online communities; IS/Agents/collaborative agents)

[Elective]

- Dealing with cross-cultural environments (cross-reference HCI/User-Centered Design and Testing/cross-cultural evaluation)
- Tradeoffs of competing risks in software projects, such as technology, structure/process, quality, people, market and financial

***Learning Outcomes:***

[Core-Tier1]

1. Write clear, concise, and accurate technical documents following well-defined standards for format and for including appropriate tables, figures, and references. [Usage]
2. Evaluate written technical documentation to detect problems of various kinds. [Assessment]
3. Develop and deliver a good quality formal presentation. [Assessment]
4. Plan interactions (e.g. virtual, face-to-face, shared documents) with others in which they are able to get their point across, and are also able to listen carefully and appreciate the points of others, even when they disagree, and are able to convey to others that they have heard. [Usage]
5. Describe the strengths and weaknesses of various forms of communication (e.g. virtual, face-to-face, shared documents) [Familiarity]
6. Examine appropriate measures used to communicate with stakeholders involved in a project. [Usage]
7. Compare and contrast various collaboration tools. [Assessment]

[Elective]

8. Discuss ways to influence performance and results in cross-cultural teams. [Familiarity]
9. Examine the tradeoffs and common sources of risk in software projects regarding technology, structure/process, quality, people, market and financial. [Usage]
10. Evaluate personal strengths and weaknesses to work remotely as part of a multinational team. [Assessment]

## SP/ Sustainability

### *[1 Core-Tier1 hour, 1 Core-Tier2 hour]*

Sustainability is characterized by the United Nations as "development that meets the needs of the present without compromising the ability of future generations to meet their own needs." Sustainability was first introduced in the CS2008 curricular guidelines. Topics in this emerging area can be naturally integrated into other familiarity areas and units, such as human-computer interaction and software evolution.

***Topics:***

[Core-Tier1]

- Being a sustainable practitioner by taking into consideration cultural and environmental impacts of implementation decisions (e.g. organizational policies, economic viability, and resource consumption).
- Explore global social and environmental impacts of computer use and disposal (e-waste)

[Core-Tier2]

- Environmental impacts of design choices in specific areas such as algorithms, operating systems, networks, databases, programming languages, or human-computer interaction (cross-reference SE/Software Evaluation/software evolution)

[Elective]

- Guidelines for sustainable design standards
- Systemic effects of complex computer-mediated phenomena (e.g. telecommuting or web shopping)
- Pervasive computing. Information processing that has been integrated into everyday objects and activities, such as smart energy systems, social networking and feedback systems to promote sustainable behavior, transportation, environmental monitoring, citizen science and activism.
- Conduct research on applications of computing to environmental issues, such as energy, pollution, resource usage, recycling and reuse, food management, farming and others.
- How the sustainability of software systems are interdependent with social systems, including the knowledge and skills of its users, organizational processes and policies, and its societal context (e.g. market forces, government policies).

***Learning Outcomes:***

[Core-Tier1]

1. Identify ways to be a sustainable practitioner [Familiarity]
2. Illustrate global social and environmental impacts of computer use and disposal (e-waste) [Usage]

[Core-Tier2]

3. Describe the environmental impacts of design choices within the field of computing that relate to algorithm design, operating system design, networking design, database design, etc. [Familiarity]
4. Investigate the social and environmental impacts of new system designs through projects. [Usage]

[Elective]

5. Identify guidelines for sustainable IT design or deployment [Familiarity]
6. List the sustainable effects of telecommuting or web shopping [Familiarity]
7. Investigate pervasive computing in areas such as smart energy systems, social networking, transportation, agriculture, supply-chain systems, environmental monitoring and citizen activism. [Usage]
8. Develop applications of computing and assess through research areas pertaining to environmental issues (e.g. energy, pollution, resource usage, recycling and reuse, food management, farming) [Assessment]

## SP/ History

### *[Elective]*

This history of computing is taught to provide a sense of how the rapid change in computing impacts society on a global scale. It is often taught in context with foundational concepts, such as system fundamentals and software developmental fundamentals.

***Topics:***

- Prehistory—the world before 1946
- History of computer hardware, software, networking (cross-reference AR/Digital logic and digital systems/ history of computer architecture)
- Pioneers of computing
- History of Internet

***Learning Outcomes:***

1. Identify significant continuing trends in the history of the computing field. [Familiarity]
2. Identify the contributions of several pioneers in the computing field. [Familiarity]
3. Discuss the historical context for several programming language paradigms. [Familiarity]
4. Compare daily life before and after the advent of personal computers and the Internet. [Assessment]

## SP/ Economies of Computing

### *[Elective]*

Economics of computing encompasses the metrics and best practices for personnel and financial management surrounding computer information systems. Cost benefit analysis is covered in the Information Assurance and Security Knowledge Area under Risk Management.

***Topics:***

- Monopolies and their economic implications
- Effect of skilled labor supply and demand on the quality of computing products
- Pricing strategies in the computing domain
- The phenomenon of outsourcing and off-shoring software development; impacts on employment and on economics

- Consequences of globalization for the computer science profession
- Differences in access to computing resources and the possible effects thereof
- Costing out jobs with considerations on manufacturing, hardware, software, and engineering implications
- Cost estimates versus actual costs in relation to total costs
- Entrepreneurship: prospects and pitfalls
- Use of engineering economics in dealing with finances

***Learning Outcomes:***

1. Summarize the rationale for antimonopoly efforts. [Familiarity]
2. Identify several ways in which the information technology industry is affected by shortages in the labor supply. [Familiarity]
3. Identify the evolution of pricing strategies for computing goods and services. [Familiarity]
4. Discuss the benefits, the drawbacks and the implications of off-shoring and outsourcing. [Familiarity]
5. Investigate and defend ways to address limitations on access to computing. [Usage]

## SP/ Security Policies, Laws and Computer Crimes

## *[Elective]*

While security policies, laws and computer crimes are important, it is essential they are viewed with the foundation of other Social and Professional knowledge units, such as Intellectual Property, Privacy and Civil Liberties, Social Context, and Professional Ethics. Computers and the Internet, perhaps more than any other technology, have transformed society over the past 50 years. At the same time, they have contributed to unprecedented threats to privacy; whole new categories of crime and anti-social behavior; major disruptions to organizations; and the large-scale concentration of risk into information systems.

***Topics:***

- Examples of computer crimes and legal redress for computer criminals (cross-reference IAS/Digital Forensics/rules of evidence)
- Social engineering, identity theft and recovery (cross-reference HCI/Human Factors and Security/trust, privacy and deception)
- Issues surrounding the misuse of access and breaches in security
- Motivations and ramifications of cyber terrorism and criminal hacking, “cracking”
- Effects of malware, such as viruses, worms and Trojan horses
- Crime prevention strategies
- Security policies (cross-reference IAS/Security Policy and Governance/security policies)

***Learning Outcomes:***

1. List classic examples of computer crimes and social engineering incidents with societal impact. [Familiarity]
2. Identify laws that apply to computer crimes [Familiarity]
3. Describe the motivation and ramifications of cyber terrorism and criminal hacking [Familiarity]
4. Examine the ethical and legal issues surrounding the misuse of access and various breaches in security [Usage]
5. Discuss the professional's role in security and the trade-offs involved. [Familiarity]
6. Investigate measures that can be taken by both individuals and organizations including governments to prevent or mitigate the undesirable effects of computer crimes and identity theft [Usage]

7. Write a company-wide security policy, which includes procedures for managing passwords and employee monitoring. [Usage]

# Appendix B: Migrating to CS2013

A goal of CS2013 is to create guidelines that are realistic and implementable. One question that often arises is, “How are these guidelines different from what we already do?” While it is not possible in this document to answer that question for each institution, it is possible to describe in general terms how these guidelines differ from previous versions in order to provides assistance to curriculum developers in moving a CC2001-conformant curriculum to a CS2013-conformant one.

Toward that end, we first compare the CC2001 Core with the CS2013 Core (Tier-1 and Tier-2 combined). We include brief descriptions of how the content in the KAs changed, what outcomes were removed, and what outcomes emerged.

We also present a worked example of how one of the “model” curricula presented in CC2001 would change if it migrated from the CC2001 guidelines to the CS2013 guidelines. Our purpose is explicitly not to endorse that particular curriculum model (or any particular curriculum model) nor to suggest that it fits all institutions’ needs and constraints, but rather to show how such a curriculum could change. An institution may be familiar with the model updated here even if its own curriculum departs from it. The changes mapped here could find analogous expression when applied to another curriculum. We focus on covering the core effectively, allowing in-depth coverage and specialization in elective courses in the upper division.

## Core Comparison

There is significant overlap between the CC2001 Core and the CS2013 Core, particularly in the more theoretical and fundamental content. This is an indication of growing maturity in the fundamental basis of the field. Knowledge Areas such as Discrete Structures, Algorithms and Complexity, and Programming Languages are updated in CS2013, but the central material is largely unchanged. Two new KAs, Systems Fundamentals and Software Development Fundamentals, are constructed from cross-cutting, foundational material from existing KAs. There are significant differences in the applied and rapidly-changing areas such as information assurance and security, intelligent systems, parallel and distributed computing, and topics related to professionalism and society. This, too, is to be expected of a field that is vibrant and expanding. The comparison is complicated by the fact that in CS2013, we have refined the topic list with levels of understanding and explicit student outcomes for a topic. To compare CC2001 directly, we had to make some reasonable assumptions about what CC2001 intended. The changes are summarized below by knowledge area in CS2013.

| KA | Changes in CS2013 |
|---|---|
| **AL** | This KA now includes a basic understanding of the classes P and NP, the P vs NP problem, and examples of NP-complete problems. It also includes empirical studies for the purposes of comparing algorithm performance. Note that Distributed Algorithms have been moved to PD. |
| **AR** | In this KA, multi-core parallelism, virtual machine support, and power as a constraint are more significant considerations now than a decade ago. The use of CAD tools is prescribed rather than suggested. |
| **CN** | The topics in the core of this are central to "computational thinking", and are at the heart of using computational power to solve problems in domains both inside and outside of traditional CS boundaries. The elective material covers topics that prepare students to contribute to efforts such as computational biology, bioinformatics, eco-informatics, computational finance, and computational chemistry. |
| **DS** | The concepts covered in the core are not new, but some coverage time has shifted from logic to discrete probability, reflecting the growing use of probability as a mathematical tool in computing. Several learning outcomes are also more explicit in CS2013. |
| **GV** | The storage of analog signals in digital form is a general computing idea, as is storing information vs. re-computing. (This outcome appears in SF, also.) Data compression also appears elsewhere.<br>Animation concepts are not in the CC2001 core. |
| **HCI** | Although the core hours have not increased, there is a change in emphasis within the HC KA to recognize the increased importance of design methods and interdisciplinary |

| | |
|---|---|
| | approaches within the specialty. |
| **IAS** | This is a new KA. All of these outcomes reflect the growing emphasis in the profession on security. The IAS KA contains specific security and assurance KU's, however is also heavily integrated with many other KA's. For example defensive programming is addressed in core tier 1 and tier 2 hours within the Programming Languages (PL), System Fundamentals (SF), Systems Engineering (SE), and Operating Systems (OS) KA's. |
| **IM** | The outcomes in this KA reflect topics that are broader than a typical database course. They can easily be covered in a traditional database course, but they must be explicitly addressed. |
| **IS** | Greater emphasis has been placed on machine learning than in the past. Additional guidance has been provided on what is expected of students with respect to understanding the challenges of implementing and using intelligent systems. |
| **NC** | There is greater focus on the comparison of IP and Ethernet networks, and increased attention to wireless networking. A related topic is reliable delivery. Here there is also added emphasis on implementation of protocols and applications. |
| **OS** | This knowledge area is structured to be complementary to Systems Fundamentals, Networks, Information Assurance, and the Parallel and Distributed Computing knowledge areas. While some argue that system administration is the realm of IT and not CS, the working group believes that every student should have the capability to carry out basic administrative activities, especially those impact access control. Security and protection was elective in CC2001 while it has been included in the core in CS2008, we kept it in the core as well. Realization of virtual memory using hardware and software has been moved to be an elective learning outcome (OS/Virtual Machines). Details of deadlocks and their prevention, including detailed concurrency is left to the Parallel and Distributed Systems KA. |
| **PD** | This is a new KA, which demonstrates the need for students to be able to work in parallel and distributed environments. This trend was initially identified, but not included, in |

| | |
|---|---|
| | CS2008. It is made explicit here to reflect that some familiarity with this topic has become essential for all undergraduates in CS. |
| **PL** | For the core material, the outcomes were made more uniform and general by refactoring material on object-oriented programming, functional programming, and event-oriented programming that was in multiple KAs in CC2001. Programming with less mutable state and with more use of higher-order functions (like map and reduce) have greater emphasis. For the elective material, there is greater depth on advanced language constructs, type systems, static analysis for purposes other than compiler optimization, and run-time systems particularly garbage collection. |
| **SDF** | This new KA pulls together foundational concepts and skills needed for software development. It is derived from the Programming Fundamentals KA in CC2001, but also draws basic analysis material from AL, development process from SE, fundamental data structures from DS, and programming language concepts from PL. Material specific to particular programming paradigms (e.g. object-oriented, functional) has been moved to PL to allow for a more uniform treatment with complementary material. |
| **SE** | The changes in this KA introduce or require topics such as refactoring, generic types, secure programming, code modeling, code reviews, contracts, software reliability, testing parallel and distributed software, and team and process improvement. These topics reflect the growing awareness of software process in industry, are central to any level of modern software development, and should be used for software development projects throughout the curriculum. |
| **SF** | This is a new KA. Its outcomes reflect the refactoring of the KAs to identify common themes across previously existing systems-related KAs (in particular, operating systems, networks, and computer architecture). The new cross-cutting thematic areas include parallelism, communications, performance, proximity, virtualization/isolation, and reliability. |
| **SP** | These outcomes reflect a shift in the past decade toward understanding intellectual property as related to digital IP and digital rights management, the need for global |

| | |
|---|---|
| | awareness, and a growing concern for privacy in the digital age. They further recognize the enormous impact that computing has had on society at large emphasizing a sustainable future and placing added responsibilities on computing professionals. The SP outcomes also identify the vital needs for professional ethics, professional development, professional communication, and the ability to collaborate in person as well as remotely across time zones. |

Between one and two dozen topics from CC2001 have been removed from the core, either by moving them to elective material or by elimination entirely. These are summarized in Table 1 below.

Of the 520 core outcomes in CS2013, over 100 are substantially new and another approximately 50 have significantly changed expectations for students. ("Significantly changed expectations" means that while the topic is mentioned in CC2001, the student outcomes are different than what is implied by CC2001. This, of course, is subject to some interpretation. Many programs will already be achieving these outcomes.) These new learning outcomes are identified in Table 2, which appears at the end of this appendix.

## General Observations

This comparison of learning outcomes leads to several general observations about the changes in emphasis from CC2001 to CS2013.

- The topics related to systems and networking have been substantially reorganized to recognize the fundamental principles common among operating systems, networking, and distributed systems.
- Similarly, foundational topics related to software development have been reorganized to produce a more coherent grouping and encourage curricular flexibility.
- Digital logic and numerical methods are not emphasized. A fundamental coverage of digital logic can be found in Systems Fundamentals, but more advanced coverage is considered to be the domain of computer engineering and electrical engineering. Numerical methods are elective material in the CN knowledge area and are treated as a topic geared towards a more selected group of students entering into computational sciences.

- There is significant emphasis on Parallel and Distributed Computing.
- There is a significant new emphasis on Security, Privacy, and Reliability.
- There is no distinguished emphasis on building web pages or search engine use. We assume that students entering undergraduate study in this decade are familiar with internet search, email, and social networking, an assumption that was not universally true in 2001.
- The Programming Languages core in CC2001 had a significant emphasis on language translation. The CS2013 core material in PL is more focused on language paradigms and tradeoffs, rather than implementation. The implementation content is elective.
- SP knowledge area has changed to great degree, particularly with respect to modern issues.

## Conclusions

The changes from CC2001 to CS2013 are significant. Approximately one third of the outcomes are new or significantly changed from those implied by CC2001. Many of these changes were suggested in the CS2008 revision, and reflect current practice in CS programs. Programs may be in a position to migrate their curricula incrementally towards the CS2013 guidelines. In other cases it will be preferable for faculty to revisit the structure of their curriculum to address the changing landscape of computing.

## Table 1: Core Learning Outcomes in CC2001 not found in CS2013

| KA | Topic From CC2001 | Comment |
|---|---|---|
| **PF** | Describe how recursion can be implemented using a stack. | Recursion remains a significant topic, much of which is described in PL now. Implementation specifics are elective topics in CS2013 |
| **AR** | Logic gates, flip flops, PLA, minimization, sum-of-product form, fan-out | Removed (and incorporated under computational paradigms in SF) |
| **AR** | VLIW, EPIC, Systolic architecture;<br>Hypercube, shuffle-exchange, mesh, crossbar as examples of interconnection networks | These topics are elective in CS2013. |
| **NC** | Evolution of early networks; use of common networked applications (e-mail, telnet, FTP, newsgroups, and web browsers, online web courses, and instant messaging);<br>Streams and datagrams;<br>CGI, applets, web servers | This is elective material in C2013. |
| **PL** | Activation records, type parameters, internal representations of objects and methods | Most implementation specifics are elective topics in CS2013, with a basic familiarity with the implementation of key language constructs appearing in Core-Tier-2. |
| **GV** | Affine transformations, homogeneous coordinates, clipping;<br>Raster and vector graphics, physical and logical input devices | This is elective material in C2013. |
| **IM** | Information storage and retrieval | This is elective material in C2013. |
| **SP** | History | This is elective material in C2013. |
| **SP** | Gender-related issues | This material has been expanded to include all under-represented populations. |
| **SP** | Growth of the internet | This material has been subsumed by topics in Social Context with an understanding that students entering undergraduate study no longer consider the Internet to be a novel concept. |

| | | |
|---|---|---|
| **SP** | Freedom of expression | This is elective material in C2013. |
| **SE** | Class browsers, programming by example, API debugging;<br>Tools. | Covered without listing a necessary and sufficient list of tools. |

**Table 2: Core Learning Outcomes in CS2013 not found in CC2001**

| KA | Core Learning Outcome as described in CS2013 |
|---|---|
| **AL** | Tier 1:<br>• Perform empirical studies to validate hypotheses about runtime stemming from mathematical analysis.  Run algorithms on input of various sizes and compare performance<br>• In the context of specific algorithms, identify the characteristics of data and/or other conditions or assumptions that lead to different behaviors<br>Tier 2:<br>• Define the classes P and NP.<br>• Explain the significance of NP-completeness.<br>• Discuss factors other than computational efficiency that influence the choice of algorithms, such as programming time, maintainability, and the use of application-specific patterns in the input data |
| **AR** | Tier 1:<br>• Comprehend the trend of modern computer architectures towards multi-core and that parallelism is inherent in all hardware systems<br>• Explain the implications of the "power wall" in terms of further processor performance improvements and the drive towards harnessing parallelism<br>• Articulate that there are many equivalent representations of computer functionality, including logical expressions and gates, and be able to use mathematical expressions to describe the functions of simple combinational and sequential circuits<br>• Use CAD tools for capture, synthesis, and simulation to evaluate simple building blocks (e.g., arithmetic-logic unit, registers, movement between registers) of a simple computer design |
| **CN** | Tier 1:<br>• Describe the relationship between modeling and simulation, i.e., thinking of simulation as dynamic modeling.<br>• Create a simple, formal mathematical model of a real-world situation and use that model in a simulation.<br>• Differentiate among the different types of simulations, including physical simulations, human-guided simulations, and virtual reality.<br>• Describe several approaches to validating models. |

| | |
|---|---|
| | Tier 2:<br>• Explain the concept of modeling and the use of abstraction that allows the use of a machine to solve a problem. |
| **DS** | Tier 1:<br>• Perform computations involving modular arithmetic.<br>Tier 2:<br>• Compute the variance for a given probability distribution.<br>• Explain how events that are independent can be conditionally dependent (and vice-versa). Identify real-world examples of such cases.<br>• Determine if two graphs are isomorphic. |
| **GV** | Tier 1:<br>• Explain in general terms how analog signals can be reasonably represented by discrete samples, for example, how images can be represented by pixels.<br>• Describe the tradeoffs between storing information vs. storing enough information to reproduce the information, as in the difference between vector and raster rendering.<br>Tier 2:<br>• Describe the differences between lossy and lossless image compression techniques, for example as reflected in common graphics image file formats such as JPG, PNG, and GIF.<br>• Describe the basic process of producing continuous motion from a sequence of discrete frames (sometimes called "flicker fusion").<br>• Describe how double-buffering can remove flicker from animation. |
| **HCI** | Tier 1:<br>• Define a user-centered design process that explicitly recognizes that the user is not like the developer or her acquaintances<br>Tier2:<br>• Create a simple application, together with help and documentation, that supports a graphical user interface<br>• Conduct a quantitative evaluation and discuss/report the results<br>• Discuss at least one national or international user interface design standard |

| | |
|---|---|
| **IAS** | Tier 1:<br>• Understand the tradeoffs and balancing of key security properties (Confidentiality, Integrity, Availability)<br>• Understand the concepts of risk, threats, vulnerabilities and attack vectors (including the fact that there is no such thing as perfect security)<br>• Understand the concept of authentication, authorization, access control<br>• Understand the concept of trust and trustworthiness<br>• Be able to recognize that there are important ethical issues to consider in computer security, including ethical issues associated with fixing or not fixing vulnerabilities and disclosing or not disclosing vulnerabilities<br>• Describe the principle of least privilege and isolation and apply to system design<br>• Understand the principle of fail-safe and deny-by-default<br>• Understand not to rely on the secrecy of design for security (but also that open design alone does not imply security)<br>• Understand the goals of end-to-end data security<br>• Understand the benefits of having multiple layers of defenses<br>• Understand that security has to be a consideration from the point of initial design and throughout the lifecycle of a product<br>• Understanding that security imposes costs and tradeoffs<br>• Understand that an adversary controls the input channel and understand the importance of input validation and data sanitization<br>• Explain why you might choose to develop a program in a type-safe language like Java, in contrast to an unsafe programming language like C/C++<br>• Understand common classes of input validation errors, and be able to write correct input validation code<br>• Demonstrate using a high-level programming language how to prevent a race condition from occurring and how to handle an exception<br>• Demonstrate the identification and graceful handling of error conditions.<br>Tier 2:<br>• Describe the concept of mediation and the principle of complete mediation<br>• Know to use standard components for security operations, instead of re-inventing fundamentals operations<br>• Understand the concept of trusted computing including trusted computing base and attack surface and the principle of minimizing trusted computing base<br>• Understand the importance of usability in security mechanism design<br>• Understand that security does not compose by default; security issues can arise at |

boundaries between multiple components

- Understand the different roles of prevention mechanisms and detection/deterrence mechanisms
- Understand the role of random numbers in security, beyond just cryptography (e.g., password generation, randomized algorithms to avoid algorithmic denial of service attacks)
- Understand the risks with misusing interfaces with third-party code and how to correctly use third-party code
- Understand the need for the ability to update software to fix security vulnerabilities
- Describe likely attacker types against a particular system
- Understand malware species and the virus and limitations of malware countermeasures (e.g., signature-based detection, behavioral detection)
- Identify instances of social engineering attacks and Denial of Service attacks
- Understand the concepts of side channels and covert channels and their differences
- Discuss the manner in which Denial of Service attacks can be identified and mitigated.
- Describe risks to privacy and anonymity in commonly used applications
- Describe the different categories of network threats and attacks
- Describe the architecture for public and private key cryptography and how PKI supports network security.
- Describe virtues and limitations of security technologies at each layer of the network stack
- Identify the appropriate defense mechanism(s) and its limitations given a network threat
- Understand security properties and limitations of other non-wired networks
- Describe the purpose of Cryptography and list ways it is used in data communications.
- Define the following terms: Cipher, Cryptanalysis, Cryptographic Algorithm, and Cryptology and describe the two basic methods (ciphers) for transforming plain text in cipher text.
- Discuss the importance of prime numbers in cryptography and explain their use in cryptographic algorithms.
- Understand how to measure entropy and how to generate cryptographic randomness.
- Demonstrate how Public Key Infrastructure supports digital signing and encryption and discuss the limitations/vulnerabilities.

| | |
|---|---|
| **IM** | Tier 1:<br>• Demonstrate uses of explicitly stored metadata/schema associated with data<br>• Critique/defend a small- to medium-size information application with regard to its satisfying real user information needs<br>Tier 2:<br>• Identify the careers/roles associated with information management (e.g., database administrator, data modeler, application developer, end-user).<br>• Identify issues of data persistence for an organization<br>• Describe several technical solutions to the problems related to information privacy, integrity, security, and preservation<br>• approaches that scale up to globally networked systems<br>• Identify vulnerabilities and failure scenarios in common forms of information systems |
| **IS** | Tier 1:<br>• List the differences among the three main styles of learning: supervised, reinforcement, and unsupervised.<br>• Identify examples of classification tasks, including the available input features and output to be predicted.<br>• Explain the difference between inductive and deductive learning.<br>• Apply the simple statistical learning algorithm such as Naive Bayesian Classifier to a classification task and measure the classifier's accuracy.<br>• Select and implement an appropriate informed search algorithm for a problem by designing the necessary heuristic evaluation function.<br>Tier 2:<br>• Formulate an efficient problem space for a problem expressed in natural language (e.g., English) in terms of initial and goal states, and operators. [Application]<br>• Describe the problem of combinatorial explosion of search space and its consequences.<br>• Describe a given problem domain using the characteristics of the environments in which intelligent systems must function. |

512

513

| | |
|---|---|
| **NC** | Tier 1:<br>• Describe the organization of a wireless network<br>• Describe how wireless networks support mobile users<br>• Describe the operation of reliable delivery protocols<br>• List the factors that affect the performance of reliable delivery protocols<br>• Design and implement a simple reliable protocol<br>Tier 2:<br>• Describe how frames are forwarded in an Ethernet network<br>• Identify the differences between IP and Ethernet<br>• List the differences and the relations between names and addresses in a network<br>• Define the principles behind naming schemes and resource location<br>• Implement a simple client-server socket-based application |
| **OS** | Tier 1:<br>• Carry out simple system administration tasks according to a security policy, for example creating accounts, setting permissions, applying patches, and arranging for regular backups |
| **PD** | Tier 1:<br>• Explain when and why multicast or event-based messaging can be preferable to alternatives<br>• Write a program that correctly terminates when all of a set of concurrent tasks have completed<br>• Use a properly synchronized queue to buffer data passed among activities<br>• Explain why checks for preconditions, and actions based on these checks, must share the same unit of atomicity to be effective<br>• Write a test program that can reveal a concurrent programming error; for example, missing an update when two activities both try to increment a variable<br>• Describe the relative merits of optimistic versus conservative concurrency control under different rates of contention among updates<br>• Compute the work and span, and determine the critical path with respect to a parallel execution diagram<br>• Decompose a problem (e.g., counting the number of occurrences of some word in a document) via map and reduce operations<br>• Implement a parallel divide-and-conquer and/or graph algorithm and empirically measure its performance relative to its sequential analog |

<table>
<tr><td></td><td>
<ul>
<li>Parallelize an algorithm by applying task-based decomposition</li>
<li>Parallelize an algorithm by applying data-parallel decomposition</li>
<li>Distinguish using computational resources for a faster answer from managing efficient access to a shared resource</li>
</ul>
Tier 2:
<ul>
<li>Define “critical path”, “work”, and “span”</li>
<li>Define “speed-up” and explain the notion of an algorithm’s scalability in this regard</li>
<li>Identify independent tasks in a program that may be parallelized</li>
</ul>
</td></tr>
<tr><td>PL</td><td>
Tier 1:
<ul>
<li>Define and use program pieces (such as functions, classes, methods) that use generic types.</li>
</ul>
Tier 2:
<ul>
<li>Write useful functions that take and return other functions.</li>
<li>Reason about memory leaks, dangling-pointer dereferences, and the benefits and limitations of garbage collection.</li>
<li>Process some representation of code for some purpose, such as an interpreter, an expression optimizer, a documentation generator, etc.</li>
</ul>
</td></tr>
<tr><td>SDF</td><td>
Tier 1:
<ul>
<li>Describe how a contract can be used to specify the behavior of a program component.</li>
<li>Refactor a program by identifying opportunities to apply procedural abstraction.</li>
<li>Apply consistent documentation and program style standards that contribute to the readability and maintainability of software.</li>
</ul>
Tier 2:
<ul>
<li>Implement a coherent abstract data type, with loose coupling between components and behaviors.</li>
<li>Identify the relative strengths and weaknesses among multiple designs or implementations for a problem.</li>
<li>Identify common coding errors that lead to insecure programs (e.g., buffer overflows, memory leaks, malicious code) and apply strategies for avoiding such errors.</li>
</ul>
</td></tr>
</table>

514

| | |
|---|---|
| **SE** | Tier 1:<br>• Conduct a review of a set of software requirements to determine the quality of the requirements with respect to the characteristics of good requirements.<br>• List the key components of a class diagram or similar description of the data that a system is required to handle.<br>• Select and use a defined coding standard in a small software project.<br>• Compare and contrast integration strategies including top-down, bottom-up, and sandwich integration.<br>• Describe the process of analyzing and implementing changes to a large existing code base.<br>• For the design of a simple software system within the context of a single design paradigm, describe the software architecture of that system.<br>• Given a high-level design, identify the software architecture by differentiating among common software architectures such as 3-tier, pipe-and-filter, and client-server.<br>• Investigate the impact of software architectures selection on the design of a simple system.<br>• Design a contract for a typical small software component for use in a given system.<br>• Identify weaknesses in a given simple design, and removed them through refactoring.<br>• Discuss the challenges of evolving systems in a changing environment.<br>• Describe the intent and fundamental similarities among process improvement approaches.<br>• Compare several process improvement models such as CMM, CMMI, CQI, Plan-Do-Check-Act, or ISO9000.<br>• Use a process improvement model such as PSP to assess a development effort and recommend approaches to improvement.<br>• Identify behaviors that contribute to the effective functioning of a team.<br>• Create and follow an agenda for a team meeting.<br>• Explain the problems that exist in achieving very high levels of reliability.<br>• Describe how software reliability contributes to system reliability<br>• Describe the issues and approaches to testing distributed and parallel systems.<br>Tier 2:<br>• List the key components of a use case or similar description of some behavior that is required for a system and discuss their role in the requirements |

| | |
|---|---|
| | engineering process.<br>• Describe how software can interact with and participate in various systems including information management, embedded, process control, and communications systems.<br>• Describe how programming in the large differs from individual efforts with respect to understanding a large code base, code reading, understanding builds, and understanding context of changes.<br>• Compare several common process models with respect to their value for development of particular classes of software systems taking into account issues such as requirement stability, size, and non-functional characteristics.<br>• Define software quality and describe the role of quality assurance activities in the software process.<br>• List approaches to minimizing faults that can be applied at each stage of the software lifecycle.<br>• Identify configuration items and use a source code control tool in a small team-based project. |
| **SF** | Tier 1:<br>• List commonly encountered patterns of how computations are organized.<br>• Articulate the concept of strong vs. weak scaling, i.e., how performance is affected by scale of problem vs. scale of resources to solve the problem. This can be motivated by the simple, real-world examples.<br>• Use tools for capture, synthesis, and simulation to evaluate a logic design.<br>• Describe the mechanisms of how errors are detected, signaled back, and handled through the layers.<br>• Construct a simple program using methods of layering, error detection and recovery, and reflection of error status across layers.<br>• Find bugs in a layered program by using tools for program tracing, single stepping, and debugging.<br>• Design and conduct a performance-oriented experiment, e.g., benchmark a parallel program with different data sets in order to iteratively improve its performance.<br>• Use software tools to profile and measure program performance.<br>• Define the differences between the concepts of Instruction Parallelism, Data Parallelism, Thread Parallelism/Multitasking, Task/Request Parallelism.<br>• Write more than one parallel program (e.g., one simple parallel program in more than one parallel programming paradigm; a simple parallel program that manages shared resources through synchronization primitives; a simple parallel program |

| | |
|---|---|
| | that performs simultaneous operation on partitioned data through task parallel (e.g., parallel search terms; a simple parallel program that performs step-by-step pipeline processing through message passing).<br>• Use performance tools to measure speed-up achieved by parallel programs in terms of both problem size and number of resources.<br>• Describe the role of error correcting codes in providing error checking and correction techniques in memories, storage, and networks.<br>• Apply simple algorithms for exploiting redundant information for the purposes of data correction.<br>• Compare different error detection and correction methods for their data overhead, implementation complexity, and relative execution time for encoding, detecting, and correcting errors.<br>Tier 2:<br>• Evaluate performance of simple sequential and parallel versions of a program with different problem sizes, and be able to describe the speed-ups achieved.<br>• Explain the distinction between program errors, system errors, and hardware faults (e.g., bad memory) and exceptions (e.g., attempt to divide by zero).<br>• Articulate the distinction between detecting, handling, and recovering from faults, and the methods for their implementation.<br>• Measure the performance of two application instances running on separate virtual machines, and determine the effect of performance isolation. |
| **SP** | Tier 1:<br>• Evaluate the ethical issues inherent in various plagiarism detection mechanisms.<br>• Identify the goals of the open source movement.<br>• Evaluate solutions to privacy threats in transactional databases and data warehouses.<br>• Recognize the fundamental role of data collection in the implementation of pervasive surveillance systems (e.g., RFID, face recognition, toll collection, mobile computing).<br>• Recognize the ramifications of differential privacy.<br>• Investigate the impact of technological solutions to privacy problems.<br>• Write clear, concise, and accurate technical documents following well-defined standards for format and for including appropriate tables, figures, and references.<br>• Evaluate written technical documentation to detect problems of various kinds.<br>• Develop and deliver a good quality formal presentation.<br>• Plan interactions (e.g. virtual, face-to-face, shared documents) with others in which they are able to get their point across, and are also able to listen carefully |

| | |
|---|---|
| | and appreciate the points of others, even when they disagree, and are able to convey to others that they have heard.<br>• Examine appropriate measures used to communicate with stakeholders involved in a project.<br>• Compare and contrast various collaboration tools.<br>• Describe the mechanisms that typically exist for a professional to keep up-to-date.<br>• Investigate forms of harassment and discrimination and avenues of assistance<br>• Examine various forms of professional credentialing<br>• Identify the social implications of ergonomic devices and the workplace environment to people's health.<br>• Identify developers' assumptions and values embedded in hardware and software design, especially as they pertain to usability for diverse populations including under-represented populations and the disabled.<br>• Investigate the implications of social media on individualism versus collectivism and culture.<br>• Discuss how Internet access serves as a liberating force for people living under oppressive forms of government; explain how limits on Internet access are used as tools of political and social repression.<br>• Analyze the pros and cons of reliance on computing in the implementation of democracy (e.g. delivery of social services, electronic voting).<br>• Describe the impact of the under-representation of diverse populations in the computing profession (e.g., industry culture, product diversity).<br>• Investigate the implications of context awareness in ubiquitous computing systems.<br>• Identify ways to be a sustainable practitioner<br>• Illustrate global social and environmental impacts of computer use and disposal (e-waste)<br>• Describe the environmental impacts of design choices within the field of computing that relate to algorithm design, operating system design, networking design, database design, etc.<br>• Investigate the social and environmental impacts of new system designs through projects.<br>Tier 2:<br>• Discuss the philosophical bases of intellectual property.<br>• Justify uses of copyrighted materials.<br>• Interpret the intent and implementation of software licensing.<br>• Discuss the issues involved in securing software patents. |

| | |
|---|---|
| | • Identify the global nature of software piracy.<br>• Discuss the philosophical basis for the legal protection of personal privacy.<br>• Describe the strengths and weaknesses of various forms of communication (e.g. virtual, face-to-face, shared documents)<br>• Describe issues associated with industries' push to focus on time to market versus enforcing quality professional standards |

# Appendix C: Course Exemplars

While the Body of Knowledge lists the topics and learning outcomes that should be included in undergraduate programs in Computer Science, there is a variety of ways in which these topics may be packaged into courses. Presently, we give a list of *course exemplars*, which provide examples of fielded courses from a variety of institutions that cover portions of the CS2013 Body of Knowledge in different ways. These examplars are not meant to be prescriptive with respect to curricular design—they are not meant to define a standard curriculum for all insitutions. Rather these course examplars are provided to give educators guidance on different ways that that portions of the Body of Knowledge may be organized into courses and to spur innovative thinking in future course design. Table A1 below provides a list of the course examplars in the order in which they appear in this Appendix. Table A2 provides a list of the same course examplars, organized by the Knowledge Area from the Body of Knowledge that they most significantly cover. As can be seen from these exemplars, a course often includes material from multiple Knowledge Areas and in some cases multiple courses may be used to cover all the material from one Knowledge Area.

### Table A1: Courses by Course Title

| Title | Institution | Major KAs | Page |
|---|---|---|---|
| 582219 Operating Systems | Univ. of Helsinki | OS | 224 |
| CS188: Artificial Intelligence | UC Berkeley | IS | 226 |
| CIS133J: Java Programming I | Portland Community College | SDF, PL | 228 |
| CMSC 471: Introduction to Artificial Intelligence | U. Maryland Baltimore County | IS | 231 |
| COS126: General Computer Science | Princeton University | SDF, AL, AR | 233 |
| COS 226: Algorithms and Data Structures | Princeton University | AL | 237 |
| COSC/Math 201: Modeling and Simulation for the Sciences | Wofford College | CN | 240 |
| CPSC 3380 Operating Systems | University of Ark. Little Rock | OS, PD | 244 |
| CS 150 Digital Logic Design | UC Berkeley | AR | 246 |
| CS 152 Computer Engineering | UC Berkeley | AR | 248 |
| CS 2200 Computer Systems and Networks | Georgia Tech | SF | 250 |
| CS 420: Operating Systems | Embry Riddle Aeronautical University | OS, PD | 254 |
| CS 522 Introduction to Computer Architecture | U. Wisconsin-Madison | AR | 256 |
| CS 61c: Great Ideas in Computer Architecture | UC Berkeley | SF | 259 |
| CS 662: Artificial Intelligence Programming | U. San Francisco | IS | 261 |
| CS1101: Introduction to Program Design | Worcester Polytechnic Institute | SDF, PL | 263 |
| CS175 Computer Graphics | Harvard | GV, SE | 266 |

## Table A2: Exemplars by Knowledge Area

NOTE: Courses listed below in parentheses have a secondary emphasis in this area.

| KA | Course | | Page |
|---|---|---|---|
| **AL** | Pomona College | CSCI 140: Algorithms | 299 |
| | Princeton University | COS 226: Algorithms and Data Structures | 237 |
| | Williams College | CSCI 256: Algorithm Design and Analysis | 304 |
| | U. Washington | CSE332: Data Abstractions | 319 |
| | Grinnell College | CSC207: Algorithms and Object-Oriented Design | 289 |
| | (Harvey Mudd College | CS5: Intro to Computer Science) | 275 |
| | (Portland Community College | Discrete Structures 2) | 328 |
| | (Utrect | Languages and Compilers) | 351 |
| | (Princeton University | COS126: General Computer Science) | 233 |
| **AR** | U. Wisconsin-Madison: | CS522: Intro to Computer Architecture | 256 |
| | UC Berkeley: | CS150: Digital Logic Design | 246 |
| | UC Berkeley: | CS152: Computer Engineering | 248 |
| | (Harvey Mudd College: | CS5: Intro to Computer Science) | 275 |
| **CN** | Wofford College | COSC/Math 201: Modeling and Simulation | 240 |
| **DS** | Union County College | Discrete Mathematics | 322 |
| | Stanford Univ. | CS103/CS109: Mathematical Foundations of CS and Probability for CS | 374 |
| | Portland Community College | Discrete Structures 1 | 325 |
| | Portland Community College | Discrete Structures 2 | 328 |
| **GV** | Harvard | CS175:Computer Graphics | 266 |
| | Williams College | CS371: Computer Graphics | 269 |
| | (Miami University (Oxford, OH) | CS262:Technology, Ethics, and Global Society ) | 368 |
| **HCI** | University of York, UK | Human Aspects of Computer Science | 337 |
| | (University of Kent, UK | Human Computer Interaction) | 339 |
| | (Miami University (Oxford, OH) | Technology, Ethics, and Global Society (CS 262)) | 368 |
| **IAS** | *forthcoming* | | |
| **IM** | *forthcoming* | | |
| **IS** | U. San Francisco | Artificial Intelligence Programming | 261 |
| | U. Maryland, Baltimore County | Introduction to Artificial Intelligence | 231 |
| | Case Western Reserve Univ. | Artificial Intelligence | 341 |
| | UC Berkeley | CS188: Artificial Intelligence | 226 |
| | Univ. Hartford | Artificial Intelligence | 344 |
| **NC** | *forthcoming* | | |
| **OS** | Williams College | CSCI 432: Operating Systems | 310 |
| | University of Ark. Little Rock | CPSC 3380: Operating Systems | 244 |
| | Embry Riddle Aeronautical Univ. | CS 420: Operating Systems | 254 |
| | Univ. of Helsinki | 582219 Operating Systems | 224 |
| **PBD** | *forthcoming* | | |
| **PD** | Nizhni Novgorod State University | Introduction to Parallel Programming | 347 |
| | (U. Washington | CSE332: Data Abstractions) | 319 |
| | (University of Ark. Little Rock | CPSC 3380: Operating Systems) | 244 |
| | (Embry Riddle Aeronautical Univ. | CS 420: Operating Systems) | 254 |
| | (Williams College | CSCI 334: Principles of Programming Languages) | 307 |
| **PL** | *Compilers* | | |
| | Colorado State University | CS 453: Introduction to Compilers | 272 |
| | Univ. Arizona, Tucson | CSC 453: Translators and Systems Software | 281 |
| | Williams College | CSCI 434T: Compiler Design | 313 |

| | | | |
|---|---|---|---|
| | Utrect | Languages and Compilers | 351 |
| | Rice | Topics in Compiler Construction | 371 |
| | *Programming Languages* | | |
| | Pomona College | CS 131: Principles of Programming Languages | 278 |
| | Brown Univ. | CSCI 1730: Introduction to Programming | 302 |
| | U. Washington | Programming Languages | 356 |
| | Williams College | CSCI 334 Principles of Programming | 307 |
| | Univ. of Penn. | Programming Languages and Techniques I | 359 |
| | (Brown Univ. | CSCI 0190: Accelerated Intro. to Computer Science) | 297 |
| | (Portland Community College | CIS 133J: Java Programming I) | 228 |
| | (Worchester Polytechnic Inst. | CS1101: Introduction to Program Design) | 263 |
| **SDF** | ***Also see Introductory Sequences (at end of table)*** | | 283, 292 |
| | Portland Community College | CIS133J: Java Programming I | 228 |
| | Harvey Mudd College | CS5: Introduction to Computer Science | 275 |
| | Worchester Polytechnic Inst. | CS1101: Introduction to Program Design | 263 |
| | (Univ. of Penn. | Programming Languages and Techniques I) | 359 |
| | (Princeton University | COS126: General Computer Science) | 233 |
| | (Brown Univ. | CSCI 0190: Accelerated Intro. to Computer Science) | 297 |
| **SE** | Embry Riddle Aeronautical Univ. | Software Engineering Practices | 364 |
| | Milwaukee School of Engineering | SE 2890:Software Engineering Practices | 362 |
| | (Colorado State University | CS453: Introduction to Compilers) | 272 |
| | (Harvard | CS175 Computer Graphics) | 266 |
| | (Williams College | CS371: Computer Graphics) | 269 |
| | (Brown Univ. | CSCI 0190: Accelerated Intro. to Computer Science) | 297 |
| **SF** | Georgia Tech | CS 2200: Computer Systems and Networks | 250 |
| | UC Berkeley | CS 61c: Great Ideas in Computer Architecture | 259 |
| | U. Washington | CSE 333: System Programming | 316 |
| **SP** | Univ. of Maryland, Univ. College | Ethics in Technology (IFSM 304) | 334 |
| | Saint Xavier University | Issues in Computing | 349 |
| | Anne Arundel Community College | Ethics & the Information Age (CSI 194) | 331 |
| | Miami University (Oxford, OH) | Technology, Ethics, and Global Society | 368 |
| | Northwest Missouri State Univ. | Professional Development Seminar | 353 |
| **Introductory Sequences** | Creighton University<br>CSC221: Introduction to Programming<br>CSC222: Object-Oriented Programming | | 292 |
| | Grinnell College<br>CSC207: Algorithms and Object-Oriented Design<br>CSC161: Imperative Problem Solving and Data Structures<br>CSC151: Functional problem solving | | 283 |

# 582219 Operating Systems
## University of Helsinki, Department of Computer Science

**Dr. Teemu Kerola**
**teemu.kerola@cs.helsinki.fi**

**https://www.cs.helsinki.fi/en/courses/582219**

**Knowledge Areas that contain topics and learning outcomes covered in the course**

| Knowledge Area | Total Hours of Coverage |
|---|---|
| Operating Systems (OS) | 23 |
| System Fundamentals (SF) | 4 |
| Parallel and Distributed Computing (PD) | 3 |
| Architecture and Organization (AR) | 2 |

### Brief description of the course's format and place in the undergraduate curriculum

Pre-requisites: Computer Organization I (24h). Course targeted to $2^{nd}$ year students. Course consists of 18 lectures (2h) and 9 homework practice sessions (2h).
Followup courses: Distributed Systems, Mobile Middleware, OS lab project (in planning).

### Course description and goals

Understand OS services to applications, concurrency problems and solution methods for them, OS basic structure, principles and methods of OS implementation.

### Course topics

OS history, process, threads, multicore, concurrency problems and their solutions, deadlocks and their prevention, memory management, virtual memory, scheduling, I/O management, disk scheduling, file management, embedded systems, distributed systems.

### Course textbooks, materials, and assignments

Textbook: "Operating Systems – Internals and Design Principles, $7^{th}$ ed." by W. Stallings, Pearson Education Ltd, 2012, ISBN 13: 978-0-273-75150-2
Homework 1: Overview, multicore, cache
Homework 2: Processes, threads
Homework 3: Mutual exclusion, scenarios, semaphores, monitors, producer/consumer
Homework 4: Message-passing, readers/writers, deadlocks
Homework 5: Memory management, virtual memory
Homework 6: Scheduling
Homework 7: I/O management
Homework 8: File management, embedded systems
Homework 9: Distributed systems
Exams: 2 (2.5h each)

**Body of Knowledge coverage**

| KA | Knowledge Unit | Topics Covered | Hours |
|---|---|---|---|
| OS | OS/Overview of Operating Systems | Role, functionality, design issues, history, evolution, SMP considerations | 2 |

| | | | |
|---|---|---|---|
| AR | AR/Memory System Organization | Cache, TLB | 2 |
| SF | SF/Computational Paradigms | Processes, threads, process/kernel states | 1 |
| SF | SF/Cross-Layer Communication | Layers, interfaces, RPC, abstractions | 1 |
| SF | SF/Parallelism | Client/server computing, HW-support for synchronization, multicore architectures | 2 |
| OS | OS/Operating System Principles | Process control, OS structuring methods, interrupts, kernel-mode | 2 |
| OS | OS/Concurrency | Execution scenarios, critical section, spin-locks, synchronization, semaphores, monitors | 4 |
| PD | PD/Communication-Coordination | Message passing, deadlock detection and recovery, deadlock prevention, deadlock avoidance | 2 |
| OS | OS/Scheduling and Dispatch | Scheduling types, process/thread scheduling, multiprocessor scheduling | 4 |
| OS | OS/Memory Management | Memory management, partitioning, paging, segmentation | 2 |
| OS | OS/Security and Protection | -- covered in Introduction to Computer Security -- | |
| OS | OS/Virtual Machines | Hypervisors, virtual machine monitor, virtual machine implementation, virtual memory, virtual file systems | 3 |
| OS | OS/Device Management | Serial and parallel devices, I/O organization, buffering, disk scheduling, RAID, disk cache | 2 |
| OS | OS/File Systems | File organization, file directories, file sharing, disk management, file system implementation, memory mapped files, journaling and log structured systems | 2 |
| OS | OS/Real Time and Embedded Systems | Real time systems, real time OS characteristics, real-time scheduling, embedded systems OS characteristics | 2 |
| PD | PD/Parallel Architectures | Multicore, SMP, shared/distributed memory, clusters | 1 |

**Additional Topics**

None.

**Other comments**

We have special emphasis on concurrency, because more and more applications are executed multithreaded in multicore environments.

# CS188: Artificial Intelligence
# University of California Berkeley

**Dan Klein**
**klein@cs.berkeley.edu**

**http://inst.eecs.berkeley.edu/~cs188/sp12/announcements.html**

**Knowledge Areas that contain topics and learning outcomes covered in the course**

| **Knowledge Area** | **Total Hours of Coverage** |
|---|---|
| *Intelligent Systems (IS)* | *27* |
| *Human Computer Interaction (HC)* | 1 |

## Brief description of the course's format and place in the undergraduate curriculum

The pre-requisites of this course are:

- **CS 61A or 61B:** Prior computer programming experience is expected (see below); most students will have taken both these courses.
- **CS 70 or Math 55:** Facility with basic concepts of propositional logic and probability are expected (see below); CS 70 is the better choice for this course.

This course has substantial elements of both programming and mathematics, because these elements are central to modern AI. Students must be prepared to review basic probability on their own. Students should also be very comfortable programming on the level of CS 61B even though it is not strictly required.

## Course description and goals

This course will introduce the basic ideas and techniques underlying the design of intelligent computer systems. A specific emphasis will be on the statistical and decision-theoretic modeling paradigm By the end of this course, you will have built autonomous agents that efficiently make decisions in fully informed, partially observable and adversarial settings. Your agents will draw inferences in uncertain environments and optimize actions for arbitrary reward structures. Your machine learning algorithms will classify handwritten digits and photographs. The techniques you learn in this course apply to a wide variety of artificial intelligence problems and will serve as the foundation for further study in any application area you choose to pursue.

## Course topics

- Introduction to AI
- Search
- Constraint Satisfaction
- Game Playing
- Markov Decision Processes
- Reinforcement Learning
- Bayes Nets
- Hidden Markov Modeling
- Speech
- Neural Nets
- Robotics
- Computer Vision

**Course textbooks, materials, and assignments**

The textbook is Russell and Norvig, *Artificial Intelligence: A Modern Approach*, Third Edition.
All the projects in this course will be in Python. The projects will be on the following topics:

1. Search
2. Multi-Agent Pacman
3. Reinforcement Learning
4. Bayes Net
5. Classification

**Body of Knowledge coverage**

| KA | Knowledge Unit | Topics Covered | Hours |
|---|---|---|---|
| *IS* | *Fundamental Issues* | *All* | 1 |
| IS | Basic Search Strategies | All | 2.5 |
| IS | Basic Knowledge Representation and Reasoning | Probability, Bayes Theorem | 2.5 |
| IS | Basic Machine Learning | All | 1 |
| IS | Advanced Search | Except Genetic Algorithms | 3 |
| IS | Reasoning Under Uncertainty | | 6 |
| IS | Agents | | 0.5 |
| IS | Natural Language Processing | | 0.5 |
| IS | Advanced Machine Learning | | 4 |
| IS | Robotics | | 1 |
| IS | Perception and Computer Vision | | 0.5 |
| HC | Design for non-mouse interfaces | | 1 |

**Additional topics**

- Neural Networks – 2 hours
- DBNs, Particle Filtering, VPI – 0.5 hours

**Other comments**

None

# CIS 133J: Java Programming I

## Portland Community College, Portland, OR

**Cara Tang**
**cara.tang@pcc.edu**

**http://www.pcc.edu/ccog/default.cfm?fa=ccog&subject=CIS&course=133J**

**Knowledge Areas that contain topics and learning outcomes covered in the course**

| Knowledge Area | Total Hours of Coverage |
|---|---|
| Software Development Fundamentals (SDF) | 26 |
| Programming Languages (PL) | 11 |
| Algorithms and Complexity (AL) | 3 |

### Where does the course fit in your curriculum?

The prerequisite for this course is the course "Software Design", which covers the basics of software design in a language-independent manner.

This course can be taken in the first or second year. A two-course programming sequence is required for the CIS degree, of which 3 are offered in the languages Java, VB.NET, and C++. This course is the first course in the Java sequence. The second follow-on course is required and a third course in the sequence is optional.

About 290 students take the course each year. While most are working towards a CIS Associate's degree, some students take it for transfer credit at a local institution such as OIT (Oregon Tech), and some take it simply to learn the Java language.

### What is covered in the course?

- Object-oriented programming concepts
- Objects, classes
- State, behavior
- Methods, fields, constructors
- Variables, parameters
- Scope, lifetime
- Abstraction, modularization, encapsulation
- Method overloading
- Data types
- Conditional statements, logical expressions
- Loops
- Collection processing
- Using library classes
- UML class diagrams
- Documentation
- Debugging
- Use of an IDE

### What is the format of the course?

The course is offered both face-to-face and online. In the face-to-face version, there are 40 classroom hours and at least 20 optional lab hours. This is a quarter course and typically meets twice a week for two hours over 10 weeks, with 2 optional lab hours each week.

All classrooms are equipped with a computer at each desk and the classroom time consists of both lecture and activities on the computers. The lab time is unstructured; students typically work on assignments and ask questions related to assignments or readings.

The online version of the course has pre-recorded videos guiding the students through the basic concepts and some of the more difficult content. There are also in-house written materials that supplement the textbook. Discussion boards provide a forum for questions and class discussions. Assignments and assessment are the same as in the face-to-face course.

### How are students assessed?

Assessment varies by instructor, but in all cases the majority of the final grade comes from programming projects (e.g., 70%), and a smaller portion (e.g., 30%) from exams.

There are seven programming projects. In six of the projects, students add functionality to existing code, ranging from adding a single one-line method in the first assignment to adding significant functionality to skeleton code in the last assignment. In one project students write all code from scratch.

Students are expected to spend approximately two hours outside of class studying and working on assignments for each one hour they spend in class.

The midterm and final exams consist of multiple choice and true-false questions, possibly with a portion where students write a small program.

### Course textbooks and materials

The textbook is Objects First with BlueJ, and the BlueJ environment is used throughout the course.

### Why do you teach the course this way?

This course was previously taught with a more procedural-oriented approach and using a full-fledged IDE. The switch was made to BlueJ and a true objects-first approach in order to concentrate more on the concepts and less on syntactical and practical details needed to get a program running. In addition, there is an emphasis on good program design.

The background and skill level of students in the course varies greatly, and some find it very challenging while others have no trouble with the course.

**Body of Knowledge coverage**

| KA | Knowledge Unit | Topics Covered | Hours |
|---|---|---|---|
| AL | Fundamental Data Structures and Algorithms | Simple numerical algorithms and sequential search are covered | 3 |
| PL | Object-Oriented Programming | Core-tier1 and core-tier2 topics are covered including classes, objects with state and behavior, encapsulation, visibility, collection classes. The topics relating to inheritance, subtyping, and class hierarchies are not covered in this course but are covered in the next course in the sequence. | 8 |

| | | | |
|---|---|---|---|
| PL | Basic Type Systems | All core-tier1 topics are covered with the exception of discussion of static typing. Of the core-tier2 topics, only generic types are covered, in connection with Java collection classes. | 2 |
| PL | Language Translation and Execution | Interpretation vs. compilation is covered, in connection with the Java language model as contrasted with straight compiled languages such as C++. | 1 |
| SDF | Algorithms and Design | The role of algorithms, problem-solving strategies (excluding recursion), and fundamental design concepts are covered. The concept and properties of algorithms, including comparison, has been introduced in a prerequisite course. | 2 |
| SDF | Fundamental Programming Concepts | All topics except recursion are covered | 10 |
| SDF | Fundamental Data Structures | Arrays, records, strings, lists, references and aliasing are covered | 12 |
| SDF | Development Methods | All topics are covered, but some of the program correctness subtopics only at the familiarity level | 2 |

# CMSC 471, Introduction to Artificial Intelligence

## University of Maryland, Baltimore, County; Baltimore, MD

**Marie desJardins**
**mariedj@cs.umbc.edu**

**http://www.csee.umbc.edu/courses/undergraduate/471/fall11/**

**Knowledge Areas that contain topics and learning outcomes covered in the course**

| Knowledge Area | Total Hours of Coverage |
|---|---|
| Intelligent Systems (IS) | 38 |
| Programming Languages (PL) | 1 |

### Where does the course fit in your curriculum?

Most students take CMSC 471 in their senior year, but some take it as a junior. It is a "core elective" in our curriculum (students have to take two of the 11 core courses, which also include areas such as databases, networking, and graphics). Students in the Game track (5-10% of CS majors) must take CMSC 471 as one of their core electives. The prerequisite course is CMSC 341 (Data Structures, which itself has a Discrete Structures prerequisite). CMSC 471 is offered once a year, capped at 40 students, and is always full with a waiting list.

### What is covered in the course?

Course description: "This course will serve as an introduction to artificial intelligence concepts and techniques. We will use the Lisp programming language as a computational vehicle for exploring the techniques and their application. Specific topics we will cover include the history and philosophy of AI, Lisp and functional programming, the agent paradigm in AI systems, search, game playing, knowledge representation and reasoning, logical reasoning, uncertain reasoning and Bayes nets, planning, and machine learning. If time permits, we may also briefly touch on multi-agent systems, robotics, perception, and/or natural language processing."

### What is the format of the course?

The course is face-to-face, two 75-minute sessions per week (three credit hours). The primary format is lecture but there are many active learning and problem solving activities integrated into the lecture sessions.

### How are students assessed?

There are typically six homework assignments (with a mix of programming and paper-and-pencil exercises), a semester project that can be completed in small groups, and midterm and final exams. Students typically spend anywhere from 5-20 hours per week outside of class completing the required readings and homeworks.

### Course textbooks and materials

The primary textbook is Russell and Norvig's "Artificial Intelligence: A Modern Approach." Students are expected to learn and use the Lisp programming language (CLISP implementation), and Paul Graham's "ANSI Common Lisp" is also assigned.

### Why do you teach the course this way?

My intention is to give students a broad introduction to the foundational principles of artificial intelligence, with enough understanding of algorithms and methods to be able to implement and analyze them. Students who have completed the course should be able to continue into a graduate program of study in AI and be able to successfully apply what they have learned in this class to solve new problems. I also believe that the foundational concepts of search, probabilistic reasoning, logical reasoning, and knowledge representation are extremely useful in other areas even if students don't continue in the field of AI. Using Lisp exposes them to a functional programming language and increases their ability to learn a new language and a different way of thinking. Most students describe the course as one of the most difficult of their undergraduate career, but it also receives very high ratings in terms of interest and quality.

**Body of Knowledge coverage**

| KA | Knowledge Unit | Topics Covered | Hours |
|---|---|---|---|
| IS | Fundamental Issues | Intelligence, agents, environments, philosophical issues | 4 |
| IS | Basic Search Strategies | Problem spaces, uninformed/informed/local search, minimax, constraint satisfaction | 4 |
| IS | Basic Knowledge Representation and Reasoning | Propositional and first-order logic, resolution theorem proving | 5.5 |
| IS | Basic Machine Learning | Learning tasks, inductive learning, naive Bayes, decision trees | 1.5 |
| IS | Advanced Search | A* search, genetic algorithms, alpha-beta pruning, expectiminimax | 6 |
| IS | Advanced Representation and Reasoning | Ontologies, nonmonotonic reasoning, situation calculus, STRIPS and partial-order planning, GraphPlan | 3.5 |
| IS | Reasoning Under Uncertainty | Probability theory, independence, Bayesian networks, exact inference, decision theory | 6 |
| IS | Agents | Game theory, multi-agent systems | 4.5 |
| IS | Advanced Machine Learning | Nearest-neighbor methods, SVMs, K-means clustering, learning Bayes nets, reinforcement learning | 3 |
| PL | Functional Programming | Lisp programming | 1 |

### Additional topics

N/A

### Other comments

Note: Additional electives are offered in Robotics, Machine Learning, Autonomous Agents and Multi-Agent Systems, and Natural Language Processing.

# COS 126: General Computer Science
# Princeton University, Princeton, NJ

**Robert Sedgewick and Kevin Wayne**
rs@cs.princeton.edu **wayne@cs.princeton.edu**

**http://www.cs.princeton.edu/courses/archive/spring12/cos226/info.php**
http://algs4.cs.princeton.edu

**Knowledge Areas that contain topics and learning outcomes covered in the course**

| **Knowledge Area** | **Total Hours of Coverage** |
|---|---|
| SDF/Software Development Fundamentals | 11 |
| AL/Algorithms and Complexity | 11 |
| AR/Architecture and Organization | 5 |
| PL/Programming Languages | 4 |
| CN/Computational Science | 1 |
| SP/Social and Professional Issues | 1 |
| IS/Intelligent Systems | 1 |

### Where does the course fit in your curriculum?

This course is an introduction to computer science, intended for all first-year students. It is intended to be analogous to commonly accepted introductory courses in mathematics, physics, biology, and chemistry. It is not just a "first course for CS majors" but also a introduction to the field that is taken by over 60% of all Princeton students.

### What is covered in the course?

We take an interdisciplinary approach to the traditional CS1 curriculum, where we teach students to program while highlighting the role of computing in other disciplines, then take them through fundamental precepts of the field of computer science. This approach emphasizes for students the essential idea that mathematics, science, engineering, and computing are intertwined in the modern world, while at the same time preparing students to use computers effectively for applications in computer science, physics, biology, chemistry, engineering, and other disciplines. Instructors teaching students who have successfully completed this course can expect that they have the knowledge and experience necessary to enable them to adapt to new computational environments and to effectively exploit computers in diverse applications. At the same time, students who choose to major in computer science get a broad background that prepares them for detailed studies in the field.
Roughly, the first half of the course is about learning to program in a modern programming model, with applications. The second half of the course is a broad introduction to the field of computer science.

- Introduction to programming in Java. Elementary data types, control flow, conditionals and loops, and arrays.
- Input and output.
- Functions and libraries.
- Analysis of algorithms, with an emphasis on using the scientific method to validate hypotheses about algorithm performance.
- Machine organization, instruction set architecture, machine language programming.
- Data types, APIs, encapsulation.

- Linked data structures, resizing arrays, and implementations of container types such as stacks and queues.
- Sorting (mergesort) and searching (binary search trees).
- Programming languages.
- Introduction to theory of computation. Regular expressions and finite automata.
- Universality and computability.
- Intractability.
- Logic design, combinational and sequential circuits.
- Processor and memory design.
- Introduction to artificial intelligence.

**What is the format of the course?**

The material is presented in two one-hour lectures per week, supported by two one-hour sections covered by experienced instructors teaching smaller groups of students. Roughly, one of these hours is devoted to presenting new material that might be covered in a lecture hour; the other is devoted to covering details pertinent to assignments and exams.

**How are students assessed?**

*Programming projects*. The bulk of the assessment is weekly programming assignments, which usually involve solving an interesting application problem that reinforces a concept learned in lecture. Students spend 10-20 hours per week on these assignments and often consult frequently with section instructors for help. Examples include:

- Monte Carlo simulation of random walks.
- Graphical simulation of the N-body problem.
- Design and plot a recursive pattern.
- Implement a dynamic programming solution to the global DNA sequence alignment problem.
- Simulate a linear feedback shift register and use it to encrypt/decrypt an image.
- Simulate plucking a guitar string using the Karplus-Strong algorithm and use it to implement an interactive guitar player.
- Implement two greedy heuristics to solve the traveling salesperson problem.
- Re-affirm the atomic nature of matter by tracking the motion of particles undergoing Brownian motion, fitting this data to Einstein's model, and estimating Avogadro's number.

*In-class programming tests*. At mid-term and at the end of the semester, students have to solve mini-programming assignments (that require 50-100 lines of code to solve) in a supervised environment in 1.5 hours. Practice preparation for these tests is a significant component in the learning experience.

*Hourly exams*. At mid-term and at the end of the semester, students take traditional hourly exams to test their knowledge of the course material.

**Course textbooks and materials.**

The first half of the course is based on the textbook *Introduction to Programming in Java: An Interdisciplinary Approach* by Robert Sedgewick and Kevin Wayne (Addison-Wesley, 2008). The book is supported by a public "booksite" (http://introcs.cs.princeton.edu) that contains a condensed version of the text narrative (for reference while online), Java code for the algorithms and clients in the book and many related algorithms and clients, test data sets, simulations, exercises, and solutions to selected exercises. The booksite also has lecture slides and other teaching material for use by faculty at other universities.

A separate website specific to each offering of the course contains detailed information about schedule, grading policies, and programming assignments.

**Why do you teach the course this way?**

The motivation for this course is the idea that knowledge of computer science is for *everyone*, not just for programmers and computer-science students. Our goal is to demystify computation, empower students to use it effectively and to build awareness of its reach and the depth of its intellectual underpinnings. We teach students to program in a familiar context to prepare them to apply their skills in later courses in whatever their chosen field and to recognize when further education in computer science might be beneficial.

Prospective computer science majors, in particular, can benefit from learning to program in the context of scientific applications. A computer scientist needs the same basic background in the scientific method and the same exposure to the role of computation in science as does a biologist, an engineer, or a physicist. Indeed, our interdisciplinary approach enables us to teach prospective computer science majors and prospective majors in other fields of science and engineering in the same course. We cover the material prescribed by CS1, but our focus on applications brings life to the concepts and motivates students to learn them. Our interdisciplinary approach exposes students to problems in many different disciplines, helping them to more wisely choose a major. Our reach has expanded beyond the sciences and engineering, to the extent that we are one of the most popular science courses taken by students in the humanities and social sciences, as well.

**Body of Knowledge coverage**

| KA | Knowledge Unit | Topics Covered | Hours |
|---|---|---|---|
| SDF | Algorithms and Design | The concept and properties of algorithms. The role of algorithms in the problem-solving process. Problem-solving strategies. Implementation of algorithms. Fundamental design concepts and principles | 2 |
| SDF | Fundamental Programming Concepts | Basic syntax and semantics of a higher-level language. Variables and primitive data types. Expressions and assignments. I/O. Conditional and iterative control structures | 3 |
| SDF | Fundamental Data Structures | Arrays, stacks, queues, priority queues, strings, references, linked structures, resizable arrays. Strategies for choosing the appropriate data structure. | 3 |
| SDF | Development Methods | Program correctness. Modern programming environments. Debugging strategies. Documentation and program style. | 3 |
| PL | Object-Oriented programming | Object-oriented design, encapsulation, iterators. | 2 |
| PL | Basic Type Systems | Primitive types, reference types. Type safety, static typing. Generic types. | 2 |
| AL | Basic Analysis | Asymptotic analysis, empirical measurements. Differences among best, average, and worst case behaviors of an algorithm. Complexity classes, such as constant, logarithmic, linear, quadratic, and exponential. Time and space trade-offs in algorithms. | 2 |
| AL | Algorithmic Strategies | Brute-force, greedy, divide-and-conquer, and recursive algorithms. Dynamic programming, reduction. | 2 |

| | | | |
|---|---|---|---|
| AL | Fundamental Data Structures and Algorithms | Binary search. Insertion sort, mergesort. Binary search trees, hashing. Representations of graphs. Graph search. | 3 |
| AL | Basic Automata, Computability and Complexity | Finite-state machines, regular expressions, P vs. NP, NP-completeness, NP-complete problems | 2 |
| AL | Advanced Automata, Computability and Complexity | Languages, DFAs. Universality. Computability. | 2 |
| AR | Architecture and Organization | Overview and history of computer architecture, Combinational vs. sequential logic | 1 |
| AR | Machine Representation of Data | Bits, bytes, and words, Numeric data representation and number bases, Fixed- and floating-point systems, Signed and twos-complement representations. Representation of non-numeric data. Representation of records and arrays | 1 |
| AR | Assembly level machine organization | Basic organization of the von Neumann machine. Control unit; instruction fetch, decode, and execution. Instruction sets and types. Assembly/machine language programming. Instruction formats. Addressing modes | 2 |
| AR | Functional Organization | Control unit. Implementation of simple datapaths. | 1 |
| CN | Fundamentals | Introduction to modeling and simulation. | 1 |
| IS | Fundamental Issues | Overview of AI problems. Examples of successful recent AI applications | 1 |
| SP | History | History of computer hardware and software. Pioneers of computing. | 1 |

# COS 226: Algorithms and Data Structures
## Princeton University, Princeton, NJ

**Robert Sedgewick and Kevin Wayne**
rs@cs.princeton.edu wayne@cs.princeton.edu

**http://www.cs.princeton.edu/courses/archive/spring12/cos226/info.php**
http://algs4.cs.princeton.edu

**Knowledge Areas that contain topics and learning outcomes covered in the course**

| Knowledge Area | Total Hours of Coverage |
|---|---|
| AL/Algorithms and Complexity | 29 |
| SDF/Software Development Fundamentals | 3 |
| PL/Programming Languages | 1 |

**Where does the course fit in your curriculum?**

This course introduces fundamental algorithms in the context of significant applications in science, engineering, and commerce. It has evolved from a traditional "second course for CS majors" to a course that is taken by over one-third of all Princeton students. The prequisite is a one-semester course in programming, preferably in Java and preferably at the college level. Our students nearly all fulfill the prerequisite with our introductory course. This course is a prerequisite for all later courses in computer science, but is taken by many students in other fields of science and engineering (only about one-quarter of the students in the course are majors).

**What is covered in the course?**

Classical algorithms and data structures, with an emphasis on implementing them in modern programming environments, and using them to solve real-world problems. Particular emphasis is given to algorithms for sorting, searching, string processing, and graph algorithms. Fundamental algorithms in a number of other areas are covered as well, including geometric algorithms and some algorithms from operations research. The course concentrates on developing implementations, understanding their performance characteristics, and estimating their potential effectiveness in applications.

- Analysis of algorithms, with an emphasis on using the scientific method to validate hypotheses about algorithm performance.
- Data types, APIs, encapsulation.
- Linked data structures, resizing arrays, and implementations of container types such as stacks and queues.
- Sorting algorithms, including insertion sort, selection sort, shellsort, mergesort, randomized quicksort, heapsort.
- Priority queue data types and implementations, including binary heaps.
- Symbol table data types and implementations (searching algorithms), including binary search trees, red-black trees, and hash tables.
- Geometric algorithms (searching in point sets and intersection).
- Graph algorithms (breadth-first search, depth-first search, MST, shortest paths, topological sort, strong components, maxflow)
- Tries, string sorting, substring search, regular expression pattern matching.
- Data compression (Huffman, LZW).
- Reductions, combinatorial search, P vs. NP, and NP-completeness.

**What is the format of the course?**

The material is presented in two 1.5 hour lectures per week with weekly quizzes and a programming assignment, supported by a one-hour section where details pertinent to assignments and exams are covered by TAs teaching smaller groups of students. An alternative format is to use online lectures supplemented by two 1.5 hour sections, one devoted to discussion of lecture material, the other devoted to assignments.

**How are students assessed?**

The bulk of the assessment is weekly programming assignments, which usually involve solving an interesting application problem using an efficient algorithm learned in lecture. Students spend 10-20 hours per week on these assignments and often consult frequently with section instructors for help.

- Monte Carlo simulation to address the percolation problem from physical chemistry, based on efficient algorithms for the union-find problem.
- Develop generic data types for deques and randomized queues.
- Find collinear points in a point set, using an efficient generic sorting algorithm implementation.
- Implement A* search to solve a combinatorial problem, based on an efficient priority queue implementation.
- Implement a data type that supports range search and near-neighbor search in point sets, using kD trees.
- Build and search a "WordNet" directed acyclic graph.
- Use maxflow to solve the "baseball elimination" problem.
- Develop an efficient implementation of Burrow-Wheeler data compression.

Exercises for self-assessment are available on the web, are a topic of discussion in sections, and are good preparation for exams. A mid-term exam and a final exam account for a significant portion of the grade.

**Course textbooks and materials**

The course is based on the textbook Algorithms, 4th Edition by Robert Sedgewick and Kevin Wayne (Addison-Wesley Professional, 2011, ISBN 0-321-57351-X). The book is supported by a public "booksite" (http://algs4.cs.princeton.edu), which contains a condensed version of the text narrative (for reference while online) Java code for the algorithms and clients in the book, and many related algorithms and clients, test data sets, simulations, exercises, and solutions to selected exercises. The booksite also has lecture slides and other teaching material for use by faculty at other universities.

A separate website specific to each offering of the course contains detailed information about schedule, grading policies, and programming assignments.

**Why do you teach the course this way?**

The motivation for this course is the idea that knowledge of classical algorithms is fundamental to any computer science curriculum, but it is not just for programmers and computer-science students. Everyone who uses a computer wants it to run faster or to solve larger problems. The algorithms in the course represent a body of knowledge developed over the last 50 years that has become indispensable. As the scope of computer applications continues to grow, so grows the impact of these basic methods. Our experience in developing this course over several decades has shown that the most effective way to teach these methods is to integrate them with applications as students are first learning to tackle significant programming problems, as opposed to the oft-used alternative where they are taught in a theory course. With this approach, we are reaching four times as many students as do typical algorithms courses. Furthermore, our CS majors have a solid knowledge of the algorithms when they later learn more about their theoretical underpinnings, and all our students have an understanding that efficient algorithms are necessary in many contexts.

**Body of Knowledge coverage**

| KA | Knowledge Unit | Topics Covered | Hours |
|---|---|---|---|
| SDF | Algorithms and Design | Encapsulation, separation of behavior and implementation. | 1 |
| PL | Object-oriented programming | Object-oriented design, encapsulation, iterators. | 1 |

| | | | |
|---|---|---|---|
| SDF | Fundamental data structures | Stacks, queues, priority queues, references, linked structures, resizable arrays. | 2 |
| AL | Basic Analysis | Asymptotic analysis, empirical measurements. Differences among best, average, and worst case behaviors of an algorithm. Complexity classes, such as constant, logarithmic, linear, quadratic, and exponential. Time and space trade-offs in algorithms. | 1 |
| AL | Algorithmic Strategies | Brute-force, greedy, divide-and-conquer, and recursive algorithms. Dynamic programming, reduction. | 2 |
| AL | Fundamental Data Structures and Algorithms | Binary search. Insertion sort, selection sort, shellsort, quicksort, mergesort, heapsort. Binary heaps. Binary search trees, hashing. Representations of graphs. Graph search, union-find, minimum spanning trees, shortest paths. Substring search, pattern matching. | 13 |
| AL | Basic Automata, Computability and Complexity | Finite-state machines, regular expressions, P vs. NP, NP-completeness, NP-complete problems | 3 |
| AL | Advanced Automata, Computability and Complexity | Languages, DFAs, NFAs, equivalence of NFAs and DFAs. | 1 |
| AL | Advanced Data Structures and Algorithms | Balanced trees, B-trees. Topological sort, strong components, network flow. Convex hull. Geometric search and intersection. String sorts, tries, Data compression. | 9 |

**Additional topics**

Use of scientific method to validate hypotheses about an algorithm's time and space usage.

# COSC/MATH 201: Modeling and Simulation for the Sciences

**Wofford College**
**Angela B. Shiflet**
**http://www.wofford.edu/ecs/**
**(Description below based on the Fall 2011 and 2012 offerings)**

**Knowledge Areas with topics and learning outcomes covered in the course**

| Knowledge Area | Total Hours of Coverage |
|---|---|
| Computational Science (CN) | 35.5 |
| Intelligent Systems (IS) | 3 |
| Software Development Fundamentals (SDF) | 2 |
| Software Engineering (SE) | 1 |
| Graphics and Visualization (GV) | 0.5 |

**Where does the course fit in your curriculum?**

Modeling and Simulation for the Sciences (COSC/MATH 201) has a pre-requisite of Calculus I. However, because the course does not require derivative or integral formulas but only an understanding of the concept of "rate of change," students with no calculus background have taken the course successfully. The course has been offered since the spring of 2001. Dual-listed as a computer science and a mathematics course and primarily targeted at second-year science majors, the course is required for Wofford College's Emphasis in Computer Science (see "Other comments" below). Moreover, Modeling and Simulation meets options for the computer science, mathematics, and environmental studies majors and counts for both computer science and mathematics minors.

Wofford College has a thirteen-week semester with an additional week for final exams. Modeling and Simulation for the Sciences has 3-semester-hours credit and meets 3 contact hours per week.

**What is covered in the course?**

- The modeling process
- Two system dynamics tool tutorials
- System dynamics problems with rate proportional to amount: unconstrained growth and decay, constrained growth, drug dosage
- System dynamics models with interactions: competition, predator-prey models, spread of disease models
- Computational error
- Simulation techniques: Euler's method, Runge-Kutta 2 method
- Additional system dynamics projects throughout, such as modeling falling and skydiving, enzyme kinetics, the carbon cycle, economics and fishing
- Six computational toolbox tutorials
- Empirical models
- Introduction to Monte Carlo simulations
- Cellular automaton random walk simulations
- Cellular automaton diffusion simulations: spreading of fire, formation of biofilms
- High-performance computing: concurrent processing, parallel algorithms

- Additional cellular automaton simulations throughout such as simulating polymer formation, solidification, foraging, pit vipers, mushroom fairy rings, clouds

### What is the format of the course?

Usually, students are assigned reading of the material before consideration in class. Then, after questions are discussed, students often are given a short quiz taken directly from the quick review questions. Answers to these questions are available at the end of each module. After the quiz, usually the class develops together an extension of a model in the textbook. Class time is allotted for the first system dynamics tutorial and the first computational toolbox tutorial. Students work on the remaining tutorials and open-ended projects, often in pairs, primarily outside of class and occasionally in class.

### How are students assessed?

Students complete two system dynamics tutorials and six computational toolbox tutorials with at least one of each type of tutorial in a lab situation. The students have approximately one project assignment per week during a thirteen-week semester. Most assignments are completed in teams of two or three students. Generally, a submission includes a completed model, results, and discussion. Students present their models at least twice during the semester. Daily quizzes occur on the quick review questions, and tests comprise a midterm and a final.

### Course textbooks and materials

Textbook: *Introduction to Computational Science: Modeling and Simulation* by Angela B. Shiflet and George W. Shiflet, Princeton University Press, with online materials available at the above website.

A second edition of the textbook is nearing completion and will include new chapters on agent-based modeling and modeling with matrices along with ten additional project-based modules and more material on high performance computing.

The first half of the semester on system dynamics uses STELLA or Vensim; and the second half of the semester on empirical modeling and cellular automaton simulations employs Mathematica or MATLAB. (Tutorials and files are available on the above website in these tools and also in Python, R, Berkeley Madonna, and Excel for system dynamics models and in Python, R, Maple, NetLogo, and Excel for the material for the second half of the semester.)

### Why do you teach the course this way?

The course has evolved since its initial offering in 2001, and the vast majority students, who have a variety of majors in the sciences, mathematics, and computer science, are successful in completing the course with good grades. Moreover, many of the students have used what they have learned in summer internships involving computation in the sciences.

**Body of Knowledge coverage**

| KA | Knowledge Unit | Topics Covered | Hours |
|---|---|---|---|
| CN | Fundamentals | Models as abstractions of situations<br>Simulations as dynamic modeling<br>Simulation techniques and tools, such as physical simulations and human-in-the-loop guided simulations.<br>Foundational approaches to validating models | 3 |
| CN | Modeling and Simulation | Purpose of modeling and simulation including optimization; supporting decision making, forecasting, safety considerations; for training and education.<br>Tradeoffs including performance, accuracy, validity, and complexity.<br>The simulation process; identification of key characteristics or behaviors, simplifying assumptions; validation of outcomes. | 29 |

| | | | |
|---|---|---|---|
| | | Model building: use of mathematical formula or equation, graphs, constraints; methodologies and techniques; use of time stepping for dynamic systems.<br>Formal models and modeling techniques: mathematical descriptions involving simplifying assumptions and avoiding detail. The descriptions use fundamental mathematical concepts such as set and function.<br>Random numbers. Examples of techniques including:<br>Monte Carlo methods<br>Stochastic processes<br>Graph structures such as directed graphs, trees, networks<br>Differential equations: ODE<br>Non-linear techniques<br>State spaces and transitions<br>Assessing and evaluating models and simulations in a variety of contexts; verification and validation of models and simulations.<br>Important application areas including health care and diagnostics, economics and finance, city and urban planning, science, and engineering.<br>Software in support of simulation and modeling; packages, languages. | |
| CN | Processing | Fundamental programming concepts, including:<br>The process of converting an algorithm to machine-executable code.<br>Software processes including lifecycle models, requirements, design, implementation, verification and maintenance.<br>Machine representation of data computer arithmetic, and numerical methods, specifically sequential and parallel architectures and computations.<br>The basic properties of bandwidth, latency, scalability and granularity.<br>The levels of parallelism including task, data, and event parallelism. | 3 |
| CN | Interactive Visualization | Image processing techniques, including the use of standard APIs and tools to create visual displays of data | 0.5 |
| GV | Fundamental Concepts | Applications of computer graphics: including visualization,. | 0.5 |
| SDF | Development Methods | Program comprehension<br>Program correctness<br>Types or errors (syntax, logic, run-time)<br>The role and the use of contracts, including pre- and post-conditions<br>Unit testing<br>Simple refactoring<br>Debugging strategies<br>Documentation and program style | 2 |
| IS | Agents | Definitions of agents<br>Agent architectures (e.g., reactive, layered, cognitive, etc.)<br>Agent theory<br>Biologically inspired models | Possibly 3 |
| SE | Software Design | The use of components in design: component selection, design, adaptation and assembly of components, components, components. | 1 |

**Additional topics**

Successful course participants will:

- Understand the modeling process
- Be able to develop and analyze systems dynamics models and Monte Carlo simulations with a team
- Understand the concept of rate of change
- Understand basic system dynamics models, such as ones for unconstrained and constrained growth, competition, predator-prey, SIR, enzyme kinetics
- Be able to perform error computations
- Be able to use Euler's and Runge-Kutta 2 Methods
- Be able to develop an empirical model from data
- Understand basic cellular automaton simulations, such as ones for random walks, diffusion, and reaction-diffusion
- Be able to verify and validate models
- Understand basic hardware and programming issues of high performance computing
- Be able to use a system dynamics tool, such as Vensim, STELLA, or Berkeley Madonna
- Be able to use a computational tool, such as MATLB, Mathematica, or Maple.

**Other comments**

Wofford College's Emphasis in Computational Science (ECS), administered by the Computer Science Department, is available to students pursuing a B.S. in a laboratory science, mathematics, or computer science. The Emphasis requires five courses—Modeling and Simulation for the Sciences (this course), CS1, CS2, Calculus I, and Data and Visualization (optionally in 2013, Bioinformatics or High Performance Computing)—and a summer internship involving computation in the sciences. Computer science majors obtaining the ECS must also complete 8 additional semester hours of a laboratory science at the 200+ level. Note: Data and Visualization covers creation of Web-accessible scientific databases, a dynamic programming technique of genomic sequence alignment, and scientific visualization programming in C with OpenGL.

# CPSC 3380 Operating Systems
# University of Arkansas at Little Rock

**Dr. Peiyi Tang**
**pxtang@ualr.edu**

**Permanent URL where additional materials and information is available (can be course website for a recent offering assuming it is public)**

**Knowledge Areas that contain topics and learning outcomes covered in the course**

| **Knowledge Area** | **Total Hours of Coverage** |
|---|---|
| Operating Systems (OS) | 24 |
| Parallel and Distributed Computing (PD) | 5 |
| System Fundamentals | 4 |

**Brief description of the course's format and place in the undergraduate curriculum**

## Course description and goals

An operating system (OS) defines an abstraction of hardware and manages resource sharing among the computer's users. The OS shares the computational resources such as memory, processors, networks, etc. while preventing individual programs from interfering with one another. After successful completion of the course, students will learn how the programming languages, architectures, and OS interact.

## Course topics

After a brief history and evolution of OS, the course will cover the major components of OS. Topics will include process, thread, scheduling, concurrency (exclusion and synchronization), deadlock (prevention, avoidance, and detection), memory management, IO management, file management, virtualization, and OS' role for realizing distributed systems. The course will also cover protection and security with respect to OS.

## Course textbooks, materials, and assignments

Textbook: “Operating System Concepts" by A.Silberschatz, P.\ Galvin and G.\ Gangne, John Wiley \& Sons, 2009, ISBN: 978-0-470-12872-5

Lab System: Nachos 4.3 (in C++).

Assignment One: Program in Execution
Assignment Two: Process and Thread
Assignment Three: Synchronization with Semaphores
Assignment Four: Synchronization with Monitors

**Body of Knowledge coverage**

| KA | Knowledge Unit | Topics Covered | Hours |
|---|---|---|---|
| *OS* | *OS/Overview of Operating Systems* | *Overview of OS basics* | *2* |
| *OS* | *OS/Operating System Principles* | *Processes, process control, threads* | *2* |
| *OS* | *OS/Scheduling and Dispatch* | *Preemptive, non-preemptive scheduling, schedulers and policies, real-time scheduling* | *3* |
| *SF* | *SF/Resource Allocation and Scheduling* | *Kinds of scheduling* | *2* |
| *OS* | *OS/Concurrency* | *Exclusion and synchronization; deadlock* | *3* |
| *OS* | *OS/Memory Management* | *Memory management, working sets and thrashing; caching* | *3* |
| *OS* | *OS/File Systems* | *Files (metadata, operations, organization, etc.); standard implementation techniques; file system partitioning; virtual file systems; memory mapped files, journaling and log structured file systems* | *2* |
| *SF* | *SF/Virtualization & Isolation* | *Rationale for protection and predictable performance, levels of indirection, methods for implementing virtual memory and virtual machines* | *2* |
| *OS* | *OS/Virtual Machines* | *Paging and virtual memory, virtual file systems, virtual file, portable virtualization, hypervisors* | *2* |
| *OS* | *OS/Device Management* | *Characteristics of serial & parallel devices, abstracting device differences, direct memory access, recovery from failures* | *3* |
| *PD* | *PD/Parallelism Fundamentals* | *Multiple simultaneous computations; programming constructs for creating parallelism, communicating, and coordinating;* | *2* |
| *PD* | *PD/Distributed Systems* | *Distributed message sending; distributed system and service design;* | *3* |
| *OS* | *OS/Security and Protection* | *Overview of system security, policy, access control, protection, authentication* | *2* |
| *OS* | *OS/Real Time and Embedded Systems* | *Memory/disk management requirements in real-time systems; failures, risks, recovery; special concerns in real-time systems* | *2* |

# CS150: Digital Components and Design

University of California, Berkeley
Randy H. Katz
randy@cs.Berkeley.edu

http://inst.eecs.berkeley.edu/~cs150/

**Knowledge Areas that contain topics and learning outcomes covered in the course**

| Knowledge Area | Total Hours of Coverage |
|---|---|
| Architecture and Organization (AR) | 37.5 |

### Where does the course fit in your curriculum?

This is a junior-level course in the computer science curriculum for computer engineering students interested in digital system design and implementation.

### What is covered in the course?

Design of synchronous digital systems using modern tools and methodologies, in particular, digital logic synthesis tools, digital hardware simulation tools, and field programmable gate array architectures.

### What is the format of the course?

Lecture, discussion section, laboratory

### How are students assessed?

Laboratories, examinations, and an independent design project

### Course textbook and materials

Harris and Harris, Digital Design and Computer Architecture

### Why do you teach the course this way?

Understand the principles and methodology of digital logic design at the gate and switch level, including both combinational and sequential logic elements. Gain experience developing a relatively large and complex digital system. Gain experience with modern computer-aided design tools for digital logic design. Understand clocking methodologies used to control the flow of information and manage circuit state. Appreciate methods for specifying digital logic, as well as the process by which a high-level specification of a circuit is synthesized into logic networks. Appreciate the tradeoffs between hardware and software implementations of a given function. Appreciate the uses and capabilities of a modern FPGA platform.

**Body of Knowledge coverage**

| KA | Knowledge Unit | Topics Covered | Hours |
|---|---|---|---|
| AR1 | Digital Logic and Digital Systems | Combinational /Sequential<br>Logic Design and CAD Tools;<br>State Machines, Counters;<br>Digital Building Blocks;<br>High Level Design w/Verilog | 4.5<br>4.5<br>3<br>4.5<br>4.5 |
| AR2 | Machine Level Representation of Data | NA | 0 |

| | | | |
|---|---|---|---|
| AR3 | Assembly Level Machine Organization | MIPS Architecture & Project | 3 |
| AR4 | Memory System Organization and Architecture | CMOS/SRAM/DRAM,<br>Video/Frame Buffers | 6 |
| AR5 | Interfacing and Communication | Timing;<br>Synchronization | 3<br>1.5 |
| AR6 | Functional Organization | NA | 0 |
| AR7 | Multprocessing and Alternative Architecture | Graphics Processing Chips | 1.5 |
| AR8 | Performance Enhancements | Power and Energy | 1.5 |

# CC152: Computer Architecture and Engineering


## University of California, Berkeley

**Randy H. Katz**
**randy@cs.Berkeley.edu**


**http://inst.eecs.berkeley.edu/~cs152/**

**Knowledge Areas that contain topics and learning outcomes covered in the course**

| Knowledge Area | Total Hours of Coverage |
|---|---|
| Architecture and Organization (AR) | 33 |

### Where does the course fit in your curriculum?

This is a senior-level course in the computer science curriculum for computer engineering students interested in computer design.

### What is covered in the course?

Historical Perspectives: RISC vs. CISC, Pipelining, Memory Hierarchy, Virtual Memory, Complex Pipelines and Out-of-Order Execution, Superscaler and VLIW Architecture, Synchronization, Cache Coherency.

### What is the format of the course?

Lectures, Discussion, Laboratories, and Examinations

### How are students assessed?

Examinations, homeworks, and hands-on laboratory exercises

### Course textbook and materials

J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 5th Edition, Morgan Kaufmann Publishing Co., Menlo Park, CA. 2012.

### Why do you teach the course this way?

The course is intended to provide a foundation for students interested in performance programming, compilers, and operating systems, as well as computer architecture and engineering. Our goal is for you to better understand how software interacts with hardware, and to understand how trends in technology, applications, and economics drive continuing changes in the field. The course will cover the different forms of parallelism found in applications (instruction-level, data-level, thread-level, gate-level) and how these can be exploited with various architectural features. We will cover pipelining, superscalar, speculative and out-of-order execution, vector machines, VLIW machines, multithreading, graphics processing units, and parallel microprocessors. We will also explore the design of memory systems including caches, virtual memory, and DRAM. An important part of the course is a series of lab assignments using detailed simulation tools to evaluate and develop architectural ideas while running real applications and operating systems. Our objective is that you will understand all the major concepts used in modern microprocessors by the end of the semester.

**Body of Knowledge coverage**

| KA | Knowledge Unit | Topics Covered | Hours |
|---|---|---|---|
| AR1 | Digital Logic and Digital Systems | NA | 0 |
| AR2 | Machine Level Representation of Data | NA | 0 |

| | | | |
|---|---|---|---|
| AR3 | Assembly Level Machine Organization | Historical Perspectives | 6 |
| AR4 | Memory System Organization and Architecture | Memory Hierarchy,<br>Virtual Memory,<br>Snooping Caches | 9 |
| AR5 | Interfacing and Communication | Synchronization,<br>Sequential Consistency | 3 |
| AR6 | Functional Organization | Pipelining | 3 |
| AR7 | Multprocessing and Alternative Architecture | Superscalar, VLIW,<br>Vector Processing | 6 |
| AR8 | Performance Enhancements | Complex Pipelining | 3 |

**Additional topics**

Case Study: Intel Sandy Bridge & AMD Bulldozer (1.5); Warehouse-Scale Computing (1.5)

# CS2200: Introduction to Systems and Networking


## Georgia Institute of Technology, Atlanta, GA

**Kishore Ramachandran**

**rama@gatech.edu**

[http://www.cc.gatech.edu/~rama/CS2200-External/](http://www.cc.gatech.edu/~rama/CS2200-External/)


**Knowledge Areas that contain topics and learning outcomes covered in the course**

| Knowledge Area | Total Hours of Coverage |
|---|---|
| Systems Fundamentals (SF) | 42 |

**Where does the course fit in your curriculum?**

This course is taken in the second semester of the sophomore year. The pre-requisite for this course is a good knowledge of C and logic design. It is required for all CS majors wishing to specialize in: operating systems, architecture, programming language and compilers, system security, and networking. It provides a thorough understanding of the hardware and the system software and the symbiotic inter-relationship between the two. Students may take advanced courses in operating systems, architecture, and networking following this course in the junior and senior years. The course is offered 3 times a year (Fall, Spring, Summer) and the enrolment is typically around 100 students.

**What is covered in the course?**

The course represents a novel integrated approach to presenting side by side both the architecture and the operating system of modern computer systems, so that students learn how the two complement each other in making the computer what it is. The course consists of five modules, corresponding to the five major building blocks of any modern computer system: processor, memory, parallelism, storage, and networking. Both the hardware and system software issues are covered concomitantly in presenting the five units. Topics covered include

- Processor design including instruction-set design, processor implementation (simple as well as pipelined with the attendant techniques for overcoming different kinds of hazards), processor performance (CPI, IPC, execution time, Amdahl's law), dealing with program discontinuities (interrupts, traps, exceptions), and design of interrupt handlers
- Processor scheduling algorithms including FCFS, SJF, priority, round robin, with Linux scheduler as a real world example
- Memory system including principles of memory management in general (paging in particular) and the necessary hardware support (page tables, TLB), page replacement algorithms, working set concepts, the inter-relationship between memory management and processor scheduling, thrashing, and context switching overheads
- Memory hierarchy including different organizations of processor caches, the path of memory access from the processor through the different levels of the memory hierarchy, interaction between virtual memory and processor caches, and page coloring
- Parallel programming (using pthreads), basic synchronization (mutex locks, condition variables) and communication (shared memory), program invariants, OS support for parallel programming, hardware support for parallel programming, rudiments of multiprocessor TLB and cache consistency
- Basics of I/O (programmed data transfer, DMA), interfacing peripherals to the computer, structure of device driver software
- Storage subsystem focusing on hard disk (disk scheduling), file systems (naming, attributes, APIs, disk allocation algorithms), example file systems (FAT, ext2, NTFS)

- Networking subsystem focusing on the transport layer protocols (stop and wait, pipelined, congestion control, windowing) , network layer protocols (Dijkstra, distance vector) and service models (circuit-, message-, and packet-switching), link layer protocols (Ethernet, token ring)
- Networking gear (NIC, hubs/repeater, bridge, switch, VLAN)
- Performance of networking (end-to-end latency, throughput, queuing delays, wire delay, time of flight, protocol overhead).

**What is the format of the course?**

Three hours of lecture per week; Two hours of TA-led recitation per week to provide help on homeworks and projects; and 3 hours of unsupervised laboratory per week. Video recordings of classroom lectures (from past offering) available as an additional study aid.

**How are students assessed?**

Two midterms, one final, 5 homeworks, 5 projects (two architecture projects: processor datapath and control implementation, and augmenting processor to handle interrupts; three OS projects: paged virtual memory management, multithreaded processor scheduler using pthreads, and reliable transport layer implementation using pthreads). Plus an extra-credit project (a uniprocessor cache simulator).

**Course textbooks and materials**

Ramachandran and Leahy Jr., Computer Systems: An Integrated Approach to Architecture and Operating Systems, Addison-Wesley, 2010.

**Why do you teach the course this way?**

There is excitement when you talk to high school students about computers. There is a sense of mystery as to what is “inside the box” that makes the computer do such things as play video games with cool graphics, play music—be it rap or symphony—send instant messages to friends, and so on. What makes the box interesting is not just the hardware, but also how the hardware and the system software work in tandem to make it all happen. Therefore, the path we take in this course is to look at hardware and software together to see how one helps the other and how together they make the box interesting and useful. We call this approach “unraveling the box”—that is, resolving the mystery of what is inside the box: We look inside the box and understand how to design the key hardware elements (processor, memory, and peripheral controllers) and the OS abstractions needed to manage all the hardware resources inside a computer, including processor, memory, I/O and disk, multiple processors, and network. Since the students take this course in their sophomore year, it also whets the appetite of the students and gets them interested in systems early so that they can pursue research as undergraduates in systems. The traditional silo model of teaching architecture and operating systems in later years (junior/senior) restricts this opportunity. The course was first offered in Fall 1999. It has been offered 3 times every year ever since. Over the years, the course has been taught by a variety of faculty specializing in architecture, operating systems, and networking. Thus the content of the course has been revised multiple times; the most recent revision was in 2010. It is a required course and it has gotten a reputation as a “tough” one, and some students end up taking it multiple times to pass the course with the required “C” passing grade.

**Body of Knowledge coverage**

| KA | Knowledge Unit | Topics Covered | Hours |
|---|---|---|---|
| SF1 | Processor architecture | HLL constructs and Instruction-set design, datapath and control, microprogrammed implementation, example (MIPS) | 6 |
| SF2 | Program discontinuities | Interrupts, traps, exceptions, nesting of interrupts, hardware for dealing with interrupts, interrupt handler code | 2 |
| SF3 | Processor performance metrics | Space and time, memory footprint, execution time, instruction frequency, IPC, CPI, SPECratio, speedup, Amdahl's law | 1 |
| SF4 | Principles of pipelining | Hardwired control, datapath of pipeline stages, pipeline registers, hazards (structural, data, and control) and solutions thereof (redundant hardware, register forwarding, branch prediction), example (Intel Core microarchitecture) | 5 |
| SF5 | Processor Scheduling | Process context block, Types of schedulers (short-, medium-, long-term), preemptive vs. non-preemptive schedulers, short-term scheduling algorithms (FCFS, SJF, SRTF, priority, round robin), example (Linux O(1) scheduler) | 2 |
| SF6 | Scheduling performance metrics | CPU utilization, throughput, response time, average response time/waiting time, variance in response time, starvation | 1 |
| SF7 | Memory management | Process address space, static and dynamic relocation, memory allocation schemes (fixed and variable size partitions), paging, segmentation | 2 |
| SF8 | Page-based memory management | Demand paging, hardware support (page tables, TLB), interaction with processor scheduling, OS data structures, page replacement algorithms (Belady's Min, FIFO, LRU, clock), thrashing, working set, paging daemon | 2 |
| SF9 | Processor caches | Spatial and temporal locality, cache organization (direct mapped, fully associative, set associative), interaction with virtual memory, virtually indexed physically tagged caches, page coloring | 3 |
| SF10 | Main memory | DRAM, page mode DRAM, Memory buses | 0.5 |
| SF11 | Memory system performance metrics | Context switch overhead, page fault service time, memory pressure, effective memory access time, memory stalls | 0.5 |
| SF12 | Parallel programming | Programming with pthreads, synchronization constructs (mutex locks and condition variables), data races, deadlock and livelock, program invariants | 3 |
| SF13 | OS support for parallel programming | Thread control block, thread vs. process, user level threads, kernel level threads, scheduling threads, TLB consistency | 1.5 |
| SF14 | Architecture support for parallel programming | Symmetric multiprocessors (SMP), atomic RMW primitives, T&S instruction, bus-based cache consistency protocols | 1.5 |
| SF15 | Input/output | Programmed data transfer, DMA, I/O buses, interfacing peripherals to the computer, structure of device driver software | 1.5 |

| | | | |
|---|---|---|---|
| SF16 | Disk subsystem | Disk scheduling algorithms (FCFS, SSTF, SCAN, LOOK), disk allocation algorithms (contiguous, linked list, FAT, indexed, multi-level indexed, hybrid indexed) | 1.5 |
| SF17 | File systems | Naming, attributes, APIs, persistent and in-memory data structures, journaling, example file systems (FAT, ext2, NTFS) | 2 |
| SF18 | Transport layer | 5-layer Internet Protocol Stack, OSI model, stop-and-wait, pipelined, sliding window, congestion control, example protocols (TCP, UDP) | 2 |
| SF19 | Network layer | Dijkstra's link state and distance vector routing algorithms, service models (circuit-, message-, and packet-switching), Internet addressing, IP network | 2 |
| SF20 | Link layer | Ethernet, CSMA/CD, wireless LAN, token ring | 0.5 |
| SF21 | Networking gear | NIC, hub/repeater, bridge, switch, router, VLAN | 1 |
| SF22 | Network performance | End-to-end latency, throughput, queuing delays, wire delay, time of flight, protocol overhead | 0.5 |

# CS 420, Operating Systems

**Embry-Riddle Aeronautical University**
**Prof. Nick Brixius**
**brixiusn@erau.edu**

**Knowledge Areas that contain topics and learning outcomes covered in the course**

| **Knowledge Area** | **Total Hours of Coverage** |
|---|---|
| Operating Systems | 24 |
| Parallel and Distributed Computing (PD) | 5 |
| System Fundamentals | 4 |

**Brief description of the course's format and place in the undergraduate curriculum**

Third or fourth year course – prerequisite is CS225, Computer Science II and third year standing – 3 semester credits – 3 hours lecture per week.

**Course description and goals**

The course will study the basic concepts, design and implementation of operating systems. Topics to be covered include an overview of basic computing hardware components, operating system structures, process management, memory management, file systems, input/output systems, protection and security. The Windows and UNIX/Linux operating systems will be reviewed as implementation examples.

The coursework will include "hands-on" application of reading assignments and lecture material through homework assignments, including several programming projects.

**Course topics**

1. Overview of an Operating System
2. Computing Hardware Overview
3. Process Management
4. CPU Scheduling
5. Deadlocks and Synchronization
6. Memory Management
7. File systems and storage
8. Distributed Systems

**Course textbooks, materials, and assignments**

Textbook: Silberschatz, A., Galvin, P.B. and Gagne, G. (2010) *Operating System Concepts with Java.* Addison Wesley Publishing Co., New York. (Eighth Edition) ISBN 978-0-470-50949-4

Java and the Java Virtual Machine are used for programming assignments

Assignment One: Java threads, OS components
Assignment Two: Process states and process management
Assignment Three: Process Scheduling and race conditions
Assignment Four: Concurrency and deadlocks
Assignment Five: Memory management

Assignment Six: File systems and HDD scheduling
Assignment Seven: Network support and distributed systems

**Body of Knowledge coverage**

| KA | Knowledge Unit | Topics Covered | Hours |
|---|---|---|---|
| OS | OS/Overview of Operating Systems | High-level overview of all topics | 2 |
| OS | OS/Operating System Principles | Processes, process control, threads. | 2 |
| OS | OS/Scheduling and Dispatch | Preemptive, non-preemptive scheduling, schedulers and policies, real-time scheduling | 3 |
| SF | SF/Resource Allocation and Scheduling | Kinds of scheduling | 2 |
| OS | OS/Concurrency | Exclusion and synchronization; deadlock | 3 |
| OS | OS/Memory Management | Memory mamagement, working sets and thrashing; caching | 3 |
| OS | OS/File Systems | Files (metadata, operations, organization, etc.); standard implementation techniques; file system partitioning; virtual file systems; memory mapped files, journaling and log structured file systems | 2 |
| SF | SF/Virtualization & Isolation | Rationale for protection and predictable performance, levels of indirection, methods for implementing virtual memory and virtual machines | 2 |
| OS | OS/Virtual Machines | Paging and virtual memory, virtual file systems, virtual file, portable virtualization, hypervisors | 2 |
| OS | OS/Device Management | Characteristics of serial & parallel devices, abstracting device differences, direct memory access, recovery from failures | 3 |
| PD | PD/Parallelism Fundamentals | multiple simultaneous computations; programming constructs for creating parallelism, communicating, and coordinating; | 2 |
| PD | PD/Distributed Systems | Distributed message sending; distributed system and service design; | 3 |
| OS | OS/Security and Protection | Overview of system security, policy, access control, protection, authentication | 2 |
| OS | OS/Real Time and Embedded Systems | Memory/disk management requirements in real-time systems; failures, risks, recovery; special concerns in real-time systems | 2 |

# CS/ECE 552: Introduction to Computer Architecture
## University of Wisconsin, Computer Sciences Department
**sohi@cs.wisc.edu**

**http://pages.cs.wisc.edu/~karu/courses/cs552/spring2011/wiki/index.php/Main/Syllabus**

**Knowledge Areas that contain topics and learning outcomes covered in the course**

| **Knowledge Area** | **Total Hours of Coverage** |
|---|---|
| Architecture and Organization | 39 |

### Where does the course fit in your curriculum?
This is taken by juniors, seniors, and beginning graduate students in computer science and computer engineering. Prerequisites include courses that cover assembly language and logic design. This course is a (recommended) prerequisite for a graduate course on advanced computer architecture. Approximately 60 students take the course per offering; it is offered two times per year (once each semester).

### What is covered in the course?
The goal of the course is to teach the design and operation of a digital computer. It serves students in two ways. First, for those who want to continue studying computer architecture, embedded systems, and other low-level aspects of computer systems, it lays the foundation of detailed implementation experience needed to make the quantitative tradeoffs in more advanced courses meaningful. Second, for those students interested in other areas of computer science, it solidifies an intuition about why hardware is as it is and how software interacts with hardware.

The subject matter covered in the course includes technology trends and their implications, performance measurement, instruction sets, computer arithmetic, design and control of a datapath, pipelining, memory hierarchies, input and output, and brief introduction to multiprocessors.

The full list of course topics is:
Introduction and Performance
- Technology trends
- Measuring CPU performance
- Amdahl's law and averaging performance metrics

Instruction Sets
- Components of an instruction set
- Understanding instruction sets from an implementation perspective
- RISC and CISC and example instruction sets

Computer Arithmetic
- Ripple carry, carry lookahead, and other adder designs
- ALU and Shifters
- Floating-point arithmetic and floating-point hardware design

Datapath and Control
- Single-cycle and multi-cycle datapaths
- Control of datapaths and implementing control finite-state machines

Pipelining

- Basic pipelined datapath and control
- Data dependences, data hazards, bypassing, code scheduling
- Branch hazards, delayed branches, branch prediction

Memory Hierarchies

- Caches (direct mapped, fully associative, set associative)
- Main memories
- Memory hierarchy performance metrics and their use
- Virtual memory, address translation, TLBs

Input and Output

- Common I/O device types and characteristics
- Memory mapped I/O, DMA, program-controlled I/O, polling, interrupts
- Networks

Multiprocessors

- Introduction to multiprocessors
- Cache coherence problem

## What is the format of the course?

The course is 15 weeks long, with students meeting for three 50-minute lectures per week or two 75-minute lectures per week. If the latter, the course is typically "front loaded" so that lecture material is covered earlier in the semester and students are able to spend more time later in the semester working on their projects.

## How are students assessed?

Assessment is a combination of homework, class project, and exams. There are typically six homework assignments. The project is a detailed implementation of a 16-bit computer for an example instruction set. The project requires both an unpipelined as well as a pipelined implementation ans typically takes close to a hundred hours of work to complete successfully. The project and homeworks are typically done by teams of 2 students. There is a midterm exam and a final exam, each of which is typically 2 hours long.

## Course textbooks and materials

David A. Patterson and John L. Hennessy, Computer Organization and Design: The Hardware and Software Interface Morgan Kaufmann Publishers,
***Fourth Edition***. ISBN: 978-0-12-374493-7

## Why do you teach the course this way?

Since the objective is to teach how a digital computer is designed and built and how it executes programs, we want to show how basic logic gates could be combined to construct building blocks which are then combined to work together to execute programs written in a machine language. The students learn the concepts of how to do so in the classroom, and then apply them in their project. Having taken this course a student can go into an industrial environment and be ready to participate in the design of complex digital.

**Body of Knowledge coverage**

| **KA** | **Knowledge Unit** | **Topics Covered** | **Hours** |
|---|---|---|---|
| AR | Introductory Material and Performance | Technology trends, measuring CPU performance, Amdahl's law and averaging performance metrics | 3 |
| AR | Instruction Set Architecture | Components of instruction sets, understanding instruction sets from an implementation perspective, | 3 |

| | | | |
|---|---|---|---|
| | | RISC and CISC and example instruction sets | |
| AR | Computer Arithmetic | Ripple carry, carry lookahead, and other adder designs, ALU and Shifters, floating-point arithmetic and floating-point hardware design | 6 |
| AR | Datapath and Control | Single-cycle and multi-cycle datapaths, control of datapaths and implementing control finite-state machines | 4 |
| AR | Pipelined Datapaths and Control | Basic pipelined datapath and control, data dependences, data hazards, bypassing, code scheduling, branch hazards, delayed branches, branch prediction | 8 |
| AR | Memory Hierarchies | Caches (direct mapped, fully associative, set associative), main memories, memory hierarchy performance metrics and their use, virtual memory, address translation, TLBs | 9 |
| AR | Input and Output | Common I/O device types and characteristics, memory mapped I/O, DMA, program-controlled I/O, polling, interrupts, networks | 3 |
| AR | Multiprocessors | Introduction to multiprocessors, cache coherence problem | 3 |

# CS61C: Great Ideas in Computer Architecture

## University of California, Berkeley

**Randy H. Katz**

**randy@cs.Berkeley.edu**

**http://inst.eecs.berkeley.edu/~cs61c/**

**Knowledge Areas that contain topics and learning outcomes covered in the course**

| Knowledge Area | Total Hours of Coverage |
| --- | --- |
| Systems Fundamentals (SF) | 39 |

### Where does the course fit in your curriculum?

This is a third course in the computer science curriculum for intended majors, following courses in "great ideas in Computer Science" and "Programming with Data Structures." It provides a foundation for all of the upper division systems courses by providing a thorough understanding of the hardware-software interface, the broad concepts of parallel programming to achieve scalable high performance, and hands-on programming experience in C.

### What is covered in the course?

Introduction to C: this includes coverage of the Hardware/Software Interface (e.g., machine and assembly language formats, methods of encoding instructions and data, and the mapping processes from high level languages, particularly C, to assembly and machine language instructions). Computer architectures: how processors interpret/execute instructions, Memory Hierarchy, Hardware Building Blocks, Single CPU Datapath and Control, and Instruction Level Parallelism. The concept of parallelisms, in particular, task level parallelism, illustrated with Map-Reduce processing; Data Level Parallelism, illustrated with the Intel SIMD instruction set; Thread Level Parallelism/multicore programming, illustrated with openMP extensions to the C programming language.

### What is the format of the course?

Three hours of lecture per week, one hour of TA-led discussion per week, two hours of laboratory per week.

### How are students assessed?

Laboratories, Homeworks, Exams, Four Projects (Map-Reduce application on Amazon EC2), MIPS Instruction Set Emulator in C, Memory and Parallelism-Aware Application Improvement, Logic Design and Simulation of a MIPS processor subset).

### Course textbooks and materials

Patterson and Hennessy, *Computer Organization and Design*, revised 4th Edition, 2012; Kernighan and Ritchie, *The C Programming Language*, 2nd Edition; Borroso, *The Datacenter as a Computer*, Morgan and Claypool publishers.

### Why do you teach the course this way?

The overarching theme of the course is the hardware-software interface, in particular, focusing on what a programmer needs to know about the underlying hardware to achieve high performance for his or her code. Generally, this concentrates on harnessing parallelism, in particular, task level parallelism (map-reduce), data level parallelism (SIMD instruction sets), multicore (openMP), and processor instruction pipelining. The six "great" ideas presented in the course are (1) Layers of Representation/Interpretation, (2) Moore's Law, (3) Principle of Locality/Memory Hierarchy, (4) Parallelism, (5) Performance Evaluation, and (6) Dependability via Redundancy.

**Body of Knowledge coverage**

| KA | Knowledge Unit | Topics Covered | Hours |
|---|---|---|---|
| SF1 | Computational Paradigms | C Programming/Cloud | 10.5 |
| SF2 | Cross-Layer Communications | Compilation/Interpretation | 1.5 |
| SF3 | State-State Transition-State Machines | Building Blocks, Control, Timing | 4.5 |
| SF4 | Parallelism | Task/Data/Thread/Instruction | 10.5 |
| SF5 | Performance | Figures of merit, measurement | 1.5 |
| SF6 | Resource Allocation and Scheduling | | 0 |
| SF7 | Proximity | Memory Hierarchy | 4.5 |
| SF8 | Virtualization and Isolation | Virtual Machines and Memory | 3.0 |
| SF9 | Reliability Through Redundancy | RAID, ECC | 3.0 |

# CS 662; Artificial Intelligence Programming

## University of San Francisco

**Christopher Brooks**

cbrooks@usfca.edu

**https://sierra.cs.usfca.edu**

**Knowledge Areas that contain topics and learning outcomes covered in the course**

| Knowledge Area | Total Hours of Coverage |
|---|---|
| *Intelligent Systems* | *60* |

**Where does the course fit in your curriculum?**

The course is taken as a senior-level elective, and as a required course for first-year Master's students.

**What is covered in the course?**

An overview of AI, including search, knowledge representation, probabilistic reasoning and decision making under uncertainty, machine learning, and topics from NLP, information retrieval, knowledge engineering and multi-agent systems.

**What is the format of the course?**

Face to face. 4 hours lecture per week. (either 3x65 or 2x105)

**How are students assessed?**

Two midterms and a final. Also, heavy emphasis on programming and implementation of techniques. Students complete 7-9 assignments (see website for examples). Expectation is 3 hours outside of class for every hour of lecture.

**Course textbooks and materials**

Russell and Norvig's AIMA is the primary text. I prepare lots of summary material (see website) and provide students with harness code for their assignments in Python. I also make use of pre-existing packages and tools such as NLTK, Protégé and WordNet when possible.

**Why do you teach the course this way?**

My goals are:

- illustrate the ways in which AI techniques can be used to solve real-world problems. I pick a specific domain (such as Wikipedia, or a map of San Francisco, and have the students apply a variety of techniques to problems in this domain. For example, as the assumptions change, the same SF map problem can be used for search, constraints, MDPs, planning, or learning.
- Provide students with experience implementing these algorithms. Almost all of our students go into industry, and need skill in building systems.
- Illustrate connections between techniques. For example, I discuss decision trees and rule learning in conjunction with forward and backward chaining to show how, once you've decided on a representation, you can either construct rules using human expertise, or else learn them (or both).

**Body of Knowledge coverage**

| KA | Knowledge Unit | Topics Covered | Hours |
|---|---|---|---|
| *IS* | *Fundamental Issues* | | *2* |

| | | | |
|---|---|---|---|
| IS | Basic Search Strategies | A*, BFS, DFS, IDA*, problem spaces, constraints | 8 |
| IS | Basic Knowledge Rep. | Predicate logic, forward chaining, backward chaining, resolution, | 8 |
| IS | Basic Machine Learning | Decision trees, rule learning, Naïve Bayes, precision and accuracy, cross-fold validation | 6 |
| IS | Adv. KR | FOL, inference, ontologies, planning | 6 |
| IS | Advanced Search | Genetic algorithms, simulated annealing | 3 |
| IS | Reasoning Under Uncertainty | Probability, Bayes nets, MDPs, decision theory | 8 |
| IS | NLP | Parsing, chunking, n-grams, information retrieval | 4 |

**Other comments**

I also integrate reflection and pre-post questions – before starting an assignment, students must answer questions (online) about the material and the challenges they expect to see. I ask similar questions afterward, both for assessment and to encourage the students to reflect on design decisions.

# CS1101: Introduction to Program Design


## WPI, Worcester, MA

**Kathi Fisler and Glynis Hamel**

kfisler@cs.wpi.edu, ghamel@cs.wpi.edu

http://web.cs.wpi.edu/~cs1101/a12/


**Knowledge Areas that contain topics and learning outcomes covered in the course**

| Knowledge Area | Total Hours of Coverage |
|---|---|
| Software Development Fundamentals (SDF) | 18 |
| Programming Languages (PL) | 8 |
| Algorithms and Complexity (AL) | 2 |

**Where does the course fit in your curriculum?**

This is the first course in our department's sequence for majors in Computer Science, Robotics Engineering, and Interactive Media and Game Development who do not have much prior programming experience (an alternative course exists for students with AP or similar background). Most students pursuing minors also take this course (before Spring 2013, there was no alternative introductory course for novice programmers). It does not have prerequisites. Enrollment has been 300-400 students per year for each of the last 8 years.

Students who take this course and continue in computing go onto one of (1) a course in Object-Oriented Program Design (for majors and most minors), (2) a non-majors course in C-programming (targeted at students majoring in Electrical and Computer Engineering), or (3) a Visual Basic course for students majoring in Management and Information Systems.

**What is covered in the course?**

Upon completion of this course, the student should be able to:

- Understand when to use and write programs over structures, lists, and trees
- Develop data models for programming problems
- Write recursive and mutually recursive programs using the Racket programming language
- Explain when state is needed in value-oriented programming
- Develop test procedures for simple programs

**Course Topics:**

- Basic data types (numbers, strings, images, booleans)
- Basic primitive operations (and, or, +, etc)
- Abstracting over expressions to create functions
- Documenting and commenting functions
- What makes a good test case and a comprehensive test suite
- Conditionals
- Compound data (records or structs)
- Writing and testing programs over lists (of both primitive data and compound data)
- Writing and testing programs over binary trees
- Writing and testing programs over n-ary trees
- Working with higher-order functions (functions as arguments)
- Accumulator-style programs
- Changing contents of data structures
- Mutating variables

**What is the format of the course?**

The course consists of 4 face-to-face lecture hours and 1 lab hour per week, for each of 7 weeks (other schools teach a superset of the same curriculum on a more standard schedule of 3-hours per week for 14 weeks).

**How are students assessed?**

Students are expected to spend roughly 15 hours per week outside of lectures and labs on the course. We assign one extended and thematically-related set of programming problems per week (7 in total in the WPI format). Students work on a shorter programming assignment during the one hour lab; lab assignments are not graded, and thus students do not usually work on them beyond the lab hour. There are 3 hour long exams. Most students report spending 12-18 hours per week on the programming assignments.

**Course textbooks and materials**

Textbook: How to Design Programs, by Felleisen, Findler, Flatt, and Krishnamurthi. MIT Press. Available (for free) online at www.htdp.org.

Language/Environment: Racket (a variant of Scheme), through the DrRacket programming environment (www.drracket.org). A web-based environment, WeScheme (www.wescheme.org) is also available. Software is cross-platform and available for free.

**Why do you teach the course this way?**

WPI adopted this course in 2004. At the time, our programming sequence started in C++ (for two courses), then covered Scheme and Java (across two more courses). After deciding that C++ was too rough an entry point for novice programmers, we redesigned our sequence around program design goals and a smooth language progression. Our current three-course sequence starts with program design, testing, and core data structures in Racket (a variant of Scheme), followed by object-oriented design, more advanced testing, and more data structures in Java, followed by systems-level programming and larger projects in C and C++. Each course in the sequence exposes more linguistic constructs and programmer responsibilities than the course before.

The "How to Design Programs" curriculum emphasizes data-driven and test-driven program design, following a step-by-step methodology. Given a programming problem, students are asked to complete the following steps in order: (1) define the datatypes, (2) write examples of data in each type, (3) write the skeleton of a function that processes each datatype (using a well-defined, step-by-step process that matches object-oriented design patterns), (4) write the contract/type signature for a specific function, (5) write test cases for that function, (6) fill in the skeleton for the main input datatype to complete the function, and (7) run the test cases.

The most useful feature of this methodology is that it helps pinpoint where students are struggling when writing programs. If a student can't write a test case for a program, he likely doesn't understand what the question is asking: writing test cases early force students to understand the question before writing code. If a student doesn't understand the shape of the input data well enough to write down examples of that data, she is unlikely to be able to write a test case for the program. This methodology helps students and instructors identify where students are actually struggling on individual programs (put differently, it gives a way to respond to the open-ended "my code doesn't work" statement from students). It also provides a way for students to get started even if they are not confident writing code: they can go from a blank page to a program in a sequence of steps that translate nicely to worksheets and other aids.

"How to Design Programs" also emphasizes interesting data structures over lots of programming constructs. Racket has a simple syntax with few constructs. In a typical college-pace course for students with no prior programming experience, students start working with lists after about 6 hours of lecture and with trees after roughly 10-12 hours of lecture. The design and programming process scales naturally to tree-shaped data, rather than require students to learn new programming patterns to handle non-linear data. The process thus lets us weave together programming, program design, and data structures starting in the first course.

Finally, the design process from "How to Design Programs" transitions naturally into object-oriented programming in Java. It takes roughly three lecture hours to teach students how to transform any of their "How to Design Programs" code into well-structured Java programs. Our Java/Object-Oriented Design course therefore starts with students able to program effectively with rich, mutually-recursive class hierarchies after under a week of class time. The natural sequence from "How to Design Programs" into Java is one of the salient curricular features of this course.

More information about the philosophy and resources are online at www.htdp.org.

**Body of Knowledge coverage**

| KA | Knowledge Unit | Topics Covered | Hours |
|---|---|---|---|
| AL | Fundamental Data Structures and Algorithms | Binary Search Trees | 2 |
| PL | Functional Programming | Processing structured data via functions for data with cases for each data variant, first-class functions, function calls have no side effects | 8 |
| SDF | Algorithms and Design | Problem-solving strategies, abstraction, program decomposition | 6 |
| SDF | Fundamental Programming Concepts | All listed except I/O; I/O deferred to subsequent programming courses | 6 |
| SDF | Fundamental Data Structures | Records/structs, Linked Lists; remaining data structures covered in CS2/Object-Oriented Design course | 2 |
| SDF | Development Methods | Testing fundamentals, test-driven development, documentation and program style | 4 |

# CS 175 Computer Graphics

## Harvard University, Cambridge, MA

**Dr. Steven Gortler**

**http://www.courses.fas.harvard.edu/~lib175**

**(description below based on the Fall 2011 offering)**

**Knowledge Areas with topics and learning outcomes covered in the course**

| Knowledge Area | Total Hours of Coverage |
|---|---|
| Graphics and Visualization (GV) | 19 |
| Software Engineering (SE) | 7 |
| Architecture and Organization (AR) | 4 |
| Software Development Fundamentals (SDF) | 2 |

**Where does the course fit in your curriculum?**

This is an elective course taken by students mostly in their third year. It requires C programming experience and familiarity with rudimentary linear algebra. Courses are on a semester system: 12 weeks long with 2 weekly 1.5-hour lectures. This course covers fundamental graphics techniques for the computational synthesis of digital images from 3D scenes. This course is not required but counts towards a breadth in computer science requirement in our program.

**What is covered in the course?**

- Shader-based OpenGL programming
- Coordinate systems and transformations
- Quaternions and the Arcball interface
- Camera modeling and projection
- OpenGL fixed functionality including rasterization
- Material simulation
- Basic and advanced use of textures including shadow mapping
- Image sampling including alpha matting
- Image resampling including mip-maps
- Human color perception
- Geometric representations
- Physical simulation in animation
- Ray tracing

**What is the format of the course?**

This course teaches the use and techniques behind modern shader-based computer graphics using OpenGL. It begins by outlining the basic shader-based programming paradigm. Then it covers coordinates systems and transformations. Special care is taken to develop a systematic way to reason about these ideas. These ideas are extended using quaternions as well as developing a scene graph data structure. The students then learn how to implement a key-frame animation system using splines. The course then covers cameras as well as some of the key fixed-function steps such as rasterization with perspective-correct interpolation.

Next, we cover the basics of shader-based material simulation and the various uses of texture mapping (including environment and shadow mapping). Then, we cover the basics of image sampling and alpha blending. We also cover image reconstruction as well as texture resampling using Mip-Maps.

The course gives an overview to a variety of geometric representations, including details about subdivision surfaces. We also give an overview of techniques in animation and physical simulation. Additional topics include human color perception and ray tracing.

**How are students assessed?**

Students implement a set of 10 to 12 programming and writing assignments.

**Course textbooks and materials**

In 12 weeks, students complete 10 programming projects in C++, and a final project.  Students use *Foundations of 3D Computer Graphics* by S. J. Gortler as their primary textbook.

**Why do you teach the course this way?**

**Body of Knowledge coverage**

| KA | Knowledge Unit | Topics Covered | Hours |
| --- | --- | --- | --- |
| GV | Fundamental Concepts | Basics of Human visual perception (HCI Foundations).<br>Image representations, vector vs. raster, color models, meshes.<br>Forward and backward rendering (i.e., ray-casting and rasterization).<br>Applications of computer graphics: including game engines, cad, visualization, virtual reality.<br>Polygonal representation.<br>Basic radiometry, similar triangles, and projection model.<br>Use of standard graphics APIs (see HCI GUI construction).<br>Compressed image representation and the relationship to information theory.<br>Immediate and retained mode.<br>Double buffering. | 3 |
| GV | Basic Rendering | Rendering in nature, i.e., the emission and scattering of light and its relation to numerical integration.<br>Affine and coordinate system transformations.<br>Ray tracing.<br>Visibility and occlusion, including solutions to this problem such as depth buffering, Painter's algorithm, and ray tracing.<br>The forward and backward rendering equation.<br>Simple triangle rasterization.<br>Rendering with a shader-based API.<br>Texture mapping, including minification and magnification (e.g., trilinear MIP-mapping).<br>Application of spatial data structures to rendering.<br>Sampling and anti-aliasing.<br>Scene graphs and the graphics pipeline. | 10 |
| GV | Geometric Modeling | Basic geometric operations such as intersection calculation and proximity tests<br>Parametric polynomial curves and surfaces.<br>Implicit representation of curves and surfaces.<br>Approximation techniques such as polynomial curves, Bezier curves, spline curves and surfaces, and non-uniform rational basis (NURB) spines, and level set method. | 6 |

| | | | |
|---|---|---|---|
| | | Surface representation techniques including tessellation, mesh representation, mesh fairing, and mesh generation techniques such as Delaunay triangulation, marching cubes. | |
| SDF | Development Methods | Program correctness<br>The concept of a specification<br>Unit testing<br>Modern programming environments, Programming using library components and their APIs<br>Debugging strategies<br>Documentation and program style | 2 |
| AR | Performance enhancements | Superscalar architecture<br>Branch prediction, Speculative execution, Out-of-order execution<br>Prefetching<br>Vector processors and GPUs<br>Hardware support for Multithreading<br>Scalability | 3 |
| CN | Modeling and Simulation | Formal models and modeling techniques: mathematical descriptions involving simplifying assumptions and avoiding detail. The descriptions use fundamental mathematical concepts such as set and function. | 2 |
| SE | Tools and Environments | Software configuration management and version control; release management<br>Requirements analysis and design modeling tools<br>Programming environments that automate parts of program construction processes | 3 |
| SE | Software Design | The use of components in design: component selection, design, adaptation and assembly of components, components and patterns, components and objects, (for example, build a GUI using a standard widget set). | 4 |

# CS371: Computer Graphics

## Williams College

**Dr. Morgan McGuire**

**http://www.cs.williams.edu/cs371.html**

**(description below based on the Fall 2010 & 2012 offerings)**

**Knowledge Areas with topics and learning outcomes covered in the course:**

| Knowledge Area | Total Hours of Coverage |
|---|---|
| Graphics and Visualization (GV) | 19 |
| Software Engineering (SE) | 7 |
| Architecture and Organization (AR) | 4 |
| Software Development Fundamentals (SDF) | 2 |

**Where does the course fit in your curriculum?**

This is an elective course taken by students mostly in their third year, following at least CS1, CS2, and a computer organization course. Courses are on a semester system: 12 weeks long with 3 weekly 1-hour lectures and a weekly four-hour laboratory session with the instructor. This course covers fundamental graphics techniques for the computational synthesis of digital images from 3D scenes. In the computer science major, this course fulfills the project course requirement and the quantitative reasoning requirement.

**What is covered in the course?**

- Computer graphics and its place in computer science
- Surface modeling
- Light modeling
- The Rendering Equation
- Ray casting
- Surface scattering (BSDFs)
- Spatial data structures
- Photon mapping
- Refraction
- Texture Mapping
- Transformations
- Rasterization
- The graphics pipeline
- GPU architecture
- Film production and effects
- Deferred shading
- Collision detection
- Shadow maps

**What is the format of the course?**

PhotoShop, medical MRIs, video games, and movie special effects all programmatically create and manipulate digital images. This course teaches the fundamental techniques behind these applications. We begin by building a mathematical model of the interaction of light with surfaces, lenses, and an imager. We then study the data structures and processor architectures that allow us to efficiently evaluate that physical model. Students will complete a series of programming assignments for both photorealistic image creation and real-time 3D rendering using C++, OpenGL, and GLSL as well as tools like SVN and debuggers and profilers. These assignments

cumulate in a multi-week final project. Topics covered in the course include: projective geometry, ray tracing, bidirectional surface scattering functions, binary space partition trees, matting and compositing, shadow maps, cache management, and parallel processing on GPUs.
The cumulative laboratory exercises bring students through the entire software research and development pipeline: domain-expert feature set, formal specification, mathematical and computational solutions, team software implementation, testing, documentation, and presentation.

### How are students assessed?

In 13 weeks, students complete 9 programming projects, two of which are multi-week team projects.

### Course textbooks and materials

Students use the iOS app *The Graphics Codex* as their primary textbook and individually choose one of the following for assigned supplemental readings, based on their interest: Fundamentals of Computer Graphics, 3rd Edition, A K Peters; Computer Graphics: Principles and Practice, 3rd Edition, Addison Wesley; or Real-Time Rendering, 3rd Edition, A K Peters.

### Why do you teach the course this way?

In this course, students work from first principles of physics and mathematics, and a body of knowledge from art. That is, I seek to lead with science and then support it with engineering. Many other CS courses--such as networking, data structures, architecture, compilers, and operating systems--develop the ability to solve problems that arise within computer science and computers themselves. In contrast, graphics is about working with problems that arise in other disciplines, specifically physics and art. The challenge here is not just solving a computer science problem but also framing the problem in computer science terms in the first place. This is a critical step of the computational and scientific approach to thinking, and the field of graphics presents a natural opportunity to revisit it in depth for upper-level students. Graphics in this case is a motivator, but the skills are intentionally presented as ones that can be applied to other disciplines, for example, biology, medicine, geoscience, nuclear engineering, and finance. That the rise of GPU computing in HPC is a great example of numerical methods and engineering originating in computer graphics being generalized in just this way.

**Body of Knowledge coverage**

| KA | Knowledge Unit | Topics Covered | Hours |
|---|---|---|---|
| GV | Fundamental Concepts | Basics of Human visual perception (HCI Foundations).<br>Image representations, vector vs. raster, color models, meshes.<br>Forward and backward rendering (i.e., ray-casting and rasterization).<br>Applications of computer graphics: including game engines, cad, visualization, virtual reality.<br>Polygonal representation.<br>Basic radiometry, similar triangles, and projection model.<br>Use of standard graphics APIs (see HCI GUI construction).<br>Compressed image representation and the relationship to information theory.<br>Immediate and retained mode.<br>Double buffering. | 3 |
| GV | Basic Rendering | Rendering in nature, i.e., the emission and scattering of light and its relation to numerical integration.<br>Affine and coordinate system transformations.<br>Ray tracing.<br>Visibility and occlusion, including solutions to this problem such as depth buffering, Painter's algorithm, and ray tracing.<br>The forward and backward rendering equation.<br>Simple triangle rasterization. | 10 |

| | | | |
|---|---|---|---|
| | | Rendering with a shader-based API.<br>Texture mapping, including minification and magnification (e.g., trilinear MIP-mapping).<br>Application of spatial data structures to rendering.<br>Sampling and anti-aliasing.<br>Scene graphs and the graphics pipeline. | |
| GV | Geometric Modeling | Basic geometric operations such as intersection calculation and proximity tests<br>Parametric polynomial curves and surfaces.<br>Implicit representation of curves and surfaces.<br>Approximation techniques such as polynomial curves, Bezier curves, spline curves and surfaces, and non-uniform rational basis (NURB) spines, and level set method.<br>Surface representation techniques including tessellation, mesh representation, mesh fairing, and mesh generation techniques such as Delaunay triangulation, marching cubes. | 6 |
| SDF | Development Methods | Program correctness<br>  The concept of a specification<br>  Unit testing<br>Modern programming environments, Programming using library components and their APIs<br>Debugging strategies<br>Documentation and program style | 2 |
| AR | Performance enhancements | Superscalar architecture<br>Branch prediction, Speculative execution, Out-of-order execution<br>Prefetching<br>Vector processors and GPUs<br>Hardware support for Multithreading<br>Scalability | 3 |
| CN | Modeling and Simulation | Formal models and modeling techniques:  mathematical descriptions involving simplifying assumptions and avoiding detail. The descriptions use fundamental mathematical concepts such as set and function. | 2 |
| SE | Tools and Environments | Software configuration management and version control; release management<br>Requirements analysis and design modeling tools<br>Programming environments that automate parts of program construction processes | 3 |
| SE | Software Design | The use of components in design:  component selection, design, adaptation and assembly of components, components and patterns, components and objects, (for example, build a GUI using a standard widget set). | 4 |

**Other comments**

http://graphics.cs.williams.edu/courses/cs371/f12/files/welcome.pdf is a carefully-crafted introduction to computer graphics and this specific style of course that may be useful to other instructors.

# CS453: Introduction to Compilers

**Colorado State University, Fort Collins, CO**
**Michelle Strout**
**mstrout@cs.colostate.edu**
**http://www.cs.colostate.edu/~cs453**

**Knowledge Areas that contain topics and learning outcomes covered in the course**

| Knowledge Area | Total Hours of Coverage |
|---|---|
| Programming Languages (PL) | 38 |
| Software Engineering (SE) | 8 |
| Algorithms and Complexity (AL) | 4 |

**Where does the course fit in your curriculum?**

This course is an elective for senior undergraduates and first year graduate students offered once a year in the spring semester. Typically about 20-25 students take this course. The prerequisite for this course is a required third year Software Development Methods course.

**What is covered in the course?**

CS 453 teaches students how to implement compilers. Although most computer science professionals do not end up implementing a full compiler, alumni of this course are surprised by how often the skills they learn are used within industry and academic settings. The subject of compilers ties together many concepts in computer science: the theoretical concepts of regular expressions and context free grammars; the systems concept of layers including programming languages, compilers, system calls, assembly language, and architecture; the embedded systems concept of an architecture with restricted resources; and the software engineering concepts of revision control, debugging, testing, and the visitor design pattern. Students write a compiler for a subset of Java called MeggyJava. We compile MeggyJava to the assembly language for the ATmega328p microcontroller in the Meggy Jr RGB devices.

**Course topics:**

- Regular and context free languages including DFAs and NFAs.
- Scanning and parsing
  - Finite state machines and push down automata
  - FIRST and FOLLOW sets
  - Top-down predictive parsing
  - LR parse table generation
- Meggy Jr Simple runtime library
- AVR assembly code including the stack and heap memory model
- Abstract syntax trees
- Visitor design pattern
- Semantic analysis including type checking
- Code generation for method calls and objects
- Data-flow analysis usage in register allocation
- Iterative compiler design and development
- Test-driven development and regression testing
- Revision control and pair programming

**What is the format of the course?**

Colorado State University uses a semester system: this course is 15 weeks long with 2 one and a half hour of lectures per week and 1 weekly recitation section (4 total contact hours / week, for approximately 60 total hours not counting the final exam). There is a 16th week for final exams. In the past this course has been only on campus, but starting in Spring 2013 we are providing it as a blended on campus and online course.

**How are students assessed?**

There are 7 programming assignments and 4 written homeworks, which together constitute 50% of the course grade. The programming assignments lead the students through the iterative development of a compiler written in Java that translates a subset of Java to AVR assembly code. The AVR assembly code is then assembled and linked with the avr-gcc tool chain to run on Meggy Jr game devices. The process is iterative in that the first programming assignment that starts building the compiler results in a compiler that can generate the AVR code for the setPixel() call; therefore students can write MeggyJava programs that draw 8x8 pictures on their devices. Later assignments incrementally add features to the MeggyJava language and data structures such as abstract syntax trees to the compiler. We also have a simulator available at (http://code.google.com/p/mjsim/) to enable debugging of the generated AVR code and for grading purposes. Students start doing their programming assignments individually, but are then encouraged to work as programming pairs. We expect students to spend approximately 8-12 hours each week outside of the classroom on the course.

**Course textbooks and materials**

Lecture notes written by the instructor and materials available online largely replace a textbook, though for additional resources, we recommend Modern Compiler Implementation in Java (Second Edition) by Andrew Appel, Cambridge, 2002. Additionally we provide the students with a link and reading assignments for a free online book "Basics in Compiler Design" by Torben Mogensen. The lecture notes are available at the webpage provided above.

**Why do you teach the course this way?**

We view the compiler course as bringing together many theoretical and practical skills from previous courses and enabling the students to use these skills within the context of a full semester project. The key elements of this course are the approach to iterative compiler development, the emphasis on many software development tools such as debuggers, revision control, etc., and mapping a high level programming language, Java, to a popular assembly language for embedded systems. All of these elements are new editions to the compiler course in our department and have been incorporated into the course since 2007. In general the move to targeting an active assembly language AVR that operates on a game device Meggy Jr has been more popular with the students than the previous targets of C and then MIPS.

This course is an elective and students do consider it to be challenging. Many students discuss the compiler developed in this course with potential employers. Additionally graduates report that understanding how a language maps to assembly helps their debugging skills after starting positions in industry after their degree.

**Body of Knowledge coverage**

| KA | Knowledge Unit | Topics Covered | Hours |
|---|---|---|---|
| AL | Basic Automata Computability and Complexity | Finite state machines, regular expressions, and context free grammars | 2 |
| AL | Advanced Automata Computability and Complexity | NFAs, DFAs, their equivalence, and push-down automata. | 2 |
| PL | Event Driven and Reactive Programming | Programming the Meggy Jr game device. | 3 |

| | | | |
|---|---|---|---|
| PL | Program Representation | All | 1 |
| PL | Language Translation and Execution | All except tail calls, closures, and garbage collection | 6 |
| PL | Syntax Analysis | All | 8 |
| PL | Compiler Semantic Analysis | All | 5 |
| PL | Code Generation | All | 6 |
| PL | Runtime Systems | All | 4 |
| PL | Static Analysis | Data-flow analysis for register allocation | 3 |
| PL | Language Pragmatics | All except lazy versus eager | 2 |
| SE | Software Verification and Validation | Test-driven development and regression testing | 4 |
| SE | Software Design | Use of the visitor design pattern | 2 |
| SE | Software Processes | All Core 1 and Core 2 topics | 2 |

**Additional topics**

- Iterative compiler development instead of the monolithic phase based approach (lexer, parser, type checker, code generator).
- Revision control

**Other comments: None**

# Introduction to Computer Science


## Harvey Mudd College, Claremont, CA 91711

**Zachary Dodds**

**dodds@cs.hmc.edu**


**https://www.cs.hmc.edu/twiki/bin/view/CS5**

**Knowledge Areas that contain topics and learning outcomes covered in the course**

| Knowledge Area | Total Hours of Coverage |
|---|---|
| Software Development Fundamentals (SDF) | 24 |
| Algorithms and Complexity (AL) | 9 |
| Architecture and Organization (AR) | 7.5 |
| Programming Languages (PL) | 3 |
| Parallel and Distributed Computing (PD) | 1.5 |

### Where does the course fit in your curriculum?

Every first-semester at Harvey Mudd College – and about 100 students from sister institutions at the Claremont Colleges – take one of the sections of this course. It has no prerequisites and is offered in three distinct "colors": CS 5 "gold" is for students with no prior experience, CS 5 "black" is for students with some experience, and CS 5 "green" is a version with a biological context to all of the computational content. 275 students were in CS 5 in fall 2012.

### What is covered in the course?

This course has five distinct modules of roughly three weeks each:
(1) We begin with conditionals and recursion, practicing a functional problem-solving approach to a variety of homework problems. Python is the language in which students solve all of their assignments in this module.
(2) In the second module students investigate the fundamental ideas of binary representation, combinational circuits, machine architecture, and assembly language; they complete assignments in each of these topics using Python, Logisim, and a custom-built assembly language named Hmmm. This unit culminates with the hand-implementation of a recursive function in assembly, pulling back the curtain on the "magic" that recursion can sometimes seem.
(3) Students return to Python in the third module, building imperative/iterative idioms and skills that build from the previous unit's assembly language jumps. Creating the Mandelbrot set from scratch, Markov text-generation, and John Conway's Game of Life are part of this module's student work.
(4) The fourth module introduces object-oriented skills, again in Python, with students implementing a Date calculator, a Board class that can host a game of Connect Four, and an Player class that implements game-tree search.
(5) The fifth module introduces mathematical and theoretical facets of computer science, including finite-state machines, Turing machines, and uncomputable functions such as Kolmogorov complexity and the halting problem. Small assignments use JFLAP to complement this in-class content, even as students' work centers on a medium-sized Python final project, such as a genetic algorithm, a game using 3d graphics with the VPython library, or a text-analysis web application.

### What is the format of the course?

This is a three-credit course with two 75-minute lectures per week. An optional, but incentivized lab attracts 90+% of the students to a two-hour supplemental session each week.

### How are students assessed?

Students complete an assignment each week of 2-5 programming or other computational problems. Survey from the past five years show that the average workload has been consistent at about 4 hours/week outside of structured time, though the distribution does range from one hour to over 12. In addition, there is one in-class midterm exam and an in-class final exam.

### Course textbooks and materials

The course has a textbook that its instructors wrote for it: CS for Scientists and Engineers by its instructors, C. Alvarado, Z. Dodds, R. Libeskind-Hadas, and G. Kuenning. Beyond that, we use Python, Logisim, JFLAP, and a few other supplemental materials.

### Why do you teach the course this way?

Our CS department redesigned its introductory CS offering in 2006 to better highlight the breadth and richness of CS over the previous introductory offering. In addition, the department's redesign sought to encourage more women to pursue CS beyond this required experience. A SIGCSE '08 [1] publication, reported the initial curricular changes and their results, including a significant and sustained increase in the number of women CS majors. Subsequent publications at SIGCSE, ITiCSE, and Inroads [2,3,4,5] flesh out additional context for this effort and several longer-term assessments of the resulting changes.

### References:

[1] Dodds, Z., Libeskind-Hadas, R., Alvarado, C., and Kuenning, G. Evaluating a Breadth-First CS 1 for Scientists. SIGCSE '08.
[2] Alvarado, C., and Dodds, Z. Women in CS: An Evaluation of Three Promising Practices. SIGCSE '10.
[3] Dodds, Z., Libeskind-Hadas, R., and Bush, E. When CS1 is Biology1: Crossdisciplinary collaboration as CS context. ITiCSE '10.
[4] Dodds, Z., Libeskind-Hadas, R., and Bush, E. Bio1 as CS1: evaluating a crossdisciplinary CS context. ITiCSE '12.
[5] Alvarado, C., Dodds, Z., and Libeskind-Hadas, R. Broadening Participation in Computing at Harvey Mudd College. ACM Inroads, Dec. 2012.

**Body of Knowledge coverage**

| KA | Knowledge Unit | Topics Covered | Hours |
|---|---|---|---|
| AL | AL/Algorithmic Strategies | Brute-force, especially as expressed recursively | 3 |
| AL | AL/Basic Automata, Computability and Complexity | Precisely those, plus Kolmogorov Complexity | 6 |
| AR | AR/Digital logic and digital systems | Combinational logic design, as well as building flip-flops and memory from them | 3 |
| AR | AR/Machine level representation of data | Binary, two's complement, other bases | 1.5 |
| AR | AR/Assembly level machine organization | Assembly constructs, von Neumann architecture, the use of the stack to support function calls (and recursion in particular) | 3 |
| PD | PD/Parallelism Fundamentals | Parallelism vs. concurrency, simultaneous computation, measuring wall-clock speedup | 1.5 |
| PL | PL/Object-Oriented Programming | Definition of classes: fields, methods, and constructors; object-oriented design | 3 |

| | | | |
|---|---|---|---|
| SDF | SDF/Algorithms and Design | The concept and properties of algorithms; abstraction; program decomposition | 6 |
| SDF | SDF/Fundamental Programming Concepts | Basic syntax and semantics of a higher-level language; Variables and primitive data types; Expressions and assignments; functions and recursive functions | 12 |
| SDF | SDF/Fundamental Data Structures | Arrays/Linked lists; Strings; Maps (dictionaries) | 6 |

# CSC 131: Principles of Programming Languages

## Pomona College, Claremont, CA 91711

**Kim B. Bruce**

**kim@cs.pomona.edu**

**http://www.cs.pomona.edu/~kim/CSC131F12/**

**Knowledge Areas that contain topics and learning outcomes covered in the course**

| Knowledge Area | Total Hours of Coverage |
|---|---|
| Programming Languages (PL) | 38 |
| Parallel & Distributed Computing (PD) | 4 |

### Where does the course fit in your curriculum?

This course is generally taken by juniors and seniors, and has a prerequisite of Data Structures (CSC 062) and Computability and Logic (CSC 081). It is required for all CS majors.

### What is covered in the course?

A thorough examination of issues and features in programming language design and implementation, including language-provided data structuring and data-typing, modularity, scoping, inheritance, and concurrency. Compilation and run-time issues. Introduction to formal semantics. Specific topics include:

- Overview of compilers and Interpreters (including lexing & parsing)
- Lambda calculus
- Functional languages (via Haskell)
- Formal semantics (mainly operational semantics)
- Writing interpreters based on formal semantics
- Static and dynamic type-checking
- Run-time memory management
- Data abstraction & modules
- Object-oriented languages (illustrated via Java and Scala)
- Shared memory parallelism/concurrency (semaphores, monitors, locks, etc.)
- Distributed parallelism/concurrency via message-passing (Concurrent ML, Scala Actors)

### What is the format of the course?

The course meets face-to-face in lecture format for 3 hours per week for 14 weeks.

### How are students assessed?

There are weekly homework assignments as well as take-home midterm and final exams. Students are expected to spend 6 to 8 hours per week outside of class on course material. Problem sets include both paper-and-pencil as well as programming tasks.

### Course textbooks and materials

The course text is "Concepts in Programming Languages", by John Mitchell, supplemented by various readings. Lecture notes are posted, as are links to supplementary material on languages and relevant topics.

### Why do you teach the course this way?

This course combines two ways of teaching a programming languages course. On the one hand it provides an overview of the design space of programming languages, focusing on features and evaluating them in terms of how they impact programmers. On the other hand, there is an important stream focusing on the implementation of programming languages. A major theme of this part of the course is seeing how formal specifications of a language (grammar, type-checking rules, and formal semantics) lead to an implementation of an interpreter for the language. Thus the grammar leads to a recursive descent compiler, type-checking rules lead to the implementation of a type-checker, and the formal semantics leads to the interpretation of abstract syntax trees.

The two weeks on parallelism/concurrency support in programming languages reflects my belief that students need to understand how these features work and the variety of ways of supporting parallelism concurrency – especially as we don't know what approach will be the most successful in the future.

From the syllabus: "Every student passing this course should be able to:

- Quickly learn programming languages and how to apply them to effectively solve programming problems.
- Rigorously specify, analyze, and reason about the behavior of a software system using a formally defined model of the system's behavior.
- Realize a precisely specified model by correctly implementing it as a program, set of program components, or a programming language."

**Body of Knowledge coverage**

| **KA** | **Knowledge Unit** | **Topics Covered** | **Hours** |
|---|---|---|---|
| PL | Object-Oriented Programming | All (with assumed knowledge of OOP from CS1 and CS2 in Java) | 5 |
| PL | Functional Programming | All (building on material from earlier course covering functional programming in ML) | 5 |
| PL | Event-Driven & Reactive Programming | Previously covered in CS 1 & CS 2 | 0 |
| PL | Basic Type Systems | All | 5 |
| PL | Program Representation | All | 2 |
| PL | Language Translation and Execution | All | 3 |
| PL | Syntax Analysis | Lexing & top-down parsing (regular expressions and cfg's covered in prerequisite) | 2 |
| PL | Compiler Semantic Analysis | AST's, scope, type-checking & type inference | 1 |
| PL | Advanced Programming Constructs | Lazy evaluation & infinite streams<br>Control abstractions: Exception Handling, continuations, monads<br>OO abstraction: multiple inheritance, mixins, Traits, multimethods<br>Module systems<br>Language support for checking assertions, invariants, and pre-post-conditions | 3 |
| PL | Concurrency and Parallelism | Constructs for thread-shared variables and shared-memory synchronization<br>Actor models<br>Language support for data parallelism | 2 |

| | | | |
|---|---|---|---|
| PL | Type Systems | Compositional type constructors<br>Type checking<br>Type inference<br>Static overloading | 2 |
| PL | Formal Semantics | Syntax vs. semantics<br>Lambda Calculus<br>Approaches to semantics: Operational, Denotational, Axiomatic<br>Formal definitions for type systems | 6 |
| PL | Language Pragmatics | Principles of language design such as orthogonality<br>Evaluation order<br>Eager vs. delayed evaluation | 2 |
| PD | Parallelism Fundamentals | Multiple simultaneous computations<br>Goals of parallelism vs. concurrency<br>Programming constructs for creating parallelism, communicating, and coordinating<br>Programming errors not found in sequential programming | 2 |
| PD | Parallel Decomposition | Need for Communication & Coordination<br>Task-based decomposition: threads<br>Data-parallel decomposition: SIMD, MapReduce, Actors | 1 |
| PD | Communication & Coordination | Shared memory<br>Consistency in shared memory<br>Message passing<br>Atomicity: semaphores & monitors<br>Synchronization | 1 |

# Csc 453: Translators and Systems Software


## The University of Arizona, Tucson, AZ

**Saumya Debray**

**debray@cs.arizona.edu**


**http://www.cs.arizona.edu/~debray/Teaching/CSc453/**

**Knowledge Areas that contain topics and learning outcomes covered in the course**

| Knowledge Area | Total Hours of Coverage |
|---|---|
| Programming Languages (PL) | 35 |

### Where does the course fit in your curriculum?

This course is an upper-division elective aimed at third-year and fourth-year students. It is one of a set of courses in the "Computer Systems" elective area, from which students are required to take (at least) one. Pre-requisites are: a third-year course in C and Unix; and a third-year course on data structures; a third-year survey course on programming languages is recommended but not required. Enrolment is typically around 40.

### What is covered in the course?

This course covers the design and implementation of translator-oriented systems software, focusing specifically on compilers, with some time spent on related topics such as interpreters and linkers.

### Course topics:

- Background. Compilers as translators. Other examples of translators: document-processing tools such as ps2pdf and latex2html; web browsers; graph-drawing tools such as dot; source-to-source translators such as f2c; etc.
- Lexical analysis. Regular expressions; finite-state automata and their implementation. Scanner-generators: flex.
- Parsing. Context-free grammars. Top-down and bottom-up parsing. SLR(1) parsers. Parser-generators: yacc, bison.
- Semantic analysis. Attributes, symbol tables, type checking.
- Run-time environments. Memory organization. Stack-based environments.
- Intermediate representations. Abstract syntax trees, three-address code. Code generation for various language constructs. Survey of machine-independent code optimization.
- Interpreters. Dispatch mechanisms: byte-code, direct-threading, indirect-threading. Expression evaluation: Registers vs. operand stack. Just-in-time compilers. Examples: JVM vs. Dalvik for Java; Spidermonkey for JavaScript; JIT compilation in the context of web browsers.
- Linking. The linking process, linkers and loaders. Dynamic linking.

### What is the format of the course?

The University of Arizona uses a semester system. The course lasts 15 weeks and consists of 2.5 hours per week of face-to-face lectures together with a 50-minute discussion class (about 3.5 contact hours per week, for a total of about 52 hours not counting exams). The lectures focus on conceptual topics while the discussion section focuses on the specifics of the programming project.

### How are students assessed?

The course has a large programming project (50% of the final score), one or two midterm exams (20%-25% of the final score), and an optional final exam (25%-30% of the final score).

The programming project involves writing a compiler for a significant subset of C. The front end is generated using flex and yacc/bison; the back end produces MIPS assembly code, which is executed on the SPIM simulator.

The project consists of a sequence of four programming assignments with each assignment builds on those before it:

1. html2txt or latex2html: a simple translator from a subset of HTML to ordinary text (html2txt) or from a subset of LaTeX to HTML (latex2html). Objective: learn how to use flex and yacc/bison.
2. Scanner and parser. Use flex and yacc/bison to generate a front-end for a subset of C.
3. Type-checking.
4. Code generation.

Each assignment is about 2-3 weeks long, with a week between assignments for students to fix bugs before the next assignment. Students are expected to work individually, and typically spend about 15-20 hours per week on the project.

### Course textbooks and materials

Lecture notes written by the instructor largely replace a textbook. The book Introduction to Compiler Design, by T. Mogensen, is suggested as an optional text (a free version is available online as Basics of Compiler Design).

### Why do you teach the course this way?

The class lectures have the dual purpose of focusing on compiler-specific topics in depth but also discussing the variety and scope of these translation problems and presenting compilation as an instance of a broader class of translation problems. Translation problems mapping one kind of structured representation to another arise in a lot of areas of computing: compilers are one example of this, but there are many others, including web browsers (Firefox, Chromium), graph drawing tools (dot, VCG), and document formatting tools (LaTeX), to name a few. Understanding the underlying similarities between such tools can be helpful to students in recognizing other examples of such translation problems and providing guidance on how their design and implementation might be approached. It also has the effect of making compiler design concepts relevant to other aspects of their computer science education. I find it helpful to revisit the conceptual analogies between compilers and other translation tools (see above) repeatedly through the course, as different compiler topics are introduced.

The programming project is aimed at giving students a thorough hands-on understanding of the nitty-gritty details of implementing a compiler for a simple language.

**Body of Knowledge coverage**

| KA | Knowledge Unit | Topics Covered | Hours |
|---|---|---|---|
| PL | Program Representation | All | 4 |
| PL | Language Translation and Execution | All | 5 |
| PL | Syntax Analysis | All | 8 |
| PL | Compiler Semantic Analysis | All | 8 |
| PL | Code Generation | All | 5 |
| PL | Runtime Systems | The following topics are covered: Target-platform characteristics such as registers, instructions, bytecodes; Data layout for activation records; Just-in-time compilation. | 5 |

### Additional topics

- Structure and content of object files
- static vs. dynamic linking
- Position-independent code
- Dynamic linking

## An Overview of the Mulit-paradigm Three-course CS Introduction at Grinnell College

Consistent with both CS 2008 and CS 2013, the CS program at Grinnell College follows a multi-paradigm approach in its introductory curriculum. Since each course emphasizes problem solving following a specified paradigm, students gain practice by tackling a range of problems. Toward that end, the first two courses utilize application themes to generate interesting problems and generate interest in interdisciplinary connections of computer science. The following list outlines some main elements of this approach:

CSC 151: Functional Problem Solving (CS1)
- Primary Paradigm: Functional problem solving
- Supporting language: Scheme
- Application area: image processing, media scripting

CSC 161: Imperative Problem Solving and Data Structures (CS2)
- Primary Paradigm: Imperative problem solving
- Supporting language: C (no objects as in C++)
- Application area: robotics

CSC 207: Algorithms and Object-Oriented Design (CS3)
- Primary Paradigm: Object-oriented problem solving
- Supporting language: Java
- Application areas: several large-scale problems

***Audience:*** As with many liberal arts colleges, incoming students at Grinnell have not declared a major. Rather students devote their first and/or second year to exploring their possible interests, taking a range of introductory courses, and developing a general background in multiple disciplines. Often students in introductory computer science are taking courses because of some interest in the subject or because they think some contact with computing might be important for their future activities within a technological society. An important role of the introductory courses, therefore, is to generate interest in the discipline. Although some students enter Grinnell having already decided to major in computer science, over half of the CS graduates each year had not initially considered CS as a likely major. Rather, they became interested by the introductory courses and want to explore the discipline further with more courses.

***Pedagogy:*** Each class session of each course meets in a lab, and each course utilizes many collaborative lab exercises throughout a semester. Generally, CSC 151 schedules a lab almost every day, with some introductory comments at the start of class. CSC 161 is composed of about eight 1.5-2 week modules, in which each module starts with a lecture/demonstration, followed by 3-5 labs, and concluded by a project. CSC 207 contains about the same number of class days devoted to lecture as to lab work. Throughout, students work in pairs on labs, and the pairs typically are changed each week. Students work individually on tests and on some additional homework (usually programming assignments).

***Spiral Approach for Topic Coverage:*** The multi-paradigm approach allows coverage of central topics to be addressed incrementally from multiple perspectives. One course may provide some introductory background on a subject, and a later course in the sequence may push the discussion further from the perspective of a different problem-solving paradigm. For example, encapsulation of data and operations arises naturally as higher-order procedures within functional problem solving and later as classes and objects within object-oriented problem solving.

One way to document this spiral approach tracks time spent on various Knowledge Areas through the three-course sequence:

| Knowledge Area | CSC 151 Functional Problem-solving | CSC 161 Imperative Problem-solving and Data Structures | CSC 207 Algorithms and Object-Oriented Design | Total Grinnell Intro-CS Hours |
|---|---|---|---|---|
| Algorithms and Complexity (AL) | 6 | 1 | 14 | 21 |
| Architecture and Organization (AR) | | 3 | | 3 |
| Computational Science (CN) | | 1 | 1 | 2 |
| Graphics and Visual Computing (GV) | 2 | | | 2 |
| Human-Computer Interaction (HCI) | 4 | | | 4 |
| Security and Information Assurance (IAS) | | 1 | | 1 |
| Intelligent Systems (IS) | | 1 | | 1 |
| Programming Languages (PL) | 13 | 9 | 12 | 34 |
| Software Development Fundamentals (SDF) | 20 | 27 | 16 | 63 |
| Software Engineering (SE) | 3 | 3 | 4 | 10 |
| Social and Professional Issues (SP) | | 2 | 1 | 3 |

# CSC 151: Functional problem solving

## Grinnell College, Grinnell, Iowa USA

**Henry M. Walker**
**walker@cs.grinnell.edu**

**http://www.cs.grinnell.edu/~davisjan/csc/151/2012S/**

**Knowledge Areas that contain topics and learning outcomes covered in the course**

| Knowledge Area | Total Hours of Coverage |
|---|---|
| Software Development Fundamentals (SDF) | 20 |
| Programming Languages (PL) | 13 |
| Algorithms and Complexity (AL) | 6 |
| Human-Computer Interaction (HCI) | 4 |
| Software Engineering (SE) | 3 |
| Graphics and Visual Computing (GV) | 2 |

### Brief description of the course's format and place in the undergraduate curriculum

This course is the first in Grinnell's three-course, multi-paradigm introduction to computer science. As the first regular course, it has no prerequisites. Many students are in their first or second year, exploring what the discipline might involve. Other students (ranging from first-years to graduating seniors) take the course as part of gaining a broad, liberal arts education; these students may be curious about computing, but they think their major interests are elsewhere. (Note, however, that the course recruits a number of these students as majors or concentrators.) Each class session meets in the lab. A class might begin with some remarks or class discussion, but most classes involve students working collaboratively in pairs on problems and laboratory exercises.

### Course description and goals

This course introduces the discipline of computer science by focusing on functional problem solving with media computation as an integrating theme. In particular, the course explores mechanisms for representing, making, and manipulating images. The course considers a variety of models of images based on pixels, basic shapes, and objects that draw.

The major objectives for this course include:

- Understanding some fundamentals of computer science: algorithms, data structures, and abstraction.
- Experience with the practice of computer programming (design, documentation, development, testing, and debugging) in a high-level language, Scheme.
- Learning problem solving from a functional programming perspective, including the use of recursion and higher-order procedures.
- Sharpening general problem solving, teamwork, and study skills.

### Course topics

1. Fundamentals of functional problem-solving using a high-level functional language
    1. abstraction
    2. modularity
    3. recursion, including helper procedures
    4. higher-order procedures
    5. analyzing of algorithms
2. Language elements
    1. symbols
    2. data types
    3. conditionals
    4. procedures and parameters
    5. local procedures
    6. scope and binding

3. Data types and structures
    1. primitive types
    2. lists
    3. pairs, pair structures, and association lists
    4. trees
    5. raster graphics and RGB colors
    6. objects in Scheme
4. Algorithms
    1. searching
    2. sorting
    3. transforming colors and images
5. Software development
    1. design
    2. documentation
    3. development
    4. testing, including unit testing
    5. debugging

## Course textbooks, materials, and assignments

This course relies upon an extensive collection of on-line materials (readings, labs); there is no published textbook. As with most courses offered by Grinnell's CS Department, this course has a detailed, on-line, day-by-day schedule of topics, readings, labs and assignments. This daily schedule contains a link to all relevant pages, handouts, labs, references, and materials.

**Body of Knowledge coverage**

| KA | Knowledge Unit | Topics Covered | Hours |
|---|---|---|---|
| AL | Fundamental Data Structures and Algorithms | Max/min, search, 2 sorts | 6 |
| AL | Algorithmic Strategies | Some divide-and-conquer mentioned | 0 |
| GV | Fundamental Concepts | Elements of graphics introduced as part of image-processing application theme | 2 |
| HCI | Designing Interaction | Overview covers much of Knowledge Unit | 4 |
| PL | Object-oriented Programming | Objects as higher-order procedures; much additional material in later courses | 2 |
| PL | Functional Problem-solving | Knowledge Unit covered in full | 7 |
| PL | Type Systems | Types, primitive types, compound list type, dynamic types, binding | 1 |
| PL | Language Translation and Execution | Program interpretation, encapsulation, basic algorithms to avoid mutable state in context of functional language | 3 |
| SDF | Algorithms and Design | Coverage of recursion-based topics | 8 |
| SDF | Fundamental Programming Concepts | Topics related to recursion and functional problem-solving covered thoroughly | 9 |
| SDF | Fundamental Structures | Lists and arrays (vectors) covered, some elements of string processing | 3 |
| SE | Software Verification and Validation | Specifications, pre- and post-conditions, unit testing | 3 |

# CSC 161: Imperative Problem Solving and Data Structures

## Grinnell College, Grinnell, Iowa USA

**Henry M. Walker**
**walker@cs.grinnell.edu**

**http://www.cs.grinnell.edu/~walker/courses/161.sp12/**

**Knowledge Areas that contain topics and learning outcomes covered in the course**

| Knowledge Area | Total Hours of Coverage |
|---|---|
| Software Development Fundamentals (SDF) | 27 |
| Programming Languages (PL) | 9 |
| Architecture and Organization (AR) | 3 |
| Software Engineering (SE) | 3 |
| Social and Professional Issues (SP) | 2 |
| Algorithms and Complexity (AL) | 1 |
| Computational Science (CN) | 1 |
| Security and Information Assurance (IAS) | 1 |
| Intelligent Systems (IS) | 1 |

### Brief description of the course's format and place in the undergraduate curriculum

This course is the second in Grinnell's three-course, multi-paradigm introductory CS sequence. Many students are first- or second-year students who are exploring computer science as a possible major (after becoming interested in the subject from the first course). Other students may enroll in the course to broaden their liberal arts background or to learn elements of imperative problems solving and C to support other work in the sciences or engineering. As with Grinnell's other introductory courses, each class session meets in a lab, and work flows seamlessly between lecture and lab-based activities. Work generally is divided into eight 1.5-2 week modules. Each module begins with a lecture giving an overview of topics and demonstrating examples. Students then work collaborative in pairs on 3-5 labs, and the pairs change for each module. Each module concludes with a project that integrates topics and applies the ideas to an interesting problem.

### Course description and goals

This course utilizes robotics as an application domain in studying imperative problem solving, data representation, and memory management. Additional topics include assertions and invariants, data abstraction, linked data structures, an introduction to the GNU/Linux operating system, and programming the low-level, imperative language C.

### Course topics

This course explores elements of computing that have reasonably close ties to the architecture of computers, compilers, and operating systems. The course takes an imperative view of problem solving, supported by programming in the C programming language. Some topics include:

- *imperative problem solving:* top-down design, common algorithms, assertions, invariants
- *C programming:* syntax and semantics, control structures, functions, parameters, macro processing, compiling, linking, program organization
- *concepts with data:* data abstraction, integer and floating-point representation, string representation, arrays, unions, structures, linked list data structures, stacks, and queues
- *machine-level issues:* data representation, pointers, memory management
- *GNU/Linux operating system:* commands, bash scripts, software development tools

**Body of Knowledge coverage**

| KA | Knowledge Unit | Topics Covered | Hours |
|---|---|---|---|
| AL | Fundamental Data Structures and Algorithms | Simple numerical algorithms, searching and sorting within an imperative context | 1 |
| AR | Machine level representation of data | Knowledge Unit covered in full | 3 |
| CN | Fundamentals | Several examples, problems, and assignments introduce modeling and simulation; Math/Stat Dept. offers a full course in Modeling | 1 |
| IAS | Fundamental Concepts | Issues of bounds checking, buffer overflow, impact introduced; other topics mentioned in later courses | 1 |
| IS | Robotics | Introduction to problems, progress, control, sensors, inherent uncertainty | 1 |
| PL | Type Systems | Static types, primitive and compound times, references, error detection of type errors | 2 |
| PL | Program Representation | Use of interpreters, compilers, type-checkers, division of programs over multiple files; higher-level materials in later Programming Language Concepts course | 2 |
| PL | Language Translation and Execution | Compilation, interpretation vs. compilation, language translation basics, run-time representation, run-time memory layout, manual memory management | 5 |
| SDF | Algorithms and Design | Coverage of iteration-based topics | 0 |
| SDF | Fundamental Programming Concepts | Topics related to iteration and imperative problem-solving covered thoroughly | 10 |
| SDF | Fundamental Structures | Low-level discussion of arrays, records, structs, strings, stacks, queues, linked structures | 14 |
| SDF | Development Methods | Program correctness, pre- and post-conditions, testing, debugging, libraries | 3 |
| SE | Tools and Environments | Testing tools, automated builds; additional material in later Software Design course | 1 |
| SE | Software Design | System design principles, component structure and behavior | 1 |
| SE | Software Verification and Validation | Test plans, black-box, white-box testing | 1 |
| SP | Social Context | Some discussion of social implications (e.g,. of robots) | 1 |
| SP | Professional Communication | Group communications, introduction to writing of technical documents | 1 |

# CSC 207: Algorithms and Object-Oriented Design
# Grinnell College, Grinnell, Iowa USA

**Henry M. Walker**
**walker@cs.grinnell.edu**

**http://www.cs.grinnell.edu/~walker/courses/207.sp12/**

**Knowledge Areas that contain topics and learning outcomes covered in the course**

| Knowledge Area | Total Hours of Coverage |
|---|---|
| Software Development Fundamentals (SDF) | 16 |
| Algorithms and Complexity (AL) | 14 |
| Programming Languages (PL) | 12 |
| Software Engineering (SE) | 3 |
| Computational Science (CN) | 1 |
| System Fundamentals (SF) | 1 |
| Social and Professional Issues (SP) | 1 |

## Brief description of the course's format and place in the undergraduate curriculum

This course is the third in Grinnell's three-course, multi-paradigm introductory CS sequence. Many students are second-year students, but the course also attracts third-year and fourth-year students who are majoring in other fields but also want to expand their background on topics they found interesting in the first two courses. As with Grinnell's other introductory courses, each class session meets in a lab, and work flows seamlessly between lecture and lab-based activities. The course includes a significant emphasis on collaboration in pairs during 23 in-class labs. Additional programming assignments and tests are done individually. All course materials, including readings and all labs, are available freely over the World Wide Web.

## Course description and goals

CSC 207, Algorithms and Object-Oriented Design, explores object-oriented problem solving using the Java programming language. Topics covered include principles of object-oriented design and problem solving, abstract data types and encapsulation, data structures, algorithms, algorithmic analysis, elements of Java programming, and an integrated development environment (IDE) (e.g., Eclipse).

## Course topics

Topics and themes covered include:

- Principles of object-oriented design and problem solving
  - Objects and classes
  - Encapsulation, abstraction, and information hiding
  - Inheritance
  - Polymorphism
  - Unit testing
  - Integration testing
- Abstract data types, data structures, and algorithms
  - Dictionaries
  - Hash tables
  - Binary search trees
  - Priority queues
  - Heaps
- Algorithmic analysis
  - Upper-bound efficiency analysis; Big-O Notation

    - Comparison of results for small and large data sets
    - Introduction of tight-bound analysis (Big-θ)
- Elements of Java programming
    - Basic syntax and semantics
    - Interfaces and classes
    - Exceptions
    - Strings
    - Arrays, ArrayLists, vectors
    - Comparators; sorting
    - Generics
    - Java type system
    - Iterators
    - Introduction to the Java class library
- An integrated development environment (IDE) (e.g., Eclipse)

**Course textbooks, materials, and assignments**

The main textbook is Mark Allen Weiss, *Data Structures and Problem Solving Using Java,* Fourth Edition, Addison-Wesley, 2009. ISBN: 0-321-54040-5. This is supplemented by numerous on-line readings. In-class work involves an equal mix of lecture and lab-based activities. Students work collaboratively in pairs on the 23 required labs. Students also work individually on several programming assignments and on tests.

**Body of Knowledge coverage**

| KA | Knowledge Unit | Topics Covered | Hours |
|---|---|---|---|
| *AL* | Basic Analysis | Knowledge Unit covered in full; additional material in later Analysis of Algorithms course | 5 |
| AL | Algorithmic Strategies | Divide and conquer ; much more in later Analysis of Algorithms course | 1 |
| AL | Fundamental Data Structures and Algorithms | Additional sorting, trees, analysis of algorithms; searching and sorting done in earlier course; graphs in later Analysis of Algorithms course | 8 |
| CN | Fundamentals | Several examples, problems, and assignments utilize modeling and simulation; Math/Stat Dept. offers a full course in Modeling | 1 |
| PL | Object-Oriented Programming | Knowledge Unit covered in full | 10 |
| PL | Type Systems | Supplement to discussion in earlier courses to cover Knowledge Unit in full; additional material in later Programming Language Concepts course | 2 |
| PL | Language Translation and Execution | Automatic vs. manual memory management, garbage collection | 0 |
| SDF | Algorithms and Design | Comparison of iterative and recursive strategies | 3 |
| SDF | Fundamental Programming Concepts | Simple I/O, iteration over structures (e.g., arrays) | 2 |
| SDF | Fundamental Structures | Stacks, queues, priority queues, sets, references and aliasing, choosing data structures covered in object-oriented context | 5 |
| SDF | Development Methods | Program correctness, defensive programming, exceptions, code reviews, test-case generation, pre- and post-conditions, modern programming environments, library APIs, debugging, documentation, program style | 6 |
| SE | Software Design | Design principles, refactoring | 1 |
| SE | Software Construction | Exception handling, coding standards | 1 |
| SE | Software Verification and Validation | Test case generation, approaches to testing | 1 |
| SF | Resource Allocation and Scheduling | Example/assignment introduce kinds of scheduling: FIFO, priority; much additional material in Operating Systems course | 1 |
| SP | Professional Ethics | Codes of ethics, accountability, responsibility | 1 |

# An Overview of the Two-Course Intro Sequence

# Creighton University, Omaha, NE

The Computer Science & Informatics program at Creighton University serves students with a wide range of interests. These include traditional computer science students who plan for careers in software development or graduate studies, as well as students whose interests overlap with business analytics, graphics design, and even journalism. All majors in the department take a foundational sequence in information, consisting of introductory informatics, professional writing, Web design, and CS0. The computer science major begins with a two-course introductory programming sequence, which covers almost all of the Software Development Fundamentals (SDF) Knowledge Area, along with Knowledge Units from Programming Languages (PL), Algorithms and Complexity (AL), Software Engineering (SE), and others. The two introductory programming courses are:

CSC 221: Introduction to Programming

- Language: Python
- Focus: Fundamental programming concepts/techniques, writing small scripts

CSC 222: Object-Oriented Programming

- Language: Java
- Focus: object-oriented design, designing and implementing medium-sized projects

It should be noted that in the course exemplars for these two courses, there is significant overlap in SDF Topics Covered. Many of the software development topics are introduced in the first course (in Python, following a procedural approach), then revisited in the second course (in Java, following an object-oriented approach).

# CSC 221: Introduction to Programming
# Creighton University, Omaha, Nebraska, USA

**David Reed**
**davereed@creighton.edu**

**http://dave-reed.com/csc221.F12**

**Knowledge Areas that contain topics and learning outcomes covered in the course**

| Knowledge Area | Total Hours of Coverage |
|---|---|
| Software Development Fundamentals (SDF) | 30 |
| Programming Languages (PL) | 5 |
| Algorithms and Complexity (AL) | 4 |
| Social and Professional Practice (SP) | 1 |

### Where does the course fit in your curriculum?

This is the first course in the introductory programming sequence. It is required for all computer science majors. Many students will have already taken or will concurrently take CSC 121, Computers and Scientific Thinking, which is a requirement of the computer science major (but not an explicit prerequisite for this course). CSC 121 is a balanced CS0 course that provides some experience with programming (developing interactive Web pages using JavaScript and HTML) while also exploring a breadth of computer science topics (e.g., computer organization, history of computing, workings of Internet & Web, algorithms, digital representation, societal impact). This course is offered every semester, with an enrollment of 20-25 students per course.

### What is covered in the course?

This course provides an introduction to problem solving and programming using the Python scripting language. The specific goals of this course are:

- To develop problem solving and programming skills to enable the student to design solutions to non-trivial problems and implement those solutions in Python.
- To master the fundamental programming constructs of Python, including variables, expressions, functions, control structures, and lists.
- To build a foundation for more advanced programming techniques, including object-oriented design and the use of standard data structures (as taught in CSC 222).

### What is the format of the course?

The course meets twice a week for two hours (although it only counts as three credit hours). The course is taught in a computer lab and integrates lectures with lab time.

### How are students assessed?

Students complete 6-8 assignments, which involve the design and implementation of a Python program and may also include a written component in which the behavior of the program is analyzed. There are "random" daily quizzes to provide student feedback (quizzes are handed out but administered with a 50% likelihood each day). There are two 75-minute tests and a cumulative 100-minute final exam.

### Course textbooks and materials

Free Online Texts:

*Scratch for Budding Computer Scientists*, David J. Malan.

*Learning with Python: Interactive Edition*, Brad Miller and David Ranum (based on material by Jeffrey Elkner, Allen B. Downey, and Chris Meyers).

Optional Text:

*Python Programming: An Introduction to Computer Science (2nd ed.)*, John Zelle.

### Why do you teach the course this way?

This course was revised in 2011 to use Python. Previously, it was taught in Java using an object-oriented approach. It was felt that the overhead of the language was too much for beginners, and the object-orientated approach was not ideal for the range of students taking this course (which include business, graphic design, and journalism majors). A scripting language, such as Python, allowed for a stronger problem-solving focus and prepared non-majors to take high-demand courses such as Web Programming and Mobile Development.

**Body of Knowledge coverage**

| KA | Knowledge Unit | Topics Covered | Hours |
|---|---|---|---|
| SDF | Algorithms and Design | concept & properties of algorithms; role of algorithms in problem solving; problem solving strategies (iteration, divide & conquer); implementation of algorithms; design concepts & principles (abstraction, decomposition) | 8 |
| SDF | Fundamental Programming Concepts | syntax & semantics; variables & primitives, expressions & assignments; simple I/O; conditionals & iteration; functions & parameters | 8 |
| SDF | Fundamental Data Structures | arrays; records; strings; strategies for choosing the appropriate data structure | 8 |
| SDF | Development Methods | program correctness (specification, defensive programming, testing fundamentals, pre/postconditions); modern environments; debugging strategies; documentation & program style | 6 |
| PL | Object-Oriented Programming | object-oriented design; classes & objects; fields & methods | 3 |
| PL | Basic Type Systems | primitive types; type safety & errors | 1 |
| PL | Language Translation | interpretation; translation pipeline | 1 |
| AL | Fundamental Data Structures and Algorithms | simple numerical algorithms; sequential search; simple string processing | 4 |
| SP | History | history of computer hardware; pioneers of computing; history of Internet | 1 |

# CSC 222: Object-Oriented Programming

## Creighton University, Omaha, Nebraska, USA

**David Reed**

**davereed@creighton.edu**

**http://dave-reed.com/csc222.S13**

**Knowledge Areas that contain topics and learning outcomes covered in the course**

| **Knowledge Area** | **Total Hours of Coverage** |
|---|---|
| Software Development Fundamentals (SDF) | 19 |
| Programming Languages (PL) | 11 |
| Algorithms and Complexity (AL) | 7 |
| Software Engineering (SE) | 3 |

### Where does the course fit in your curriculum?

This is the second course in the introductory programming sequence, following CSC 221 (Introduction to Programming). Students must have completed CSC 221 or otherwise demonstrate competence in some programming language. It is offered every spring, with an enrollment of 20-25 students.

### What is covered in the course?

Building upon basic programming skills in Python from CSC 221, this course focuses on the design and analysis of larger, more complex programs using the industry-leading language, Java. The specific goals of this course are:

- To know and use basic Java programming constructs for object-oriented problem solving (e.g., classes, polymorphism, inheritance, interfaces)
- To appreciate the role of algorithms and data structures in problem solving and software design (e.g., objected-oriented design, lists, files, searching and sorting)
- To be able to design and implement a Java program to model a real-world system, and subsequently analyze its behavior.
- To develop programming skills that can serve as a foundation for further study in computer science.

### What is the format of the course?

The course meets twice a week for two hours (although it only counts as three credit hours). The course is taught in a computer lab and integrates lectures with lab time.

### How are students assessed?

Students complete 5-7 assignments, which involve the design and implementation of a Python program and may also include a written component in which the behavior of the program is analyzed. There are "random" daily quizzes to provide student feedback (quizzes are handed out but administered with a 50% likelihood each day). There are two 75-minute tests and a cumulative 100-minute final exam.

### Course textbooks and materials

*Objects First with Java, 5th edition*, David Barnes and Michael Kolling.

### Why do you teach the course this way?

This course was revised in 2011. Previously, it was part of a two-course intro sequence in Java that integrated programming fundamentals, problem solving, and object-oriented design. The new division, in which basic programming (scripting) is covered in the first course and object-oriented design is covered in this second, is proving much more successful.

**Body of Knowledge coverage**

| KA | Knowledge Unit | Topics Covered | Hours |
|---|---|---|---|
| SDF | Algorithms and Design | all topics (including recursion, encapsulation, information hiding) | 4 |
| SDF | Fundamental Programming Concepts | all topics (including recursion) | 4 |
| SDF | Fundamental Data Structures | all topics, with the possible exception of priority queues, sets and maps (which are covered in the subsequent Data Structures course) | 5 |
| SDF | Development Methods | all topics | 6 |
| PL | Object-Oriented Programming | all topics | 9 |
| PL | Basic Type Systems | reference types; generic types | 2 |
| AL | Basic Analysis | best/average/worst case behavior; asymptotic analysis; Big O; empirical measurement | 3 |
| AL | Fundamental Data Structures and Algorithms | sequential and binary search; $O(N^2)$ sorts, O(N log N) sorts | 4 |
| SE | Software Design | design principles; structure & behavior | 1 |
| SE | Software Construction | defensive coding; exception handling | 1 |
| SE | Software Verification and Validation | verification & validation; testing fundamentals (unit testing, test plan creation) | 1 |

# CSCI 0190: Accelerated Introduction to Computer Science

**Brown University, Providence, RI, USA**
**Shriram Krishnamurthi**
**sk@cs.brown.edu**

**http://www.cs.brown.edu/courses/csci0190/**

**Knowledge Areas that contain topics and learning outcomes covered in the course**

| Knowledge Area | Total Hours of Coverage |
|---|---|
| Programming Languages | 13 |
| Software Development Fundamentals | 10 |
| Software Engineering | 6 |
| Algorithms and Complexity | 5 |
| Parallel and Distributed Computing | 1 |

### Where does the course fit in your curriculum?

Brown has three introductory computing sequences as routes into the curriculum. The other two are spread over a whole year, and cover roughly a semester of content in programming and a semester of algorithms and data structures. This course, which is one of these, compresses most of this material into a single semester.

Students elect into this course, either through high school achievement or performance in the early part of one of the other sequences.

Approximately 30 students it every year, compared to 300-400 in the other two sequences.

### What is covered in the course?

The course is a compressed introduction into programming along with basic algorithms and data structures. It interleaves these two. The data structures cover lists, trees, queues, heaps, DAGs, and graphs; the algorithms go up through classic ones such as graph shortest paths and minimum spanning trees. The programming is done entirely with pure functions. It begins with graphical animations (such as simple video games), then higher-order functional programming, and encodings of laziness.

### What is the format of the course?

Classroom time is a combination of lecture and discussion. We learn by exploration of mistakes.

### How are students assessed?

There are about ten programming assignments. Students spend over 10 and up to 20 hours per week on the course.

### Course textbooks and materials

There is no textbook. Students are given notes and code from class.

All programming is done in variations of the Racket programming language using the DrRacket programming environment.

### Why do you teach the course this way?

The material of the course is somewhat constrained by the department's curricular needs. However, the arrangement represents my desires.

In interleaving programming and algorithmic content to demonstrate connections between them, I am especially interested in ways in which programming techniques enhance algorithmic ones: for instance, the use of memoization to alter a computation's big-*O* performance (and the trade-offs in program structure relative to dynamic programming).

The course is revised every year, based on the previous year's outcome.

Students consider the course to be extremely challenging, and of course recommend it only for advanced students.

**Body of Knowledge coverage**

| KA | Knowledge Unit | Topics Covered | Hours |
|---|---|---|---|
| AL | Advanced Data Structures, Algorithms, and Analysis | Balanced trees, graphs, string-based data structures, amortized analysis | 5 |
| PD | Parallelism Fundamentals | Parallel data structures, map-reduce | 1 |
| PL | Object-Oriented Programming | All | 3 |
| PL | Functional Programming | All | 6 |
| PL | Event-Driven and Reactive Programming | All | 2 |
| PL | Basic Type Systems | All | 2 |
| SDF | All | All | 10 |
| SE | Software Design | Design recipe | 3 |
| SE | Software Verification Validation | Test suites, testing oracles, test-first development | 3 |

# CSCI 140: Algorithms

## Pomona College, Claremont, CA 91711, USA

**Tzu-Yi Chen**
**tzuyi@cs.pomona.edu**

**http://www.cs.pomona.edu/~tzuyi/Classes/CC2013/Algorithms/index.html**

**Knowledge Areas that contain topics and learning outcomes covered in the course**

| Knowledge Area | Total Hours of Coverage |
|---|---|
| AL/Algorithms and Complexity | 29-32 |
| SDF/Software Development Fundamentals | 1.5 |
| PD/Parallel and Distributed Computing | 0 - 3 |

### Where does the course fit in your curriculum?

This is a required course in the CS major that is typically taken by juniors and seniors. The official prerequisites are Data Structures (CSCI 062, the 3rd course in the introductory sequence) and Discrete Math (CSCI 055). However, we regularly make exceptions for students who have had only the first 2 courses in the introductory sequence as long as they have also taken a proof-based math class such as Real Analysis or Combinatorics. Algorithms is not a prerequisite for any other required classes, but is a prerequisite for electives such as Applied Algorithms.

### What is covered in the course?

This class covers basic techniques used to analyze problems and algorithms (including asymptotics, upper/lower bounds, best/average/worst case analysis, amortized analysis, complexity), basic techniques used to design algorithms (including divide & conquer / greedy / dynamic programming / heuristics, choosing appropriate data structures), and important classical algorithms (including sorting, string, matrix, and graph algorithms). The goal is for students to be able to apply all of the above to

### What is the format of the course?

This is a one semester (14 week) face-to-face class with 2.5 hours of lecture a week.

### How are students assessed?

There is a written assignment (written up individually) due almost every class as well as 1 or 2 programming assignments (done in groups of 1-3) due during the semester; solutions are evaluated on clarity, correctness, and (when appropriate) efficiency. Students are expected to spend 6-10 hours a week outside of class on course material. There are also 1 or 2 midterms and a final exam. Students are expected to attend lectures and to demonstrate engagement either by asking/answering questions in class or by going to office hours (the professor's or the TAs').

### Course textbooks and materials

The textbook is *Introduction to Algorithms, 3rd Edition* by Cormen, Leiserson, Rivest, and Stein. For the programming assignments students are strongly encouraged to use their choice of C, C++, Java, or Python, though other languages may be used with permission. Students are required to use LaTeX to format their first 2-3 weeks of assignments, after which its use is encouraged but not required.

### Why do you teach the course this way?

This course serves as a bridge between theory and practice. Lectures cover classical algorithms and techniques for reasoning about their correctness and efficiency. Assignments allow students to practice skills necessary for developing, describing, and justifying algorithmic solutions for new problems. The 1 or 2 programming assignments go a step further by also requiring an implementation; these assignments help students better appreciate both what it means to describe an algorithm clearly and what issues can remain in implementation. To encourage students not to fall behind in the material, two problem sets are due every week (one every lecture). By the end of the semester students should also have a strong appreciation for the role of algorithms.

**Body of Knowledge coverage**

| KA | Knowledge Unit | Topics Covered | Hours |
|---|---|---|---|
| SDF | algorithms and design | concept and properties of algorithms, role of algorithms, problem-solving strategies, separation of behavior and implementation | 1.5 |
| AL | basic analysis | all core-tier1<br>all core-tier2 | 1<br>1.5 |
| AL | algorithmic strategies | core-tier1: brute-force, greedy, divide-and-conquer, dynamic programming<br>core-tier2: heuristics | 6<br>.5 |
| AL | fundamental data structures and algorithms | all core-tier1<br>core-tier2: heaps, graph algorithms (minimum spanning tree, single source shortest path, all pairs shortest path), string algorithms (longest common subsequence) | 3<br>6 |
| AL | basic automata computability and complexity | no core-tier1 (covered in other required courses),<br>core-tier2: introduction to P/NP/NPC with examples | 0<br>1 |
| AL | advanced computational complexity | P/NP/NPC, Cook-Levin, classic NPC problems, reductions | 2 |
| AL | advanced automata theory and computability | none (covered in other required courses) | 0 |
| AL | advanced data structures algorithms and analysis | balanced trees (1-2 examples), graphs (topological sort, strongly connected components), advanced data structures (disjoint sets, mergeable heaps), network flows, linear programming (formulating, duality, overview of techniques), approximation algorithms (2-approx for metric-TSP, vertex cover), amortized analysis | 8 |

## Additional topics

The above table covers approximately 30 hours of lecture and gives the material that is covered every semester. The remaining hours can be used for review sessions, to otherwise allow extra time for topics that students that semester find particularly confusing, for in-class midterms, or to cover a range of additional topics. In the past these additional topics have included:

| KA | Knowledge Unit | Topics Covered |
|---|---|---|
| AL | advanced computational complexity | P-space, EXP |
| AL | advanced data structures algorithms and | more approximation algorithms (e.g. Christofides, subset- |

| | | |
|---|---|---|
| | analysis | sum), geometric algorithms, randomized algorithms, online algorithms and competitive analysis, more data structures (e.g. log* analysis for disjoint-sets) |
| PD | parallel algorithms, analysis, and programming | critical path, work and span, naturally parallel algorithms, specific algorithms (e.g. mergesort, parallel prefix) |
| PD | formal models and semantics | PRAM |

**Other comments**

Starting in the Fall of 2013 approximately 2-3 hours of the currently optional material on parallel algorithms will become a standard part of the class.

# CSCI 1730: Introduction to Programming Languages

## Brown University, Providence, RI, USA

**Shriram Krishnamurthi**
**sk@cs.brown.edu**

**http://www.cs.brown.edu/courses/csci1730/**

**Knowledge Areas that contain topics and learning outcomes covered in the course**

| Knowledge Area | Total Hours of Coverage |
|---|---|
| *Programming Languages (PL)* | *35* |

### Where does the course fit in your curriculum?

The course is designed for third- and fourth-year undergraduates and for PhD students who are either early in their study or outside this research area. Over the years I have shrunk the prerequisites, so it requires only the first year introductory computer science sequence, discrete math, and some theory. The course is not required. However, it consistently has one of the highest enrollments in the department.

### What is covered in the course?

The course uses definitional interpreters and related techniques to teach the core of several programming languages.

The course begins with a quick tour of writing definitional interpreters by covering substitution, environments, and higher-order functions. The course then dives into several topics in depth:

- Mutation
- Recursion and cycles
- Objects
- Memory management
- Control operators
- Types
- Contracts
- Alternate evaluation models

### What is the format of the course?

Classroom time is a combination of lecture and discussion. We learn by exploration of mistakes.

### How are students assessed?

There are about ten programming assignments and three written homeworks. The written ones are open-ended and ask students to explore deign alternatives. Advanced students are given a small number of classic papers to read. Students spend over 10 and up to 20 hours per week on the course.

### Course textbooks and materials

The course uses *Programming Languages: Application and Interpretation* by Shriram Krishnamurthi. All programming is done in variations of the Racket programming language using the DrRacket programming environment. Some versions of the course task students with writing programs in a variety of other languages such as Haskell and Prolog.

**Why do you teach the course this way?**

My primary goal in the design of this course is to teach “the other 90%”: the population of students who will not go on to become programming language researchers. My goal is to infect them with linguistic thinking: to understand that by embodying properties and invariants in their design, languages can solve problems.

Furthermore most of them, as working developers, will inevitably build languages of their own; I warn them about classic design mistakes and hope they will learn enough to not make ones of their own.
The course is revised with virtually every offering. Each time we pick one module and try to innovate in the presentation or learning materials.

Independent student feedback suggests the course is one of the most challenging in the department. Nevertheless, it does not prevent high enrollments, since students seem to appreciate the linguistic mindset it engenders.

**Body of Knowledge coverage**

| KA | Knowledge Unit | Topics Covered | Hours |
|---|---|---|---|
| PL | Object-Oriented Programming | Object representations and encodings, types | 3 |
| PL | Functional Programming | All | 3 |
| PL | Basic Type Systems | All | 9 |
| PL | Language Translation and Execution | Interpretation, representations | 3 |
| PL | Runtime Systems | Value layout, garbage collection, manual memory management | 3 |
| PL | Advanced Programming Constructs | Almost all (varies by year) | 6 |
| PL | Type Systems | All | 3 |
| PL | Language Pragmatics | Almost all (varies by year) | 3 |
| PL | Logic Programming | Relationship to unification and continuations | 2 |

# Algorithm Design and Analysis


## Williams College, Williamstown, MA

**Brent Heeringa**

**heeringa@cs.williams.edu**


**www.cs.williams.edu/~heeringa/classes/cs256**

**Knowledge Areas that contain topics and learning outcomes covered in the course**

| Knowledge Area | Total Hours of Coverage |
| --- | --- |
| AL/Algorithms and Complexity | *28* |
| DS/Discrete Structures | 2 |

### Where does the course fit in your curriculum?

Students commonly take this course in their second year. It is required. The course has two prerequisites: Discrete Mathematics (taken in the Department of Mathematics) and Data Structures. Over the past five years, the course averages 20 students per year. In 2013 there are 38 students enrolled.

### What is covered in the course?

Analysis: asymptotic analysis including lower bounds on sorting, recurrence relations and their solutions.

Graphs: directed, undirected, planar, and bipartite.

Greedy Algorithms: shortest paths, minimum spanning trees, and the union-find data structure (including amortized analysis).

Divide and Conquer Algorithms: integer and matrix multiplication, the fast-fourier transform.

Dynamic Programming: matrix parenthesization, subset sum, RNA secondary structure, DP on trees.

Network Flow: Max-Flow, Min-Cut (equivalence, duality, algorithms).

Randomization: randomized quicksort, median, min-cut, universal hashing, skip lists.

String Algorithms: string matching, suffix trees and suffix arrays.

Complexity Theory: Complexity classes, reductions, and approximation algorithms.

### What is the format of the course?

The course format is face-to-face lecture. The lectures last 50 minutes and happen 3 times a week for 12 weeks for a total of 30 contact hours. Office hours often increase contact hours significantly. There is no lab or discussion section.

### How are students assessed?

Nine problem sets, each worth 5% of the total grade. I drop the lowest score. One take-home midterm exam worth 25% of the grade. One take-home final exam worth 25% of the grade. 6 pop quizzes, each worth 1% of the grade. I drop the lowest score. A class participation grade based on attendance, promptness, and participation worth 5% of the grade. I expect students will spend 7-10 hours on the problem sets and exams.

**Course textbooks and materials**

*Algorithm Design* by Kleinberg and Tardos, supplemented liberally with my own lecture notes. There are two programming assignments, one in Python and one in Java.

**Why do you teach the course this way?**

The goal of *Algorithms* is for students to learn and practice a variety of problem solving strategies and analysis techniques. Students develop algorithmic maturity. They learn how to think about problems, their solutions, and the quality of those solutions.

I have taught Algorithms since 2007 except for 2010 when I was on sabbatical. My sense is that my course is non-trivial revision of the offering pre-2007. Students consider the course challenging in a rewarding way.

**Body of Knowledge coverage**

| KA | Knowledge Unit | Topics Covered | Hours |
|---|---|---|---|
| *AL* | Basic Analysis | Asymptotic analysis including definitions of asymptotic upper, lower, and tight bounds. Discussion of worst, best, and expected running time (Monte Carlo and Las Vegas for randomized algorithms and a brief discussion about making assumptions about the distribution of the input). Natural complexity classes in P (log n, linear quadratic, etc.), recurrence relations and their solutions (mostly via the recursion tree and master method although we mention generating functions as a more general solution). | *5* |
| AL | Algorithmic Strategies | Brute-force algorithms (i.e., try-'em-all), divide and conquer, greedy algorithms, dynamic programming, and transformations. We do not cover recursive backtracking, branch and bound, or heuristic programming although these topics are given some attention in Artificial Intelligence. | 6 |
| AL | Fundamental Data Structures and Algorithms | Order statistics including deterministic median, We do not cover heaps directly in this course although we mention various heap implementations and their trade-offs (e.g., Fibonacci heaps, soft heaps, etc.) when discussing shortest path and spanning tree algorithms. Undirected and directed graphs, bipartite graphs, graph representations and trade-offs, fundamental graph algorithms including BFS and DFS, shortest-path algorithms, and spanning tree algorithms. Many of the topic areas included in this knowledge unit are covered in Data Structures so we review them quickly and use them as a launching point for more advanced material. | 4 |
| AL | Basic Automata, Computability and Complexity | Algorithm Design and Analysis contains very little complexity theory–these topics are all covered in detail in our Theory of Computation course. However, we do spend 1 lecture on the complexity classes P and NP, and approaches to dealing with intractability including approximation algorithms (mentioned below). | 1 |
| AL | Advanced Data Structures, Algorithms and Analysis | A quick review of ordered dynamic dictionaries (including balanced BSTs like Red-Black Trees and AVL-Trees) as a way of motivating Skip Lists. Graph algorithms to find a maximum matching and connected components. Some advanced data structures like union-find (including the log*n amortized analysis). Suffix trees, suffix arrays (we follow the approach of Karkkainen and Sanders that recursively builds a suffix array and then transforms it into a suffix tree). Network flow including max-flow, min-cut, bipartite matching | 12 |

| | | | |
|---|---|---|---|
| | | and other applications including Baseball Elimination. Randomized algorithms including randomized median, randomized min-cut, randomized quicksort, and Rabin-Karp string matching. We cover the geometric problem of finding the closest pair of points in the plane and develop the standard randomized solution based on hashing. Sometimes we cover linear programming. Very little number theoretic and geometric algorithms are covered due to time. We spend two lectures on approximation algorithms because it is my research area. | |
| DS | Discrete Probability | We review concepts from discrete probability in support of randomized algorithms. This includes expectation, variance, and (very quickly) concentration bounds (we use these to prove that many of our algorithms run in their expected time with very high probability) | 2 |

**Other comments**

There is some overlap with the topics covered here and DS / Graphs and Trees.

# CSCI 334: Principles of Programming Languages

**Williams College, Williamstown, MA**
**Stephen N. Freund**
*freund@cs.williams.edu*

**http://www.cs.williams.edu/~freund/cs334-exemplar/**

**Knowledge Areas that contain topics and learning outcomes covered in the course**

| Knowledge Area | Total Hours of Coverage |
|---|---|
| Programming Languages (PL) | 31 |
| Parallel & Distributed Computing (PD) | 5 |
| Information Assurance and Security (IAS) | 1 |

**Where does the course fit in your curriculum?**

This course is designed for any students who have successfully completed CS1 and CS2. It is required for the Computer Science major.

**What is covered in the course?**

Specific topics covered in this course include:

- Functional programming concepts in Lisp
- Syntax, semantics, and evaluation strategies
- ML programming, including basic types, datatypes, pattern matching, recursion, and higher order functions
- Types, dynamic/static type checking, type inference, parametric polymorphism
- Run-time implementations: stacks, heaps, closures, garbage collection
- Exception handlers
- Abstract types and modularity
- Object-oriented programming and systems design
- Object-oriented language features: objects, dynamic dispatch, inheritance, subtyping, etc.
- Multiple inheritance vs. interfaces vs. traits
- Scala programming, including most basic language features.
- Language-based security mechanisms and sandboxing
- Models of concurrency: shared memory and actors

**What is the format of the course?**

Semesters are twelve weeks long. This course meets twice per week for 75 minutes, with most of that time being spent as a lecture, discussing primary literature, or working on interactive programming tasks. (Total lecture hours: 30)

**How are students assessed?**

Students are assessed via weekly problem sets, a midterm, and a final. The problem sets include pencil-and-paper exercises, as well as programming problems in various languages.

**Course textbooks and materials**

The primary textbook is “Concepts in Programming Languages”, by John Mitchell. This is augmented with PowerPoint slides and web-based materials on additional topics, as well as some primary literature on the design goals and histories of various programming languages.

**Why do you teach the course this way?**

This course presents a comprehensive introduction to the principle features and overall design of programming languages. The material should enable successful students to

- recognize how a language's underlying computation model can impact how one writes programs in that language;
- quickly learn new programming languages, and how to apply them to effectively solve programming problems;
- understand how programming language features are implemented;
- reason about the tradeoffs among different languages; and
- use a variety of programming languages with some proficiency. These currently include Lisp, ML, Java, C++, and Scala.

It is also designed to force students to think about expressing algorithms in programming languages beyond C, Java and similar languages, since those are the languages most students have been previously exposed to in our CS1, CS2, and systems courses.

**Body of Knowledge coverage**

| KA | Knowledge Unit | Topics Covered | Hours |
|---|---|---|---|
| PL | Object-Oriented Programming | All (with assumed knowledge of OOP from CS1 and CS2 in Java) | 4 |
| PL | Functional Programming | All | 6 |
| PL | Basic Type Systems | All | 5 |
| PL | Program Representation | All | 1 |
| PL | Language Translation and Execution | All | 3 |
| PL | Advanced Programming Constructs | Lazy evaluation<br>Exception Handling<br>Multiple inheritance, Mixins, Traits<br>Dynamic code evaluation ("eval") | 3 |
| PL | Concurrency and Parallelism | Constructs for thread-shared variables and shared-memory synchronization<br>Actor models<br>Language support for data parallelism | 3 |
| PL | Type Systems | Type inference<br>Static overloading | 2 |
| PL | Formal Semantics | Syntax vs. semantics<br>Lambda Calculus | 2 |
| PL | Language Pragmatics | Principles of language design such as orthogonality<br>Evaluation order<br>Eager vs. delayed evaluation | 2 |
| IAS | Secure Software Design and Engineering | Secure Design Principles and Patterns (Saltzer and Schroeder, etc)<br>Secure Coding techniques to minimize vulnerabilities in code<br>Secure Testing is the process of testing that security requirements are met (including Static and Dynamic analysis). | 1 |

| | | | |
|---|---|---|---|
| PD | Parallelism Fundamentals | All | 2 |
| PD | Parallel Decomposition | Need for communication and coordination/synchronization<br>Task-base decomposition<br>Data-parallel decomposition<br>Actors | 2 |
| PD | Communication & Coordination | Shared Memory<br>Message Passing<br>Atomicity<br>Mutual Exclusion | 1 |

# CSCI 432 Operating Systems

## Williams College, Williamstown, MA

**Jeannie Albrecht**

**jeannie@cs.williams.edu**

**http://www.cs.williams.edu/~jeannie/cs432/index.html**

**Knowledge Areas that contain topics and learning outcomes covered in the course**

| **Knowledge Area** | **Total Hours of Coverage** |
|---|---|
| Operating Systems (OS) | 30 |
| Systems Fundamentals (SF) | (Overlap with OS hours) |
| Networking and Communications | 2 |
| Parallel and Distributed Computing | (Overlap with OS hours) |

### Where does the course fit in your curriculum?

Operating Systems is typically taken by juniors and seniors. It is not compulsory, although it does satisfy our “project course” requirement, and students are required to take at least one project course to complete the major. The prerequisites are Computer Organization and either Algorithms or Programming Languages. The latter requirement is mostly used to ensure a certain level of maturity rather than specific skills or knowledge. Average class size is 15-20 students.

### What is covered in the course?

This course explores the design and implementation of computer operating systems. Topics include historical aspects of operating systems development, systems programming, process scheduling, synchronization of concurrent processes, virtual machines, memory management and virtual memory, I/O and file systems, system security, OS/architecture interaction, and distributed operating systems. The concepts in this course are not limited to any particular operating system or hardware platform. We discuss examples that are drawn from historically significant and modern operating systems including Unix, Windows, Mach, and the various generations of Mac OS.

The objective of this course is threefold: to demystify the interactions between the software written in other courses and hardware, to familiarize students with the issues involved in the design and implementation of modern operating systems, and to explain the more general systems principles that are used in the design of all computer systems.

### What is the format of the course?

The course format is primarily lectures with some interactive discussion. There are no officially scheduled lab sessions, but a few lectures are held in the lab to help with project setup.

### How are students assessed?

Student evaluation is largely based on 4 implementation projects that include significant programming, as well as 2-3 written homework assignments, 6-8 research paper evaluations, and a midterm exam. Projects typically span 2-3 weeks and require 20-30 hours of work each. Written homework assignments and paper evaluations require 3-10 hours of work each.

### Course textbooks and materials

Textbook: *Modern Operating Systems,* 3rd ed., by Andrew Tanenbaum
Other assigned reading material: 6-8 research papers
Programming Project One: Inverted Index in C++ (warmup project)

Programming Project Two: Threads and Monitors in C++
Programming Project Three: Virtual Memory Manager in C++
Programming Project Four: Smash the Stack in C++

### Why do you teach the course this way?

The course combines classical OS concepts with more modern technologies. The combination of textbook readings as well as select research papers gives students a breadth of knowledge about current and recent OS topics. The programming projects are challenging, but most students are able to successfully finish all of them in the allotted time.

**Body of Knowledge coverage**

| KA | Knowledge Unit | Topics Covered | Hours |
|---|---|---|---|
| OS | Overview of Operating Systems | Role and purpose of OS, key design issues, influences of security and networking | 2 |
| OS | Operating System Principles | Structuring methods, abstractions, processes, interrupts, dual-mode (kernel vs. user mode) operation | 3 |
| OS/PD/SF | Concurrency (OS)/Communication and Coordination (PD)/Parallelism (SF) | Structures, atomic access to OS objects, synchronization primitives, spin-locks | 5 |
| OS/PD/SF | Scheduling and Dispatch (OS)/System Performance Evaluation (OS)/Parallel Performance (PD)/Resource Allocation and Scheduling (SF) | Preemptive and non-preemptive scheduling, evaluating scheduling policies, processes and threads | 4 |
| OS | Memory Management | Virtual memory, paging, caching, thrashing | 4 |
| OS | Security and Protection | Access control, buffer overflow exploits, OS mechanisms for providing security and controlling access to resources | 4 |
| OS/SF | Virtual Machines (OS)/Cross-Layer Communications (SF)/Virtualization and Isolation (SF) | Types of virtualization, design of different hypervisors, virtualization trade-offs | 3 |
| OS | Device Management | Briefly discuss device drivers, briefly discuss mechanisms used in interfacing a range of devices | 1 |
| OS | File Systems | File system design and implementation, files, directories, naming, partitioning | 3 |
| OS | Fault Tolerance | Discuss and define relevance of fault tolerance, reliability, and availability in OS design | 1 |
| NC | Reliable Data Delivery | OS role in reliable data delivery | 0.5 |
| NC | Networked Applications/Introduction | Role of OS in network naming schemes, role of layering | 1 |

| | | | |
|---|---|---|---|
| NC | Routing and Forwarding | Role of OS in routing and forwarding | 0.5 |
| OS | Real Time and Embedded Systems | | 0 |

**Other comments**

I have often contemplated replacing Project Four with one that focused on File Systems rather than Security. However, the students really enjoy the Stack Smashing project, and we do not offer another course that focuses on Security in our curriculum.

# CSCI 434T: Compiler Design


## Williams College, Williamstown, MA

**Stephen N. Freund**

*freund@cs.williams.edu*


**http://www.cs.williams.edu/~freund/cs434-exemplar/**

**Knowledge Areas that contain topics and learning outcomes covered in the course**

| **Knowledge Area** | **Total Hours of Coverage** |
|---|---|
| Programming Languages (PL) | 33 |
| Architecture and Organization (AR) | 3 |
| Software Design (SE) | 2 |

**Where does the course fit in your curriculum?**

Compiler Design is a senior-level course designed for advanced undergraduates who have already taken courses on computer organization, algorithms, and theory of computation. It is an elective for the Computer Science major.

**What is covered in the course?**

Specific topics covered in this course include:

- Overview of compilation
- Lexical analysis
- Context-free grammars, top-down and bottom-up parsing, error recovery
- Abstract syntax trees, symbol tables
- Lexical scoping, types (primitive, record, arrays, references), type checking
- Object-oriented type systems, subtyping, interfaces, traits
- Three-address code and other intermediate representations
- Code generation, data representation, memory management, object layout
- Code transformation and optimization
- Class hierarchy analysis
- Dataflow analysis
- Register allocation
- Run-time systems, just-in-time compilation, garbage collection

**What is the format of the course?**

This course is offered as a tutorial, and there are no lectures. Students meet with the instructor in pairs each week for 1-2 hours to discuss the readings and problem set questions. In addition, students work in teams of two or three on a semester-long project to build a compiler for a Java-like language. The target is IA-64 assembly code. Students submit weekly project checkpoints that follow the topics discussed in the tutorial meetings. The last three weeks are spent implementing a compiler extension of their own design. The students also attend a weekly 2-hour lab in which general project issues and ideas are discussed among all the groups.

The project assignment, and some of the related problem set material, is based on a project developed by Radu Rugina and others at Cornell University.

Given the nature of tutorials, it can be difficult to quantify the number of hours spent on a topic. Below, I base the hours dedicated to each unit by assuming roughly 3 hours of contact time with the students during each week of the 12-week semester.

**How are students assessed?**

Students are assessed via the preparedness and contributions to the weekly discussions, by their written solutions to problem set questions, and by the quality and correctness of their compiler implementations. Students also give presentations on their final projects to the entire class.

**Course textbooks and materials**

The primary textbook is *Compilers: Principles, Techniques, and Tools* by Aho, Lam, Sethi, and Ullman. Papers from the primary literature are also included when possible. Supplementary material for background reading and for the project is provided on a website.

**Why do you teach the course this way?**

The tutorial format offers a unique opportunity to tailor material specifically to student interest, and to allow them to explore and learn material on their own. The interactions between tutorial partners in the weekly meetings develops communication skills and thought processes that cannot be as easily fostered in lecture-style courses. The group projects also enable students to develop solid software engineering practices and to appreciate the theoretical foundations of each phase of compilation.

The students all enjoy the tutorial-style and collaborative environment it fosters, and they typically rate this class among the most challenging offered at Williams.

**Body of Knowledge coverage**

| KA | Knowledge Unit | Topics Covered | Hours |
|---|---|---|---|
| PL | Basic Type Systems | All (mostly review of existing knowledge) | 2 |
| PL | Program Representation | All | 2 |
| PL | Language Translation and Execution | All | 3 |
| PL | Syntax Analysis | All | 5 |
| PL | Compiler Semantic Analysis | All | 5 |
| PL | Code Generation | All | 5 |
| PL | Runtime Systems | All | 5 |
| PL | Static Analysis | Relevant program representations, such as basic blocks, control-flow graphs, def-use chains, static single assignment, etc.<br>Undecidability and consequences for program analysis<br>Flow-insensitive analyses: type-checking<br>Flow-sensitive analyses: forward and backward dataflow analyses<br>Tools and frameworks for defining analyses<br>Role of static analysis in program optimization<br>Role of static analysis in (partial) verification and bug-finding | 6 |
| SE | Software Design | System design principles<br>Refactoring designs and the use of design patterns. | 2 |

| | | | |
|---|---|---|---|
| AR | Machine-level representation of data | Bits, bytes, and words<br>Representation of records and arrays<br>(This is mostly review, in the context of IA-64) | 1 |
| AR | Assembly level machine organization | Assembly/machine language programming<br>Addressing modes<br>Subroutine call and return mechanisms<br>Heap vs. Static vs. Stack vs. Code segments<br>(This is mostly review, in the context of IA-64) | 2 |

# CSE333: Systems Programming

## University of Washington, Department of Computer Science & Engineering

**Steven D. Gribble**
**gribble@cs.washington.edu**

**http://www.cs.washington.edu/education/courses/cse333/11sp**

**Knowledge Areas that contain topics and learning outcomes covered in the course**

| **Knowledge Area** | **Total Hours of Coverage** |
|---|---|
| Systems Fundamentals (SF) | 8 |
| Operating Systems (OS) | 7 |
| Programming Languages (PL) | 5 |
| Networking and Communications (NC) | 3 |
| Architecture and Organization (AR) | 3 |
| Software Engineering (SE) | 3 |
| Information Management (IM) | 1 |

**Where does the course fit in your curriculum?**

This is an optional course taken by undergraduates in their second year, following at least CS1, CS2, and a hardware/software interface course. Students are encouraged to have taken data abstractions/structures. The course is a prerequisite for several senior-level courses, including operating systems, networking, and computer graphics. Approximately 60 students take the course per offering; it is offered four times per year (i.e., once each quarter, including summer).

**What is covered in the course?**

The major goal of the course is to give students principles, skills, and experience in implementing complex, layered systems. The course includes a quarter-long programming project in which students: (a) build rudimentary data structures in C, such as linked lists, chained hash tables, AVL trees; (b) use them to build an in-memory inverted index and file system crawler; (c) construct a C++-based access methods for writing indexes to disk and accessing disk-based indexes efficiently; and (d) construct a concurrent (threaded or event-driven) web server that exposes a search application.

A substantial portion of the course focuses on giving students in-depth C and C++ skills and experience with practical engineering tools such as debuggers, unit testing frameworks, and profilers. The course stresses the discipline of producing well-structured and readable code, including techniques such as style guidelines and code reviews. Additionally, the course covers topics such as threaded vs. event-driven concurrency, the Linux system call API, memory management, and some security and defensive programming techniques.

The full list of course topics is:

C programming
- pointers, structs, casts; arrays, strings
- dynamic memory allocation

- C preprocessors, multifile programs
- core C libraries
- error handling without exceptions

C++ programming
- class definitions, constructors and destructors, copy constructors
- dynamic memory allocation (new / delete), smart pointers, classes with dynamic data
- inheritance, overloading, overwriting
- C++ templates and STL

Tools and best practices
- compilers, debuggers, make
- leak detectors, profilers and optimization, code coverage
- version control
- code style guidelines; code review

Systems topics: the layers below (OS, compiler, network stack)
- concurrent programming, including threading and asynchronous I/O
- file system API
- sockets API
- understanding the linker / loader
- fork / join, address spaces, the UNIX process model

**What is the format of the course?**
The course is 10 weeks long, with students meeting for 3 1-hour lectures and a 1-hour lab session per week.

**How are students assessed?**
Over the 10 weeks, students complete 4 major parts of the programming assignment, ~15 small programming exercises (handed out at the end of each lecture), ~8 interactive lab exercises, a midterm, and a final exam. Students spend approximately 10-15 hours per week outside of class on the programming assignment and exercises.

**Course textbooks and materials**
Required texts:

Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective
Harbison & Steele, C: A Reference Manual
Lippman, Lajoie & Moo, C++ Primer

Optional text:

Myers, Effective C++ (optional)

**Why do you teach the course this way?**
As mentioned above, a major goal of the course is to give students principles, skills, and experience in implementing complex, layered systems. The course as structured emphasizes significant programming experience in combination with exposure to systems programming topics.

**Body of Knowledge coverage**

| KA | Knowledge Unit | Topics Covered | Hours |
|---|---|---|---|
| SF | Cross-layer Communications | Abstractions, interfaces, use of libraries; applications vs. OS services | 4 |
| SF | Support for Parallelism | thread parallelism (fork-join), event-driven concurrency, client/server web services | 3 |
| SF | Proximity | memory vs. disk latency, demonstrated by in-memory vs. on-disk indexes | 1 |
| AR | Assembly level machine org. | heap, stack, code segments | 2 |
| AR | Memory system org. and arch. | virtual memory concepts | 1 |
| IM | Indexing | building an inverted file / web index; storing and accessing indexes efficiently on disk | 1 |
| NC | Networked applications | client/server; HTTP; multiplexing with TCP; socket APIs | 3 |
| OS | Principles | abstractions, processes, APIs, layering | 3 |
| OS | Concurrency | pthreads interface, basics of synchronization | 3 |
| OS | File systems | files and directories; posix file system API; basics of file search | 1 |
| PL | Object-oriented Programming | OO design, class definition, subclassing, dynamic dispatch (all taught based on C++) | 3 |
| PL | Event-driven programming | Events and event handlers, asynchronous I/O and non-blocking APIs | 2 |
| SE | Tools and environments | Unit testing, code coverage, bug finding tools | 1 |
| SE | Software construction | Coding practices, standards; defensive programming | 1 |
| SE | Software verification validation | Reviews and audits; unit and system testing | 1 |

# CSE332: Data Abstractions

## University of Washington, Seattle, WA
**Dan Grossman**
**djg@cs.washington.edu**

**http://www.cs.washington.edu/education/courses/cse332/**
**(description below based on, for example, the Spring 2012 offering)**

**Knowledge Areas that contain topics and learning outcomes covered in the course**

| Knowledge Area | Total Hours of Coverage |
|---|---|
| AL / Algorithms and Complexity | 19 |
| PD / Parallel and Distributed Computing | 9 |
| DS / Discrete Structures | 3 |
| SDF / Software Development Fundamentals | 2 |

### Where does the course fit in your curriculum?

This is a required course taken by students mostly in their second year, following at least CS1, CS2, and a "Foundations" course that covers much of Discrete Structures. This course is approximately 70% classic data structures and 30% an introduction to shared-memory parallelism and concurrency. It is a prerequisite for many senior-level courses.

### What is covered in the course?

The core of this course is fundamental "classical" data structures and algorithms including balanced trees, hashtables, sorting, priority queues, graphs and graph algorithms like shortest paths, etc. The course includes asymptotic complexity (e.g., big-O notation). The course also includes an introduction to concurrency and parallelism grounded in the data structure material. Concurrent access to shared data motivates mutual exclusion. Independent subcomputations (e.g., recursive calls to mergesort) motivate parallelism and cost models that account for time-to-completion in the presence of parallelism.

More general goals of the course include (1) exposing students to non-obvious algorithms (to make the point that algorithm selection and design is an important and non-trivial part of computer science & engineering) and (2) giving students substantial programming experience in a modern high-level programming language such as Java (to continue developing their software-development maturity).

Course topics:

- Asymptotic complexity, algorithm analysis, recurrence relations
- Review of stacks, queues, and binary search trees (covered in CS2)
- Priority queues and binary heaps
- Dictionaries and AVL trees, B trees, and hashtables
- Insertion sort, selection sort, heap sort, merge sort, quicksort, bucket sort, radix sort
- Lower bound for comparison sorting
- Graphs, graph representations, graph traversals, topological sort, shortest paths, minimum spanning trees
- Simple examples of amortized analysis (e.g., resizing arrays)
- Introduction to multiple explicit threads of execution
- Parallelism via fork-join computations
- Basic parallel algorithms: maps, reduces, parallel-prefix computations

- Parallel-algorithm analysis: Amdahl's Law, work, span
- Concurrent use of shared resources, mutual exclusion via locks
- Data races and higher-level race conditions
- Deadlock
- Condition variables

**What is the format of the course?**

This is a fairly conventional course with 3 weekly 1-hour lectures and 1 weekly recitation section led by a teaching assistant. The recitation section often covers software-tool details not covered in lecture. It is a 10-week course because the university uses a "quarter system" with 10-week terms.

**How are students assessed?**

Students complete 8 written homework assignments, 3 programming projects in Java (1 using parallelism), a midterm, and a final exam.

**Course textbooks and materials**

For the classic data structures material, the textbook is *Data Structures and Algorithm Analysis in Java* by Weiss. For parallelism and concurrency, materials were developed originally for this course and are now used by several other institutions (see url below). Programming assignments use Java, in particular Java's Fork-Join Framework for parallelism.

**Why do you teach the course this way?**

Clearly the most novel feature of this course is the integration of multithreading, parallelism, and concurrency. The paper Introducing Parallelism and Concurrency in the Data Structures Course, by Dan Grossman and Ruth E. Anderson, published in SIGCSE2012, provides additional rationale and experience for this approach. In short, data structures provides a rich source of canonical examples to motivate both parallelism and concurrency. Moreover, an introduction to parallelism benefits from the same mix of algorithms, analysis, programming, and practical considerations that is the main ethos of the data structures course.

**Body of Knowledge coverage**

| KA | Knowledge Unit | Topics Covered | Hours |
|---|---|---|---|
| AL | Basic Analysis | All except the Master Theorem | 3 |
| AL | Fundamental Data Structures and Algorithms | All topics except string/text algorithms. Also, the preceding CS2 course covers some of the simpler topics, which are then quickly reviewed | 10 |
| AL | Advanced Data Structures Algorithms and Analysis | Only these: AVL trees, topological sort, B-trees, and a brief introduction to amortized analysis | 6 |
| DS | Graphs and Trees | All topics except graph isomorphism | 3 |
| PD | Parallelism Fundamentals | All | 2 |
| PD | Parallel Decomposition | All topics except actors and reactive processes, but at only a cursory level | 1 |
| PD | Communication and | All topics except Consistency in shared memory models, | 2 |

| | | | |
|---|---|---|---|
| | Coordination | Message passing, Composition, Transactions, Consensus, Barriers, and Conditional actions. (The treatment of atomicity and deadlock is also very elementary.) | |
| PD | Parallel Algorithms, Analysis, and Programming | All Core-Tier-2 topics; none of the Elective topics | 4 |
| SD F | Fundamental Data Structures | Only priority queues (all other topics are in CS1 and CS2) | 2 |

**Additional topics**

The parallel-prefix algorithm

**Other comments**

The parallelism and concurrency materials are freely available at http://www.cs.washington.edu/homes/djg/teachingMaterials/spac/.

# Discrete Mathematics (MAT 267)

## Union County College, Cranford, NJ
## Community College

**Dr. Cynthia Roemer, Department Chair**
**roemer@ucc.edu**
**www.ucc.edu**

**Per College policy, all course materials are password-protected. Instructional resources are available upon email request.**

**Knowledge Areas that contain topics and learning outcomes covered in the course**

| **Knowledge Area** | **Total Hours of Coverage** |
|---|---|
| *Discrete Structures (DS)* | *42 hours* |

**Where does the course fit in your curriculum?**
Union County College offers this course both Fall and Spring semesters. Computer Science majors typically complete the required Discrete Mathematics course as sophomores. Students are eligible to enroll in this course after passing pre-calculus (MAT 143) with a grade of C or better, or scoring well enough on the College Level Mathematics Test to place directly into it. CS majors are also required to complete Calculus I (MAT 171).

**What is covered in the course?**
This course will develop advanced mathematics skills appropriate for students pursuing STEM studies such as Engineering, Science, Computer Science, and Mathematics. Topics include sets, numbers, algorithms, logic, computer arithmetic, applied modern algebra, combinations, recursion principles, graph theory, trees, discrete probability, and digraphs.

**What is the format of the course?**
This course earns 3 credit hours and consists of 3 lecture hours per week for 14 weeks. Discrete Mathematics offered at Union County College currently meets twice per week for 80 minutes each.

**How are students assessed?**
Students are assessed on a combination of homework, quizzes/tests, group activities, discussion, projects, and a comprehensive final exam. Students are expected to complete homework assignments/projects on a weekly basis. For a typical student, each assignment will require at least 3 hours to complete.

**Course textbooks and materials**
Text: Discrete Mathematics by Sherwood Washburn, Thomas Marlowe, & Charles T. Ryan (Addison-Wesley)

A graphing calculator (e.g. TI-89) and a computer algebra system (e.g. MAPLE) are required for completing certain homework exercises and projects.

Union County College has a Mathematics Success Center that is available for tutoring assistance for all mathematics courses.

**Why do you teach the course this way?**
Discrete Mathematics is a transfer-oriented course designed to meet the requirements of Computer Science, Engineering and Mathematics degree programs. Many of the Computer Science majors at Union County College matriculate to New Jersey Institute of Technology. Furthermore, this course is designed to meet the following program objectives. (Also see Other Comments below). Upon successful completion of this course, students

will be able to:

- Demonstrate critical thinking, analytical reasoning, and problem solving skills
- Apply appropriate mathematical and statistical concepts and operations to interpret data and to solve problems
- Identify a problem and analyze it in terms of its significant parts and the information needed to solve it
- Formulate and evaluate possible solutions to problems, and select and defend the chosen solutions
- Construct graphs and charts, interpret them, and draw appropriate conclusions

**Body of Knowledge coverage**

| KA | Knowledge Unit | Topics Covered | Hours |
|---|---|---|---|
| DS | Sets, Relations, Functions | all topics | 6 |
| DS | Basic Logic | all topics | 9 |
| DS | Proof Techniques | all topics | 9 |
| DS | Basics of Counting | all topics | 7 |
| DS | Graphs and Trees | all topics except Graph Isomorphism (core tier-2) | 6 |
| DS | Discrete Probability | all topics except Conditional Independence (core tier-2) | 5 |

**Other comments**

Correlation of Program Objectives, Student Learning Outcomes, and Assessment Methods

| Program Objectives | Student Learning Outcomes | Assessment Methods |
|---|---|---|
| Demonstrate critical thinking, analytical reasoning, and problem solving skills | Recognize, identify, and solve problems using set theory, elementary number theory, and discrete probability<br><br>Recognize, identify, and apply the concepts of functions and relations and graph theory in problem solving<br><br>Apply proof techniques in logic | **Written:** Homework assignments, examinations in class, and projects to be completed at home<br><br>**Verbal:** Classroom exercises and discussion |
| Apply appropriate mathematical and statistical concepts and operations to interpret data and to solve problems | Recognize, identify, and solve problems using set theory, elementary number theory, and discrete probability<br><br>Recognize, identify, and apply the concepts of functions and relations | **Written:** Homework assignments, examinations in class, and projects to be completed at home<br><br>**Verbal:** Classroom exercises and discussion |

| | | |
|---|---|---|
| | and graph theory in problem solving | |
| Identify a problem and analyze it in terms of its significant parts and the information needed to solve it | Recognize, identify, and solve problems using set theory, elementary number theory, and discrete probability<br><br>Recognize, identify, and apply the concepts of functions and relations and graph theory in problem solving<br><br>Apply proof techniques in logic | **Written:** Homework assignments, examinations in class, and projects to be completed at home<br><br>**Verbal:** Classroom exercises and discussion |
| Formulate and evaluate possible solutions to problems, and select and defend the chosen solutions | Recognize, identify, and solve problems using set theory , elementary number theory, and discrete probability<br><br>Recognize, identify, and apply the concepts of functions and relations and graph theory in problem solving<br><br>Apply proof techniques in logic | **Written:** Homework assignments, examinations in class, and projects to be completed at home<br><br>**Verbal:** Classroom exercises and discussion |
| Construct graphs and charts, interpret them, and draw appropriate conclusions | Recognize, identify, and apply the concepts of functions and relations and graph theory in problem solving | **Written:** Homework assignments, examinations in class, and projects to be completed at home<br><br>**Verbal:** Classroom exercises and discussion |

# CS 250 - Discrete Structures I

## Portland Community College, 12000 SW 49th Ave, Portland, OR 97219

**Doug Jones**
**cdjones@pcc.edu**

**Knowledge Areas that contain topics and learning outcomes covered in the course**

| Knowledge Area | Total Hours of Coverage |
|---|---|
| Discrete Structures (DS) | 26 |
| Algorithms and Complexity (AL) | 4 |

### Where does the course fit in your curriculum?

CS 250 is the first course in a two-term required sequence in discrete mathematics for Computer Science transfer students. Students typically complete the sequence in their second year.

College algebra and 1 term of programming are pre-requisites for CS 250. The second course in the sequence (CS 251) requires CS 250 as a pre-requisite.

Approximately 80 students per year complete the discrete mathematics sequence (CS 250 and CS 251).

### What is covered in the course?

- Introduction to the Peano Axioms and construction of the natural numbers, integer numbers, rational numbers, and real numbers.
- Construction and basic properties of monoids, groups, rings, fields, and vector spaces.
- Introduction to transfinite ordinals and transfinite cardinals, and Cantor's diagonalization methods
- Representation of large finite natural numbers using Knuth's "arrow notation"
- Introduction to first order propositional logic, logical equivalence, valid and invalid arguments
- Introduction to digital circuits
- Introduction to first order monadic predicate logic, universal and existential quantification, and predicate arguments
- Elementary number theory, prime factors, Euclid's algorithm
- Finite arithmetic, Galois Fields, and RSA encryption
- Proof techniques, including direct and indirect proofs, proving universal statements, proving existential statements, proof forms, common errors in proofs
- Sequences, definite and indefinite series, recursive sequences and series
- Developing and validating closed-form solutions for series
- Well ordering and mathematical induction
- Introduction to proving algorithm correctness
- Second order linear homogeneous recurrence relations with constant coefficients
- General recursive definitions and structural induction
- Introduction to classical (Cantor) set theory, Russell's Paradox, introduction to axiomatic set theory (Zermelo-Fraenkel with Axiom of Choice).
- Set-theoretic proofs
- Boolean algebras
- Halting Problem

### What is the format of the course?

CS 250 is a 4 credit course with 30 lecture hours and 30 lab hours. Classes typically meet twice per week for lecture, with lab sessions completed in tutoring labs outside of lecture.

Course material is available online, but this is not a distance learning class and attendance at lectures is required.

**How are students assessed?**

Students are assessed using in-class exams and homework. There are 5 in-class exams that count for 40% of the student's course grade, and 5 homework assignments that account for 60% of the student's course grade. In-class exams are individual work only, while group work is permitted on the homework assignments.

It is expected that students will spend 10 to 15 hours per week outside of class time completing their homework assignments. Surveys indicate a great deal of variability in this - some students report spending 6 hours per week to complete assignments, other report 20 or more hours per week.

**Course textbooks and materials**

The core text is Discrete Mathematics with Applications by Susanna S. Epp (Brooks-Cole/Cengage Learning). The text is supplemented with instructor-developed material to address topics not covered in the core text.

Students are encouraged to use computer programs to assist in routine calculations. Many students write their own programs, some use products such as Maple or Mathematica. Most calculators are unable to perform the calculations needed for this course. No specific tools are required.

**Why do you teach the course this way?**

This is a transfer course designed to meet the lower-division requirements of Computer Science and Engineering transfer programs in the Oregon University System with respect to discrete mathematics. As such, it serves many masters - there is no consistent set of requirements across all OSU institutions.

The majority of Portland Community College transfer students matriculate to Portland State University, Oregon Institute of Technology, or Oregon State University, and these institutions have the greatest influence on this course. PCC changes the course content as needed to maintain compatibility with these institutions.

The most recent major course revision occurred approximately 24 months ago, although minor changes tend to occur every Fall term. Portland State University is reviewing all of their lower-division Computer Science offerings, and when they complete their process PCC expects a major revision of CS 250 and CS 251 will be required.

Students generally consider the discrete mathematics sequence to be difficult. Most students have studied some real number algebra, analysis, and calculus, but often have very limited exposure to discrete mathematics prior to this sequence.

**Body of Knowledge coverage**

| KA | Knowledge Unit | Topics Covered | Hours |
|---|---|---|---|
| AL | Basic Analysis | Differences among best, expected, and worst case behaviours<br><br>Big-O, Big-Omega, Big-Theta definitions<br><br>Complexity classes<br><br>Note: Remainder of Basic Analysis topics covered in CS 251 | 4 |
| DS | Basic Logic | Propositional logic, connectives, truth tables, normal forms, validity, inference, predicate logical, quantification, limitations | 10 |

| | | | |
|---|---|---|---|
| DS | Proof Techniques | Implications, equivalences, converse, inverse, contrapositive, negation, contradiction, structure, direct proofs, disproofs, natural number induction, structural induction, weak/string induction, recursion, well orderings | 10 |
| DS | Basics of Counting | Basic modular arithmetic<br><br>Other counting topics in CS 251 | 2 |
| DS | Sets, Relations, Functions | Sets only: Venn diagrams, union, intersection, complement, product, power sets, cardinality, proof techniques.<br><br>Relations and functions covered in CS 261 | 4 |

**Additional topics**

Elementary number theory, Peano Axioms, Zermelo-Fraenkel Axioms, Knuth arrow notation, simple digital circuits, simple encryption/decryption

# CS 251 - Discrete Structures II

**Portland Community College, 12000 SW 49th Ave, Portland, OR 97219**
**Doug Jones**
**cdjones@pcc.edu**

**Knowledge Areas that contain topics and learning outcomes covered in the course**

| Knowledge Area | Total Hours of Coverage |
|---|---|
| Discrete Structures (DS) | 22 |
| Algorithms and Complexity (AL) | 8 |

**Where does the course fit in your curriculum?**

CS 251 is the second course in a two-term required sequence in discrete mathematics for Computer Science transfer students. Students typically complete the sequence in their second year.

College algebra (PCC's MTH 111 course) and 1 term of programming (PCC's CS 161 course) are pre-requisites for CS 250. The second course in the sequence (CS 251) requires CS 250 as a pre-requisite.

Approximately 80 students per year complete the discrete mathematics sequence (CS 250 and CS 251).

**What is covered in the course?**

- Set-based theory of functions, Boolean functions
- Injection, surjection, bijection
- Function composition
- Function cardinality and computability
- General set relations
- Equivalence relations
- Total and partial orderings
- Basic counting techniques: multiplication rule, addition rule, Direclet's Box Principle
- Combinations and permutations
- Pascal's Formula and the Binomial Theorem
- Kolmogorov Axioms and expected value
- Markov processes
- Conditional probability and Bayes' Theorem
- Classical graph theory: Euler and Hamilton circuits
- Introduction to spectral graph theory, isomorphisms
- Trees, weighted graphs, spanning trees
- Algorithm analysis
- Formal languages
- Regular expressions
- Finite-state automata

**What is the format of the course?**

CS 251 is a 4 credit course with 30 lecture hours and 30 lab hours. Classes typically meet twice per week for lecture, with lab sessions completed in tutoring labs outside of lecture.

Course material is available online, but this is not a distance learning class and attendance at lectures is required.

### How are students assessed?

Students are assessed using in-class exams and homework. There are 5 in-class exams that count for 40% of the student's course grade, and 5 homework assignments that account for 60% of the student's course grade. In-class exams are individual work only, while group work is permitted on the homework assignments.

It is expected that students will spend 10 to 15 hours per week outside of class time completing their homework assignments. Surveys indicate a great deal of variability in this - some students report spending 6 hours per week to complete assignments, other report 20 or more hours per week.

### Course textbooks and materials

The core text is Discrete Mathematics with Applications by Susanna S. Epp (Brooks-Cole/Cengage Learning). The text is supplemented with instructor-developed material to address topics not covered in the core text.

Students are encouraged to use computer programs to assist in routine calculations. Many students write their own programs, some use products such as Maple or Mathematica. Most calculators are unable to perform the calculations needed for this course. No specific tools are required.

### Why do you teach the course this way?

This is a transfer course designed to meet the lower-division requirements of Computer Science and Engineering transfer programs in the Oregon University System with respect to discrete mathematics. As such, it serves many masters - there is no consistent set of requirements across all OSU institutions.

The majority of Portland Community College transfer students matriculate to Portland State University, Oregon Institute of Technology, or Oregon State University, and these institutions have the greatest influence on this course. PCC changes the course content as needed to maintain compatibility with these institutions.

The most recent major course revision occurred approximately 24 months ago, although minor changes tend to occur every Fall term. Portland State University is reviewing all of their lower-division Computer Science offerings, and when they complete their process PCC expects a major revision of CS 250 and CS 251 will be required.

Students generally consider the discrete mathematics sequence to be difficult. Most students have studied some real number algebra, analysis, and calculus, but often have very limited exposure to discrete mathematics prior to this sequence.

**Body of Knowledge coverage**

| KA | Knowledge Unit | Topics Covered | Hours |
|---|---|---|---|
| AL | Basic Analysis | Empirical measurement and performance<br>Time and space trade-offs in algorithms<br>Recurrence relations<br>Analysis of iterative and recursive algorithms | 4 |
| DS | Sets, Relations, and Functions | Reflexivity, symmetry, transitivity<br>Equivalence relations<br>Partial ordes<br>Surjection, injection, bijection, inverse, composition of functions | 4 |
| DS | Basics of Counting | Counting arguments: cardinality, sum and product rule, IE principle, arithmetic and geometric progressions, pigeonhole principle, permutations, combinations, Pascal's identity, recurrence relations | 10 |

| | | | |
|---|---|---|---|
| DS | Graphs and Trees | Tree, tree traversal, undirected graphs, directed graphs, weighted graphs, isomorphisms, spanning trees | 4 |
| DS | Discrete Probability | Finite probability space, events, axioms and measures, conditional probability, Bayes' Theorem, independence, Bernoulli and binomial variables, expectation, variance, conditional independence | 4 |
| AL | Basic Automata Computability and Complexity | Finite state machines, regular expressions, Halting problem | 4 |

**Additional topics**

Basic linear algebra, graph spectra, Markov processes

# Ethics & the Information Age (CSI 194)


## Anne Arundel Community College, Arnold, MD

***Cheryl Heemstra (crheemstra@aacc.edu), Jonathan Panitz (japanitz@aacc.edu), Kristan Presnell (lkpresnell@aacc.edu)***


**Knowledge Areas that contain topics and learning outcomes covered in the course**

| Knowledge Area | Total Hours of Coverage |
|---|---|
| Social Issues and Professional Practice (SP) | 41 (plus 4 hours testing) |

### Where does the course fit in your curriculum?

The course covers two requirements: it is a general education humanities course and it is a program requirement for the Information Assurance and Cybersecurity degree. The course is cross-listed as a Philosophy course since it fulfills the general education (core) requirement. Students are free to take the course at any time but are encouraged to take it in the second year. The only prerequisite for the course is eligibility for college English.

### What is covered in the course?

Students learn ethics and moral philosophy as a means for providing a framework for ethically grounded decision making in the information age. Topics include the basic concepts and theories of ethics (moral reasoning and normative frameworks); basic concepts of argumentation and inductive reasoning; an introduction to cyberethics; issues related to networking and network security (threats related to breaches, countering breaches; privacy and personal autonomy (anonymity and accountability, identity theft); intellectual property and ownership rights (Digital Millennium Copyright Act, digital rights management, alternatives to the property model); computing and society, social justice, community, and self-identity digital divide, free speech and censorship; professional ethics and codes of conduct. Four hours are assigned to testing.

### What is the format of the course?

CSI 194 Ethics & the Information Age is taught online and face to face. Faculty teaching the course are free to present the material in any way they like. Generally there is a combination of lectures, class discussion, case studies, written term papers, and team research and presentation.

### Course textbooks and materials

*Ethics & Technology, Ethical Issues in an Age of Information and Communication Technology, Third Edition* by Herman T. Tavani; John Wiley & Sons, Inc., 2011

### Why do you teach the course this way?

Early in our computer security and networking programs, students are trusted with access to the practices, procedures and technologies used to attack and protect valuable information assets and systems. This trust requires an uncompromising commitment to the highest moral and ethical standards. The increasing dependence on and use of technology has created many ethical dilemmas across many disciplines and professions. Many schools are requiring this type of course in programs to address these realities. This is a relatively new area and we would like to be on the cutting edge and provide the work force with students that understand how to apply sound ethical reasoning to various situations. The goal of this course is to provide students in computer and business-related fields with the framework and tools for ethical decision-making in their professions and to heighten ethical awareness of the standards of conduct in these areas.

Ethical decision making is an inductive thought process that is not routinely taught in any normative educational area. This class, which exists on the cutting edge of technological advance, equips the student to think outside the box and apply the new rubric of ethical deliberation to the expanding world of the cyber-arena. The course equips students majoring in Cyberforensics and Cybersecurity to apply practical knowledge to the monumental challenges they will face in their careers as the world of the cyber-arena becomes more and more pervasive and invasive.

When developing this course, we looked at requiring a philosophical ethics class that would count as a general education requirement in the humanities. But the issue we had is that the cases in that course would be divorced from the situations faced by information system security professionals. We wanted the cases to be those that would fit in with our curriculum. In addition, there was not room in our program to have two courses, so we decided to develop one that would count as a general education ethics course and present ethical theory as the basis for examining cases.

We looked at many computer/cyber ethics textbooks and discovered that most of them only provided a cursory overview of ethical theory, if any. This was not enough to warrant classification under general education. We also did not want to require two textbooks because we are mindful of textbook costs for our students. We then found Herman T. Tavani's text that covered ethical theory in depth and provided the practical cases in the field of computer ethics.

**Body of Knowledge coverage**

| KA | Knowledge Unit | Topics Covered | Hours |
|---|---|---|---|
| SP | Social Context | Social Justice<br>Digital divide<br>Distributive justice theories<br>Accessibility issues<br>Social interaction<br>Cultural issues<br>Commerce, Free Speech and Censorship<br>Define the Internet as public or private space<br>Regulatory agencies and laws regarding regulation in physical space<br>Jurisdictional issues with regulating cyberspace<br>Free speech and hate speech<br>Free speech and pornography | 6 |
| SP | Analytical Tools | Introduction to Ethical Thought – values, morals, normative analysis<br>Introduction to Cyberethics<br>Ethical theories – Virtue Ethics, Utilitarianism, Deontology, Just Consequentialism, Social Contract Theory<br>Evaluate stakeholder positions<br>Concepts of argumentation and debate | 12 |
| SP | Professional Ethics | Moral responsibility of a professional<br>Pervasive nature of computing applies to all, not only professionals<br>Professional Codes of Conduct<br>Principles of the Joint IEEE-CS/ACM Code of Ethics and Professional Practice<br>Purpose of a code of ethics<br>Weaknesses of codes of ethics<br>Accountability, responsibility and liability | 4 |
| SP | Intellectual Property | Overview and history of intellectual property – trade secrets, patents, trademarks, copyrights<br>Philosophical views of property – Labor Theory, Utilitarian Theory, Personality Theory<br>Fair Use<br>Digital Millennium Copyright Act.<br>Digital Rights management<br>Alternatives to the property model – GNU project, Open Source Initiative, Creative Commons<br>Software piracy | 6 |
| SP | Privacy and Civil Liberties | Technology's impact on privacy<br>Difference between naturally private and normatively private situations | 6 |

| | | | |
|---|---|---|---|
| | | Philosophical foundations of privacy rights<br>Three types of personal privacy – accessibility, decisional, and informational<br>How different cultures view privacy<br>Public and personal information<br>Information matching technique's impact on privacy<br>Legal rights to privacy<br>Solutions for privacy violations | |
| SP | Security Policies, Laws and Computer Crimes | Need to protect computer data, systems, and networks<br>Ethical issues related to computer security<br>Social engineering<br>Identity theft<br>Computer hacking<br>Security issues related to anonymity on the Internet<br>Cyberterrorism and information warfare<br>Ethical issues related to cybercrime and cyber-related crimes. | 7 |

**Additional topics**

Artificial Intelligence and Ambient Intelligence and the impact upon ethical and moral deliberations

# Ethics in Technology (IFSM304)

## University of Maryland, University College

**Al Fundaburk PhD**
**Albert.fundaburk@faculty.umuc.edu**

**http://www.umuc.edu/undergrad/ugprograms/ifsm.cfm**

**Knowledge Areas that contain topics and learning outcomes covered in the course**

| Knowledge Area | Total Hours of Coverage |
| --- | --- |
| Social Issues and Professional Practice (SP) | 48 |

### Brief description of the course's format and place in the undergraduate curriculum

Recommended pre-requisite: IFSM 201: Concepts and Applications of Information Technology.

IFSM 304 is a required foundation course typically open to all levels from freshman to senior. It is a required course for all programs in IFSM. The course is typically taught in an eight week on-line and hybrid format.

### Course description and goals

This course is a comprehensive study of ethics and of personal and organizational ethical decision making in the use of information systems in a global environment. The aim is to identify ethical issues raised by existing and emerging technologies, apply a structured framework to analyze risk and decision alternatives, and understand the impact of personal ethics and organizational values on an ethical workplace. The objectives of this course are to:

- apply relevant ethical theories, laws, regulations, and policies to decision making to support organizational compliance
- recognize business needs, social responsibilities, and cultural differences of ethical decision making to operate in a global environment
- identify and address new and/or increased ethical issues raised by existing and emerging technologies
- foster and support an ethical workforce through an understanding of the impact of personal ethics and organizational values
- apply a decision-making framework to analyze risks and decision alternatives at different levels of an organization

Course topics

- Technology-related Ethical Global issues (multi-national corporation)
- Decision making frameworks to technology-related ethical issues
- Organizational policy to address the technology-related ethical issue
- Research existing or emerging technology and its ethical impact
- Study group presentation of research on existing or emerging technology and related ethical issues
- a reflective piece on class learning as it applies to ethics in information technology

### Course textbooks, materials, and assignments

Reynolds, George Walter (2012) Ethics in Information Technology, 4th edition, Cengage (ISBN: 1111534128)

The course is taught as both hybrid and on-line. It is a writing intensive course requiring written assignments and student-to-teacher (as well as student-to-student interactions) in discussion conferences. The major assignment consists of eight weekly conferences, including the analysis of an ethical issue drawn from current events with global impact/implications. The conference topics consist of privacy, crime, corporate ethics, social media, and current ethical dilemmas. The significant written assignments include a policy paper, a research paper, a study group developed PowerPoint presentation and the development of a decision matrix to help in analyzing ethical decisions. The course uses the portfolio method to determine student comprehension of the learning outcomes.

**Body of Knowledge coverage**

| KA | Knowledge Unit | Topics Covered | Hours |
|---|---|---|---|
| SP | Social Context | Investigate the implications of social media on individualism versus collectivism and culture. | 2 |
| SP | Analytical Tools | Evaluate stakeholder positions in a given situation.<br>Analyze basic logical fallacies in an argument.<br>Analyze an argument to identify premises and conclusion.<br>Illustrate the use of example and analogy in ethical argument. | 6 |
| SP | Professional Ethics | The strengths and weaknesses of relevant professional codes as expressions of professionalism and guides to decision-making.<br>Analyze a global computing issue, observing the role of professionals and government officials in managing the problem.<br>The consequences of inappropriate professional behavior.<br>Develop a computer use policy with enforcement measures.<br>The consequences of inappropriate personal behavior | 7 |
| Sp | Intellectual Property | The philosophical bases of intellectual property.<br>The rationale for the legal protection of intellectual property.<br>Describe legislation aimed at digital copyright infringements.<br>Identify contemporary examples of intangible digital intellectual property.<br>Justify uses of copyrighted materials.<br>The consequences of theft of intellectual property | 4 |
| Sp | Privacy and Civil Liberties | The philosophical basis for the legal protection of personal privacy.<br>The the fundamental role of data collection in the implementation of pervasive surveillance systems.<br>The impact of technological solutions to privacy problems.<br>The global nature of software piracy | 12 |
| SP | Professional Communication | Write clear, concise, and accurate technical documents following well-defined standards for format and for including appropriate tables, figures, and references.<br>Develop and deliver a good quality formal presentation.<br>Plan interactions (e.g. virtual, face-to-face, shared documents) with others in which they are able to get their point across, and are also able to listen carefully and appreciate the points of others, even when they disagree, and are able to convey to others that they have heard. | 5 |

**Other comments, such as teaching modality: face-to-face, online or blended.**

IFSM 304 is taught as both hybrid and on-line. It is a writing intensive course requiring both written assignments and student to teacher discussion conferences. Both formats are completed in eight weeks. Both formats use the same sequence of events, the primary difference is that the hybrid utilizes a face-to-face component. Both the hybrid and on-line course rely heavily on a faculty led discussion forum to equate theory to practice.

Attachment 1,

Hours Assignment relating to outcomes

| | |
|---|---|
| Social Context (SP) | 5 hours |
| Investigate the implications of social media on individualism versus collectivism and culture. | Week 2 conference, Facebook |
| Analytical Tools (SP) | 10 hours |
| Evaluate stakeholder positions in a given situation. | Current Events Article; Privacy-related Matrix |

| | |
|---|---|
| Analyze basic logical fallacies in an argument. | Current Events Article; Privacy-related Matrix. Week 4 conference Hewlett Packard |
| Analyze an argument to identify premises and conclusion. | Current Events Article; Privacy-related Matrix. Week 6 conference, contributions to economy |
| Illustrate the use of example and analogy in ethical argument. | Current Events Article; Privacy-related Matrix. Week 6 conference, contributions to economy |
| Professional Ethics (SP) | 10 hours |
| Describe the strengths and weaknesses of relevant professional codes as expressions of professionalism and guides to decision-making. | |
| Analyze a global computing issue, observing the role of professionals and government officials in managing the problem. | Current Events Article. Week 5 conference, Computer Crime |
| Describe the consequences of inappropriate professional behavior. | |
| The consequences of inappropriate personal behavior | Current Events Article. Week 5 conference, Computer Crime |
| Develop a computer use policy with enforcement measures. | Organizational Policy paper |
| Intellectual Property (SP) | 7 hours |
| Discuss the philosophical basis of intellectual property. | Week 2 conference, Facebook |
| Discuss the rationale for the legal protection of intellectual property. | Week 2 conference, Facebook |
| Describe legislation aimed at digital copyright infringements. | Week 2 conference, Facebook |
| Identify contemporary examples of intangible digital intellectual property | Week 2 conference, Facebook |
| Justify uses of copyrighted materials.<br>The consequences of theft of intellectual property | Week 2 conference, Facebook |
| Privacy and Civil Liberties (SP) | 10 hours |
| Discuss the philosophical basis for the legal protection of personal privacy. | Reflective paper on class learning; Week 2 conference, Facebook |
| Recognize the fundamental role of data collection in the implementation of pervasive surveillance systems (e.g., RFID, face recognition, toll collection, mobile | Individual research paper on existing or emerging technology and related ethical issue. Week 2 conference, Facebook |
| Investigate the impact of technological solutions to privacy problems. | Individual research paper on existing or emerging technology and related ethical issue. Week 2 conference, Facebook |
| Identify the global nature of software piracy. | Individual research paper on existing or emerging technology and related ethical issue. Week 2 conference, Facebook |
| Professional Communication (SP) | 6 hours |
| Write clear, concise, and accurate technical documents following well-defined standards for format and for including appropriate tables, figures, and references. | Individual research paper on existing or emerging technology and related ethical issue |
| Develop and deliver a good quality formal presentation. | Group PowerPoint presentation |
| Plan interactions (e.g. virtual, face-to-face, shared documents) with others in which they are able to get their point across, and are also able to listen carefully and appreciate the points of others, even when they disagree, and are able to convey to others that they have heard | Group PowerPoint presentation |

3244

# Human Aspects of Computer Science

**Department of Computer Science, University of York**
**Paul Cairns**
**paul.cairns@york.ac.uk**

**Knowledge Areas that contain topics and learning outcomes covered in the course**

| **Knowledge Area** | **Total Hours of Coverage** |
|---|---|
| Human Computer Interaction (HCI) | 18 |

### Where does the course fit in your curriculum?

Students take this course in the first term of Stage 1 (first year) of the undergraduate degree programmes in single honours Computer Science subjects eg BSc in Computer Science, MEng in Computer Systems Software Engineering. There are no pre-reqs for obvious reasons and there are no modules that require it as a pre-requisite. There are usually around 100 students each year.

### What is covered in the course?

The course is centred around Human-Computer Interaction. The topics covered are:

- Experiments and data
- Seeing data
- Inferential statistics
- More inferential tests
- Writing up experiments
- Writing up results and discussion
- User-Centred Design
- Developing requirements
- Personas and scenarios
- Conceptual design
- Interaction Design
- Static Prototyping
- Dynamic Prototyping
- VisualDesign
- Analytic Evaluation
- UsabilityEngineering

### What is the format of the course?

It is face-to-face. Students attend 2 one-hour lectures, 1 two-hour practical and a one-hour reading seminar each week for 9 weeks of the autumn term. Lectures are a mix of lecturing, small group exercises and class discussion. Practicals are primarily individual work or group work related to assessments. Reading seminars are presentations on research papers and class discussions on the papers.

### How are students assessed?

There are three assessments. There are two open assessments for which students work in groups of (ideally) four. The first is two design and conduct an experiment having been giving a basic experimental hypothesis to investigate. The second is to do a user-centred design project though there is not time for iteration or formal evaluation. The third assessment is a closed exam in which students critique a research paper in order to answer short questions on the paper. Students are expected to do 100 hours of work in total on the assessments.

### Course textbooks and materials

Preece, Rogers and Sharp, 3$^{rd}$ edn; Harris, Designing and Reporting Experiments in Psychology, 3$^{rd}$ edn; Cairns and Cox eds, Research Methods in HCI

R is used as the statistics package for analysing experimental data.

There are not formal requirements to use other software for prototyping though students do occasionally use a programming language like Java or C#.

### Why do you teach the course this way?

The course was partly designed to fill a need in the recently revised undergraduate curriculum. The two core content areas were experimental design and HCI. I put these together and produced a research oriented course to show how experiments are done in HCI. It still has a feel of a course of two halves but the idea of considering a common subject area helps and the reading seminars are intended to bind the two halves together by showing the students how research methods lead to advances in HCI that can be used in design.

**Body of Knowledge coverage**

| KA | Knowledge Unit | Topics Covered | Hours |
|---|---|---|---|
| HCI | Foundations | All | 4 |
| HCI | Designing Interaction | Visual design, paper prototyping, UI standards | 2 |
| HCI | Programming Interactive Systems | Choosing interaction styles, designing for resource constrained devices | 2 |
| HCI | UCD and testing | All | 4 |
| HCI | Statistical methods for HCI | Experiment design, EDA, presenting statistical data, using statistical data | 6 |

# Human Computer Interaction

## School of Computing, University of Kent, UK
**Sally Fincher, Michael Kölling**

**Knowledge Areas that contain topics and learning outcomes covered in the course**

| Knowledge Area | Total Hours of Coverage |
|---|---|
| *HCI Human computer interaction* | *12* |

### Where does the course fit in your curriculum?

This is a compulsory first year course, taught in the second semester. It represents one-eighth of student effort for the year. It is taken by about 100-150 students (numbers fluctuate year-on-year).

### What is covered in the course?

This module provides an introduction to human-computer interaction. Fundamental aspects of human physiology and psychology are introduced and key features of interaction and common interaction styles delineated. A variety of analysis and design methods are introduced (e.g. GOMS. heuristic evaluation, user-centred and contextual design techniques). Throughout the course, the quality of design and the need for a professional, integrated and user-centred approach to interface development is emphasised. Rapid and low-fidelity prototyping feature as one aspect of this.

### Course topics

- Evaluating interfaces: heuristic evaluation, GOMS
- Evaluation Data & Empirical Data
- Lo-fi Prototyping
- Colour, Vision & Perception
- Some Features of Human Memory
- Errors
- Controls & widgets; metaphors, icons & symbols
- Elements of visual design
- Documentation

### What is the format of the course?

This is only taught face-to-face. There are two timetabled lecture slots and one small-group, hands-on class slot per week, although we don't always use the lecture slots to deliver lectures.

### How are students assessed?

There are 3 pieces of assessed work:

1. A group assignment to go "out into the world" and observe real behaviour (about 10 hours, over the course of a single week)
2. An individual assignment to undertake analysis of an existing interface (about 10 hours, over 3 weeks)
3. A group design task using lo-fi prototyping (about 40 hours, over 5 weeks)

Together these are worth 50% of the total grade. The remaining 50% is a formal exam.

### Course textbooks and materials

There is no single textbook for this course, although we recommend Dan Saffer's *Designing for Interaction* if students ask. The following readings are required:

- Donald A. Norman: *The Design of Everyday Things*, Chapter 1
- Bruce "Tog" Tognazzini's First Principles of Interaction Design
- Marc Rettig & Aradhana Goel *Designing for Experience: Frameworks and Project Stories*
- Marc Rettig *Prototyping for Tiny Fingers*

- William Horton *Top Ten Blunders by Visual Designers*
- George Miller: *The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information*
- Thomas Kelley The Perfect Brainstorm (chapter from *The Art of Innovation*)
- Bill Buxton The Anatomy of Sketching (chapter from *Sketching User Experiences*)

### Why do you teach the course this way?

HCI used to be an elective course available to second and third year students. In the recent curriculum review (2011) it was moved into the first year. We teach this course as a full-on, hands-on experience.

Some students (typically the less technical) like it very much; some students (typically the more technical) find it troubling and “irrelevant”. Both sorts can find it challenging.

**Body of Knowledge coverage**

| KA | Knowledge Unit | Topics Covered | Hours |
|---|---|---|---|
| HC | Foundations | All | 4 |
| HC | Designing Interaction | All | 4 |
| HC | User-centred design & testing | All | 4 |

# Introduction to Artificial Intelligence


## Case Western Reserve University, Cleveland, OH, USA

**Soumya Ray**

**sray@case.edu**


**Course offered Spring 2012: http://engr.case.edu/ray_soumya/ai_course_exemplar/**

**Knowledge Areas that contain topics and learning outcomes covered in the course**

| Knowledge Area | Total Hours of Coverage |
|---|---|
| Intelligent Systems (IS) | 30 |

### Where does the course fit in your curriculum?

Students take the course typically in either their junior or senior years. It is required for the BS/CS degree and an elective for the BA/CS degree. The minimum required prerequisite for this course is an introductory programming in Java course. A background in data structures and algorithms is strongly recommended but not required. Usually around 40 students take the course each year. This course is required for graduate level artificial intelligence and machine learning courses.

### What is covered in the course?

- Problem solving with search: uninformed, informed search, search for optimization (hill climbing, simulated annealing, genetic algorithms), adversarial search (minimax, game trees)
- Logic and Planning: Propositional Logic, syntactic and model-based inference, first order logic (FOL), FOL inference complexity, unification and resolution, planning as FOL inference, STRIPS encoding, state space and plan space planning, partial order planning.
- Probability and Machine Learning: Axioms of probability, basic statistics (expectation and variance), inference by enumeration, Bayesian networks, inference through variable elimination and Monte Carlo, intro to supervised machine learning, probabilistic classification with naive Bayes, parameter estimation with maximum likelihood, Perceptrons, parameter estimation with gradient descent, evaluating algorithms with cross validation, confusion matrices and hypothesis testing.
- Decision making under uncertainty: Intro to sequential decision making, Markov decision processes, Bellman equation/optimality, value and policy iteration, model-based and model free reinforcement learning, temporal difference methods, Q learning, Function approximation.
- I also have one lecture on natural language processing with a very brief introduction to language models, information retrieval and question answering (Watson), but students are not evaluated on this material.

### What is the format of the course?

2 Classroom lectures 75 minutes each per week. 3 Office hours per week (1.5 instructor/1.5 TA).

### How are students assessed?

The course is divided into 4 parts as outlined above. Each part has two written homework assignments except the last part which has one (7 total). Each written homework is followed by a quiz (closed book) that tests the material on that homework (7 total). Written homeworks typically consists of numerical problems and proofs.

There are five programming assignments: 2 on search (1 A*, 1 game trees), 1 on planning, 1 on probabilistic inference and 1 on Q-learning.

The theoretical part (homeworks + best 6 quizzes) is worth 60%. The programming part is worth 40%.

Students are expected to spend about 6 hours per week on the homework and programming assignments.

All assignments (not quizzes) can be done in pairs (optional).

## Course textbooks and materials

Textbook: Artificial Intelligence: A Modern Approach 3ed, Russell and Norvig, supplemented by notes for the machine learning part

Programs are in Java. Programming assignments are implemented in the SEPIA environment. SEPIA (Strategy Engine for Programming Intelligent Agents) is a strategy game similar to an RTS (e.g. Warcraft, Age of Empires etc) that my students and I have built.

## Why do you teach the course this way?

I reviewed and restructured this course in 2011.

The course is intended to be a broad coverage of AI subfields. Unfortunately AI is too broad to cover everything, but I try to hit many of the key points. It also mostly follows the textbook, which I have found is less confusing for students (some do not like jumping around a book). I try to balance exposure to the theoretical aspects with fun/interesting implementation.

For the quizzes and homework, having many short ones gives more frequent feedback to students about how well they understand the material, as well distributes the risk of them losing too many points in any one assignment because they were having a bad day. I evaluate students in a quiz immediately after a homework because they have the material fresh in their minds at that point. Doing assignments in pairs builds teamwork and community, reduces the pressure of assignments and should be more fun.

From the student evaluations, the course is not viewed as “challenging” as such, but work-intensive.

**Body of Knowledge coverage**

| KA | Knowledge Unit | Topics Covered | Hours |
|---|---|---|---|
| IS | Fundamental Issues | Overview of AI problems, Examples of successful recent AI applications<br>What is intelligent behavior?<br>The Turing test<br>Rational versus non-rational reasoning<br>Nature of environments<br>Fully versus partially observable<br>Single versus multi-agent<br>Deterministic versus stochastic<br>Static versus dynamic<br>Discrete versus continuous<br>Nature of agents<br>Autonomous versus semi-autonomous<br>Reflexive, goal-based, and utility-based<br>The importance of perception and environmental interactions | 1.0 |
| IS | Basic Search Strategies | Problem spaces (states, goals and operators), problem solving by search<br>Uninformed search (breadth-first, depth-first, depth-first with iterative deepening)<br>Heuristics and informed search (hill-climbing, generic best-first, A*)<br>Space and time efficiency of search<br>Two-player games (Introduction to minimax search) | 4.5 |
| IS | Basic Knowledge | Review of propositional and predicate logic (cross-reference DS/Basic | 7.5 |

| | | | |
|---|---|---|---|
| | Representation and Reasoning | Logic)<br>Resolution and theorem proving (propositional logic only)<br>DPLL, GSAT/WalkSAT<br>First Order Logic resolution<br>Review of probabilistic reasoning, Bayes theorem, inference by enumeration<br>Review of basic probability (cross-reference DS/Discrete Probability)<br>Random variables and probability distributions<br>Axioms of probability<br>Probabilistic inference<br>Bayes' Rule | |
| IS | Basic Machine Learning | Definition and examples of broad variety of machine learning tasks, including classification<br>Inductive learning<br>Statistical learning with Naive Bayes and Perceptrons<br>Maximum likelihood and gradient descent parameter estimation<br>Cross validation<br>Measuring classifier accuracy, Confusion Matrices | 6.0 |
| IS | Advanced Search | Constructing search trees<br>Stochastic search<br>Simulated annealing<br>Genetic algorithms<br>Implementation of A* search, Beam search<br>Minimax Search, Alpha-beta pruning<br>Expectimax search and chance nodes | 2.25 |
| IS | Advanced Representation and Reasoning | Totally-ordered and partially-ordered Planning | 1.75 |
| IS | Reasoning Under Uncertainty | Conditional Independence<br>Bayesian networks<br>Exact inference (Variable elimination)<br>Approximate Inference (basic Monte Carlo) | 2.0 |
| IS | Agents | Markov Decision Processes, Bellman Equation/Optimality, Value and Policy Iteration | 1.25 |
| IS | Natural Language Processing | Language models, n-grams, vector space models, bag of words, text classification, information retrieval, pagerank, information extraction, question-answering (Watson) [Overview, students are not evaluated on NLP] | 1.25 |
| IS | Advanced Machine Learning | Model based and model free reinforcement learning, temporal difference learning, Q learning, function approximation | 2.5 |

# Introduction to Artificial Intelligence

**Department of Computer Science, University of Hartford**
**Ingrid Russell**
**irussell@hartford.edu**

*http://uhaweb.hartford.edu/compsci/ccli/*

**Knowledge Areas that contain topics and learning outcomes covered in the course**

| **Knowledge Area** | **Total Hours of Coverage** |
|---|---|
| *Intelligent Systems (IS)* | *24* |
| *Programming Languages (PL)* | 3 |

## Where does the course fit in your curriculum?

The course is typically taken in the Junior or Senior year as an upper level elective. It is taken mostly by Computer Science and Computer Engineering students. The Data Structures course is the prerequisite. There is no required course that has this course as a prerequisite. Instructors may offer independent study courses that require this course as a prerequisite. Student enrolment range is 10-24 per offering.

## What is covered in the course?

The AI topics below follow the topic coverage in Russell & Norvig book.

Introduction to Lisp

Fundamental Issues
What is AI? Foundations of AI, History of AI.

Intelligent Agents
Agents and Environments, Structure of Agents.

Problem Solving by Searching
Problem Solving Agents, Searching for Solutions, Uninformed Search Strategies: Breadth-First Search, Depth-First Search, Depth-limited Search, Iterative Deepening Depth-first Search, Comparison of Uninformed Search Strategies.

Informed Search and Exploration
Informed (Heuristic) Search Strategies: Greedy Best-first Search, A* Search, Heuristic Functions, Local Search Algorithms, Optimization Problems.

Constraint Satisfaction Problems
Backtracking Search for CSPs, Local Search for CSPs.

Adversarial Search
Games, Minimax Algorithm, Alpha-Beta Pruning.

Reasoning and Knowledge Representation
Introduction to Reasoning and Knowledge Representation, Propositional Logic, First Order Logic, Semantic Nets, Other Knowledge Representation Schemes.

Reasoning with Uncertainty & Probabilistic Reasoning
Acting Under Uncertainty, Bayes' Rule, Representing Knowledge in an Uncertain

Domain, Bayesian Networks.

Machine Learning
Forms of Learning, Decision Trees and the ID3 Algorithm, Nearest Neighbor, Statistical Learning.

**What is the format of the course?**
The course is a face to face course with 2.5 contact hours per week. It is approximately 50% lecture/discussion and 50% lab sessions.

**How are students assessed?**
Two exams and a final exam are given that constitute 45% of the grade. Assignments include programming and non- programming type problems. Students are given 1-1.5 weeks to complete. A term-long project with 4-5 deliverables is assigned. The project involves the development of a machine learning system. More information on our approach is included in Section VI.

**Grading Policy**

| | |
|---|---|
| Exams 1, 2 | 30% |
| Final Exam | 15% |
| Assignments | 15% |
| Term Project | 30% |
| Class Presentation | 10% |

**Course textbooks and materials**
Book: Artificial Intelligence by Russell and Norvig
Software: Allegro Common Lisp

**Why do you teach the course this way?**
Our approach to teaching introductory artificial intelligence unifies its diverse core topics through a theme of machine learning, through a set of hands-on term long projects, and emphasizes how AI relates more broadly with computer science. Machine learning is inherently connected with the AI core topics and provides methodology and technology to enhance real-world applications within many of these topics. Using machine learning as a unifying theme is an effective way to tie together the various AI concepts while at the same time emphasizing AI's strong tie to computer science. In addition, a machine learning application can be rapidly prototyped, allowing learning to be grounded in engaging experience without limiting the important breadth of an introductory course. Our work involves the development, implementation, and testing of a suite of projects that can be closely integrated into a one-term AI course.

With funding from NSF, our multi-institutional project, Machine Learning Experiences in Artificial Intelligence (MLeXAI), involved the development and implementation of a suite of 26 adaptable machine learning projects that can be closely integrated into a one-term AI course. Our approach would allow for varying levels of mathematical sophistication, with implementation of concepts being central to the learning process. The projects have been implemented and tested at over twenty institutions nationwide. Associated curricular modules for each project have also been developed. Each project involves the design and implementation of a learning system which enhances a particular commonly-deployed AI application. In addition, the projects provide students with an opportunity to address not only core AI topics, but also many of the issues central to computer science, including algorithmic complexity and scalability problems. The rich set of applications that students can choose from spans several areas including network security, recommender systems, game playing, intelligent agents, computational chemistry, robotics, conversational systems, cryptography, web document classification, computer vision, data integration in databases, bioinformatics, pattern recognition, and data mining.

Additional information on MLeXAI, the machine learning projects, and the participating faculty and institutions is available at the project web page at: http://uhaweb.hartford.edu/compsci/ccli.

**Body of Knowledge coverage**

| KA | Knowledge Unit | Topics Covered | Hrs |
|---|---|---|---|
| | Lisp | A Brief Introduction | 3 |
| IS | Fundamental Issues | AI problems, Agents and Environments, Structure of Agents, Problem Solving Agents | 3 |
| IS | Basic Search Strategies | Problem Spaces, Uninformed Search (Breadth-First, Depth-First Search, Depth-first with Iterative Deepening), Heuristic Search (Hill Climbing, Generic Best-First, A*), Constraint Satisfaction (Backtracking, Local Search) | 5 |
| IS | Advanced Search | Constructing Search Trees, Stochastic Search, A* Search Implementation, Minimax Search, Alpha-Beta Pruning | 3 |
| IS | Basic Knowledge Representation and Reasoning | Propositional Logic, First-Order Logic, Forward Chaining and Backward Chaining, Introduction to Probabilistic Reasoning, Bayes Theorem | 3 |
| IS | Advanced Knowledge Representation and Reasoning | Knowledge Representation Issues, Non-monotonic Reasoning, Other Knowledge Representation Schemes. | 3 |
| IS | Reasoning Under Uncertainty | Basic probability, Acting Under Uncertainty, Bayes' Rule, Representing Knowledge in an Uncertain Domain, Bayesian Networks | 3 |
| IS | Basic Machine Learning | Forms of Learning, Decision Trees, Nearest Neighbor Algorithm, Statistical-Based Learning such as Naïve Bayesian Classifier. | 4 |

## Additional topics:

A brief introduction to 1-2 additional AI sub-fields. (2 hours)

## Additional Comments

The machine learning algorithms covered vary based on the machine learning project selected for the course.

**Acknowledgement**: This work is funded in part by the National Science Foundation DUE-040949 and DUE-0716338.

## References:

- Russell, I., Coleman, S., Markov, Z. 2012. A Contextualized Project-based Approach for Improving Student Engagement and Learning in AI Courses. Proceedings of CSERC 2012 Conference, ACM Press, New York, NY, 9-15, DOI= http://doi.acm.org/10.1145/2421277.242127

- Russell, I., Markov, Z., Neller,T., Coleman, S. 2010. MLeXAI: A Project-Based Application Oriented Model, The ACM Transactions on Computing Education, 20(1), pages 17-36.

- Russell, I., Markov, Z. 2009. Project MLeXAI Home Page, http://uhaweb.hartford.edu/compsci/ccli/.

- Russell, S. and Norvig, P. 2010. Artificial Intelligence: A Modern Approach, Upper Saddle River, NJ: Prentice-Hall.

# Introduction to Parallel Programming

**Nizhni Novgorod State University, Nizhni Novgorod, Russia**
**Victor Gergel**
**gergel@unn.ru**

**http://www.hpcc.unn.ru/?doc=98 (In Russian)**
**http://www.hpcc.unn.ru/?doc=107 (In English)**

**Knowledge Areas that contain topics and learning outcomes covered in the course**

| Knowledge Area | Total Hours of Coverage |
|---|---|
| PD - Parallel and Distributed Computing | 20 |

### Where does the course fit in your curriculum?

3rd year (5th semester). Compulsory course. Usually has 40-50 students.
The students are supposed to know the following:

- CS101 "Introduction to Programming ",
- CS105 "Discrete mathematics",
- CS220 "Computer Architecture",
- CS225 "Operating Systems",
- CS304 "Numerical Methods ".

Experience with programming in C is required in order to carry out the laboratory works.

### What is covered in the course?

- Introduction to Parallel Programming
- Overview of Parallel System Architectures
- Modeling and Analysis of Parallel Computations
- Communication Complexity Analysis of Parallel Algorithms
- Parallel Programming with MPI
- Parallel Programming with OpenMP
- Principles of Parallel Algorithm Design
- Parallel Algorithms for Solving Time Consuming Problems (Matrix calculation, System of linear equations, Sorting, Graph algorithms, Solving PDE, Optimization)
- Modeling the parallel program executing

### What is the format of the course?

In-person lectures. Lectures: 36 contact hours. Labs: 18 hours. Homework: 18 hours.

### How are students assessed?

Assignments include reading papers and implementing programming exercises.

### Course textbooks and materials

- Gergel V.P. (2007) Theory and Practice of Parallel Programming. – Moscow, Intuit. (In Russian)
- Gergel V.P. (2010) High-Performance Computations for Multiprocessor Multicore Systems. – Moscow: Moscow State University. (In Russian)
- Quinn, M. J. (2004). Parallel Programming in C with MPI and OpenMP. – New York, NY: McGraw-Hill.
- Grama, A., Gupta, A., Kumar V. (2003, 2nd edn.). Introduction to Parallel Computing. – Harlow, England: Addison-Wesley.
- To develop parallel programs C/C++ is used, MS Visual Studio, Intel Parallel Studio, cluster under MS HPC Server 2008.

**Why do you teach the course this way?**

The goal of the course is to study the mathematical models, methods and technologies of parallel programming for multiprocessor systems. Learning the course is sufficient for a successful start to practice in the area of parallel programming.

The course has a good reputation among students. The students that are studied this course were the winner in the track “Max Linpack performance” of the Cluster Student Competition at Supercomputing 2011. The course was a part of the proposal that was the winner of the contest “Curriculum Best Practices: Parallelism and Concurrence” of European Association “Informatics Europe” (2011) – see. http://www.informatics-europe.org/services/curriculum-award/105-curriculum-award-2011.html

**Body of Knowledge coverage**

| KA | Knowledge Unit | Topics Covered | Hours |
|---|---|---|---|
| PD | Parallelism Fundamentals | Overview: Information dependencies analysis, decomposition, synchronization, message passing | 2 |
| PD | Parallel Decomposition | Data and task decomposition, Domain specific (geometric) decomposition | 0.5 |
| PD | Communication and Coordination | Basic communication operations, Evaluating communication overhead | 2 |
| PD | Parallel Algorithms, Analysis, and Programming | Characteristics of parallel efficiency,<br>Spectrum of parallel algorithms (Matrix calculation, System of linear equations, Sorting, Graph algorithms, Solving PDE, Optimization)<br>OpenMP, MPI, MS VS, Intel Parallel Studio | 10 |
| PD | Parallel Architecture | Structure of the MIMD class of Flynn’s taxonomy,<br>SMP, Clusters, NUMA | 1.5 |
| PD | Parallel Performance | Characteristics of parallel efficiency: speed-up, cost, scalability, isoefficiency. Theoretical prediction of parallel performance and comparing with computational results | 2 |
| PD | Formal Models and Semantics | Information dependencies analysis, Evaluating characteristics of parallel efficiency (superlinear and linear speed-up, max possible speed up for a given problem, speed-up of a given parallel algorithm), Equivalent transformation of parallel programs | 2 |

**Additional topics**

Isoefficiency,
Redundancy of parallel computations,
Classic illustrative parallel problems (readers-writers, dining philosophers, sleeping barbarian, etc.),
Tools for development of parallel programs,
Formal models based on Information dependencies analysis

# Issues in Computing
## Saint Xavier University

**Florence Appel**
**appel@sxu.edu**

**Per University requirements, all course materials are password-protected. Resources are available upon request.**

**Knowledge Areas that contain topics and learning outcomes covered in the course**

| **Knowledge Area** | **Total Hours of Coverage** |
|---|---|
| Social Issues and Professional Practice | 42 |

### Where does the course fit in your curriculum?

*Issues in Computing* is a *required* 300-level course intended for all junior and senior computing majors. All students must have successfully completed English composition and Speech courses prior to their enrollment in the course. We do admit students who are not at the junior/senior level if they are computing practitioners or have been computing practitioners prior to enrolling in our program. The course is offered annually and has an average enrollment of 25.

### What is covered in the course?

In the context of widespread computer usage and society's ever-growing dependence on computer technology, the course focuses on issues of ethics for the computing professional. A list of topics:

- Introduction to Computer Ethics
- Survey of the tools of ethical analysis
- Practical applications of the tools of ethical analysis
- Professional ethics
- Privacy issues
- Intellectual property protection issues
- Freedom of expression and the Internet
- Ethical dimensions of computer system reliability
- Digital Divide
- Social impact of technology in the workplace, in education, in healthcare

### What is the format of the course?

It is a 3 credit hour class that has traditionally been face-to-face, with a growing blended online component. Plans to offer it completely online are underway. It has been offered in two 1.5 blocks as well as in one 3 hour block. Within the three contact hours, the distribution of activity is roughly:

- Lecture 15%
- Full class discussion 20%
- Small group work 25%
- Student reports on small group work 15%
- Peer review of assignments 20%
- Individual student presentation 5%

### How are students assessed?

The basic grading scheme is designed to emphasize student participation, writing and reflection. Students are expected to spend 9-12 hours per week on outside classwork:

Homework and class participation…………………………. 40%
All classroom discussions
Quizzes/short writing assignments on readings
Web-based forum discussion postings
Essays on specified topics………………………………………. 30%

Ethical analyses of given situations

Exams……………………………………………………………………. 30%

Take-home midterm

In-class final

**Course textbooks and materials**

Current textbook is Brinkman & Sanders, *Ethics in a Computing Culture*, which is supplemented by Abelson et al, *Blown to Bits*, available free of charge in pdf format from bitsbook.com. These books are supplemented *heavily* by current and recent articles from the New York Times, Atlantic Monthly, Technology Review, New Yorker, Chicago Tribune (local), Huffington Post etc. Readings on computer ethics theory come from the ACM and IEEE digital libraries as well as other sources. We also make use of a variety of websites, including those sponsored by civil liberties organizations (e.g., eff.org, aclu.org), privacy advocacy groups (e.g., epic.org, privacyrights.org), intellectual property rights groups, free/open source advocates (e.g., fsf.org), government sites (e.g., ftc.gov, fcc.gov); we also draw from ethics education sites such as Michael Sandel's Justice website (justiceharvard.org) and Lawrence Hinman's ethics education site (ethics.sandiego.edu). Additionally, we reference a library collection of books and films on a variety of computer ethics and social impact themes.

**Why do you teach the course this way?**

The overarching goal is to educate students about the practice of professional ethics in the computing field. We situate the special problems faced by computer professionals in the context of widespread computer usage and society's ever-growing dependence on computer technology. We work to develop within our students the critical thinking skills required to identify ethical issues and apply the tools of ethical analysis to address them. Students find this course to be very demanding; they almost always outside their comfort zones. The curriculum for *Issues in Computing* was last reviewed in 2010.

**Body of Knowledge coverage**

| KA | Knowledge Unit | Topics Covered | Hours |
|---|---|---|---|
| SP | Social Context | All | 6 |
| SP | Analytical Tools | All | 6 |
| SP | Professional Ethics | All | 6* |
| SP | Intellectual Property | All | 6 |
| SP | Privacy/Civil Liberties | All | 6 |
| SP | Professional Communication** | Dynamics of oral, written, electronic team communication<br>Communicating effectively with stakeholders<br>Dealing with cross-cultural environments<br>Trade-offs of competing risks in software projects | 3 |
| SP | Economies of Computing** | Effect of skilled labor supply & demand on quality of products<br>Impacts of outsourcing & off-shoring software development on employment<br>Consequences of globalization on the computing profession<br>Differences in access to computing resources and their effects | 6 |
| SP | Security Policies, Laws, Computer Crime | All | 3 |

*Overlaps many other topics – these instructional hours are dedicated to the topic of professional ethics

**Topics missing from these Knowledge Units are found in other courses in our curriculum, most notably our Software Engineering course and our Capstone Professional Practice Seminar.

**Other comments**

Many topics in this course can be integrated throughout the computing curriculum in a manner suggested by the cross-listings in CS2013 document. This integration can nicely complement a stand-alone course as described here.

At its core, this course is interdisciplinary, drawing content and pedagogy from computer science, philosophical ethics, sociology, psychology, law and other disciplines. There is great value in placing primary responsibility for this course on the computing faculty, who are recognized by students as content experts who know the computing field's potentials, limitations and realities. The primary instructor can be joined by faculty members from other disciplines in the delivery of the course.

# Languages and Compilers

## Department of Information and Computing Science, Utrecht University

**The Netherlands**

**Johan Jeuring**

**J.T.Jeuring@uu.nl**

**http://www.cs.uu.nl/wiki/bin/view/TC/**

**Knowledge Areas that contain topics and learning outcomes covered in the course**

| Knowledge Area | Total Hours of Coverage |
|---|---|
| Programming Languages | 22 |
| Algorithms and Complexity | 10 |

### Where does the course fit in your curriculum?

This is a second year elective course in the Computer Science programme of Utrecht University. It assumes students have taken courses on imperative and functional programming.

### What is covered in the course?

The intended learning objectives of this course are:

- To describe structures (i.e., "formulas") using grammars;
- To parse, i.e., to recognise (build) such structures in (from) a sequence of symbols;
- To analyse grammars to see whether or not specific properties hold;
- To compose components such as parsers, analysers, and code generators;
- To apply these techniques in the construction of all kinds of programs;
- To familiarise oneself with the concept of computability.

Topics:

- Context-free grammars and languages
- Concrete and abstract syntax
- Regular grammars, languages, and expressions
- Pumping lemmas
- Grammar transformations
- Parsing, parser design
- Parser combinators (top-down recursive descent parsing)
- LL parsing
- LR parsing
- Semantics: datatypes, (higher-order) folds and algebras

### What is the format of the course?

This course consists of 16 2-hour lectures, and equally many lab sessions, in which students work on both pen-and-paper exercises and programming lab exercises, implementing parsers and components of compilers.

### How are students assessed?

Students are assessed by means of a written test halfway (2 hours) and at the end of the course (3 hours), and by means of three programming assignments. Students should spend 32 hours on lectures, another 32 hours on attending lab sessions, 75 hours on reading lecture notes and other material, and 75 hours on the lab exercises.

**Course textbooks and materials**

We have developed our own set of lecture notes for the course. We use Haskell to explain all concepts in the course, and the students use Haskell to implement the programming assignments. The lecture notes for LR parsing have not been fully developed, and for this we use a chapter from a book from Roland Backhouse.

**Why do you teach the course this way?**

This course is part of a series of courses, following a course on functional programming, and followed by a course on compiler construction and a course on advanced functional programming. Together these courses deal with programming concepts, languages and implementations. All these courses use Haskell as the main programming language, but several other programming languages or paradigms are used in the examples. For example, in the third lab exercise of this course, the students have to write a small compiler for compiling the imperative core of C# to a stack machine.

**Body of Knowledge coverage**

| KA | Knowledge Unit | Topics Covered | Hours |
|---|---|---|---|
| PL | Program representation | All | 1 |
| PL | Language Translation and Execution | Interpretation vs compilation, language translation pipeline | 2 |
| PL | Syntax analysis | All | 8 |
| PL | Compiler Semantic Analysis | Abstract syntax trees, scope and binding resolution, declarative specifications | 4 |
| PL | Code Generation | Very introductory | 1 |
| PL | Language Pragmatics | Some language design | 2 |
| AL | Basic Automata Computability and Complexity | Finite-state machines, regular expressions, context-free grammars. | 4 |
| AL | Advanced Automata Theory and Computability | Regular and context-free languages. DFA, NFA, but not PDA. Chomsky hierarchy, pumping lemmas. | 6 |
| PL | Functional Programming | Defining higher-order operations | 4 |

Many, but not all, of the topics of the Knowledge Units above not covered appear in our course on Compiler Construction.

# Professional Development Seminar


## Northwest Missouri State University
## Public University
**Carol Spradling**
**c_sprad@nwmissouri.edu**

**http://catpages.nwmissouri.edu/m/c_sprad/**


**Knowledge Areas that contain topics and learning outcomes covered in the course**

| **Knowledge Area** | **Total Hours of Coverage** |
|---|---|
| *Social Issues and Professional Practice (SP)* | 15 hours |

### Where does the course fit in your curriculum?

Professional Development Seminar, a required, three hour 200 level course for computer science majors, is taken in the fall of the sophomore year. The student population for this course is approximately 30 undergraduate computer science majors that are required to take this course for their major and 10 international graduate computer science students that elect to take this course.

### What is covered in the course?

While the course covers Social and Professional Practice topics such as social context, analytical tools, professional ethics, intellectual property, privacy and civil liberties, this exemplar will focus on professional communications.

The course provides opportunities for students to develop their professional communication skills. This exemplar includes examples of four Professional Communication outcomes:

- Write clear, concise, and accurate technical documents following well-defined standards for format and for including appropriate tables, figures, and references.
- Develop and deliver a good quality formal presentation.
- Plan interactions (e.g. virtual, face-to-face, shared documents) with others in which they are able to get their point across, and are also able to listen carefully and appreciate the points of others, even when they disagree, and are able to convey to others that they have heard.
- Describe the strengths and weaknesses of various forms of communication (e.g. virtual, face-to-face, shared documents)

### What is the format of the course?

The course format is face-to-face weekly class meetings with some online threaded discussions which are used to augment face to face class discussions. Most class meetings include a short instructor lecture of no more than 10-15 minutes followed by small group topic discussions, consisting of groups of no more than 4-5 students. Additionally, the course utilizes group discussions (face to face and online threaded discussion), a group research project, a group research presentation and a unit on preparing for professional interviews which includes a unit on technical resume preparation and technical and situational interview preparation. .

*Group Discussions*

Students are provided a current news article or issue that pertains to the class topic, asked to read the article before class and then discuss the merits of the article with their group members. Groups of no more than 4-5 students self-select a note taker and spokesperson. The role of the note taker is to record the summary of the group discussion and submit the summary of their discussion by the end of the class period. The spokesperson provides a short summary of their group findings orally to the rest of the class and is provided the opportunity to communicate the group views with the entire class.

Students are also provided online opportunities to discuss topics using online threaded discussions. The instructor selects an article or case study that illustrates issues surrounding a particular topic, such as intellectual property or privacy. Students are asked to share their individual opinions supported by facts to agree either for or against their particular view. They are also required to respond to other student's threaded discussions and explain why they either agree or disagree with the other person's posts.

*Group Research Paper and Presentation*

A group of students work to select a research topic, write a group research paper and give a group presentation on the research topic. The group research paper must utilize peer reviewed references as well as follows APA formatting. The group research paper and presentation includes several individual and group milestones to encourage students to complete their paper in a timely manner. Students use group collaboration tools in the preparation their paper.

Student groups give their group research presentation twice during the semester. The first presentation is videotaped and posted online. Students are asked to view their portion of the presentation and write a short paper critiquing their portion of the presentation. Student groups then carry out their final class presentation and are graded on their group presentation.

*Professional Interviews Preparation*

Students are asked to prepare a professional technical resume and prepare for a mock interview with a "real" industry employer. Students are instructed regarding how to write a professional resume that highlights their technical skills and relevant experiences. Three drafts of their resume are developed in progressive stages. During this process, students receive critiques on their resume from the instructor, the Career Services Office and an industry professional. Students are also required to write a cover letter and prepare a list of references.

Students are instructed on how to prepare for a technical and situational interview. Students participate in a class interview with another student. This practice interview with another student heightens their awareness of what may occur during a "real" interview. Students are also critiqued on their interview skills through a Mock Interview Day in which "real" industry professionals conduct a face to face interview and provide feedback on the student's resume and interview skills.

Students are also required to attend a Career Day to meet "real" employers. They are encouraged to set up interviews for summer internships or to make contacts for future internships or full-time employment. In short, students are encouraged to network with employers.

*Cross Cultural Skills*

Undergraduate and graduate international students work together in groups and are exposed different cultural viewpoints as well as approaches to problem solving.

**How are students assessed?**

Students receive course points for their professional communication through group work participation, their research paper and presentation and their technical resume development and interview practice and preparation.

Professional Communication Assessment is as follows:

| Topic | Percentage of Final Grade |
|---|---|
| Technical Resume Development | 10% |
| Technical Interview Development | 14% |
| Group Research Paper | 16% |
| Group Research Presentation | 16% |
| Discussion Threads | 7% |
| Class/Group Participation | 20% |
| Other Assignments | 17% |

**Course textbooks and materials**

A textbook is utilized but most materials for the group work, group research paper and presentation and the professional interview preparation is offered through hand-outs and oral instructions. Students are encouraged to use online library resources and online current articles to support their work.

**Why do you teach the course this way?**

Professional communication has been emphasized in this course since 2002. This approach has impacted our students' abilities to develop their written and oral communication skills, to learn to discuss social and professional issues with other students, and has enhanced student's ability to obtain internships. A side effect of this intense component of professional communication has allowed students to apply technical skills in a professional environment.

**Body of Knowledge coverage**

| KA | Knowledge Unit | Topics Covered | Hours |
|---|---|---|---|
| *SP* | *Professional Communication* | *All* | 15 |

The following outcomes are covered in the course under the Professional Communication area.

- Write clear, concise, and accurate technical documents following well-defined standards for format and for including appropriate tables, figures, and references.
- Develop and deliver a good quality formal presentation.
- Plan interactions (e.g. virtual, face-to-face, shared documents) with others in which they are able to get their point across, and are also able to listen carefully and appreciate the points of others, even when they disagree, and are able to convey to others that they have heard.
- Describe the strengths and weaknesses of various forms of communication (e.g. virtual, face-to-face, shared documents)

# CSE341: Programming Languages

## University of Washington, Seattle, WA

**Dan Grossman**
djg@cs.washington.edu

http://www.cs.washington.edu/homes/djg/teachingMaterials/spl/

**Knowledge Areas that contain topics and learning outcomes covered in the course**

| Knowledge Area | Total Hours of Coverage |
|---|---|
| Programming Languages (PL) | 32 |

**Where does the course fit in your curriculum?**

This course is aimed at second- and third-year students (after CS1 and CS2 but before advanced elective courses). It is no longer required, but it is recommended and the vast majority of students choose to take it.

A version of this course has also been taught as a MOOC on Coursera, first offered in January-March 2013.

**What is covered in the course?**

Successful course participants will:

- Internalize an accurate understanding of what functional and object-oriented programs mean
- Develop the skills necessary to learn new programming languages quickly
- Master specific language concepts such that they can recognize them in strange guises
- Learn to evaluate the power and elegance of programming languages and their constructs
- Attain reasonable proficiency in the ML, Racket, and Ruby languages and, as a by-product, become more proficient in languages they already know

Course topics:

- Syntax vs. semantics
- Basic ML programming: Pairs, lists, datatypes and pattern-matching, recursion
- Higher-order functions: Lexical scope, function closures, programming idioms
- Benefits of side-effect free programming
- Type inference
- Modules and abstract types
- Parametric polymorphism
- Subtyping
- Dynamically typed functional programming
- Static vs. dynamic typing
- Lazy evaluation: thunks, streams, memoization
- Implementing an interpreter
- Implementing function closures
- Dynamically typed object-oriented programming
- Inheritance and overriding
- Multiple inheritance vs. interfaces vs. mixins
- Object-oriented decomposition vs. procedural/functional decomposition

… a few more minor topics in the same basic space

**What is the format of the course?**

The University of Washington uses a quarter system: courses are 10 weeks long with 3 weekly lectures and 1 weekly recitation section (4 total contact hours / week, for approximately 36 total not counting exams).

**How are students assessed?**

Over 10 weeks, there are 7 programming assignments, 3 in Standard ML, 2 in Racket, and 2 in Ruby, each done individually. There is a midterm and a final -- all homeworks and exams are available at the URL above. A majority of students report spending 8-13 hours / week on the course.

**Course Textbooks and Materials**

Lecture notes (and/or mp4 videos) written by the instructor largely replace a textbook, though for additional resources, we recommend Elements of ML Programming by Ullman, the Racket User's Guide (available online), and Programming with Ruby by Thomas. The lecture notes and videos are available.

**Why do you teach the course this way?**

This course introduces students to many core programming-language topics and disabuses them of the notion that programming must look like Java, C, or Python. The emphasis on avoiding mutable variables and leveraging the elegance of higher-order functions is particularly important. By not serving as an introductory course, we can rely on students' knowledge of basic programming (conditionals, loops, arrays, objects, recursion, linked structures). Conversely, this is not an advanced course: the focus is on programming and precise definitions, but not theory, and we do not rely on much familiarity with data structures, algorithmic complexity, etc. Finally, we use three real programming languages to get students familiar with seeing similar ideas in various forms. Using more than three languages would require too much treatment of surface-level issues. Using fewer languages would probably be fine, but ML, Racket, and Ruby each serve their purposes very well. Moving the ML portion to OCaml or F# would work without problem. Haskell may also be tempting but the course materials very much assume eager evaluation.

**Body of Knowledge coverage**

| KA | Knowledge Unit | Topics Covered | Hours |
|---|---|---|---|
| PL | Object-Oriented Programming | All, with some topics re-enforced from CS1/CS2 (hour count is for just this course) | 4 |
| PL | Functional Programming | All | 10 |
| PL | Basic Type Systems | All | 5 |
| PL | Program Representation | All | 2 |
| PL | Language Translation and Execution | Only these topics are covered: interpretation vs. compilation, run-time representation of objects and first-class functions, implementation of recursion and tail calls. The other topics are covered in another required course. | 2 |
| PL | Advanced Programming Constructs | Only these topics are covered: Lazy evaluation and infinite streams, multiple inheritance, mixins, multimethods, macros, module systems, "eval". Exception handling and invaraints, pre/post-conditions are covered in another required course. | 6 |
| PL | Type Systems | Only these topics are covered (and at only a very cursory level): Type inference, Static overloading | 2 |
| PL | Language | Only this topic is covered: Eager vs. delayed evaluation | 1 |

| | | | |
|---|---|---|---|
| | Pragmatics | | |

**Additional topics**

Pattern-matching over algebraic data types

# Programming Languages and Techniques I

## University of Pennsylvania, Philadelphia PA

*Stephanie Weirich, Steve Zdancewic, and Benjamin C. Pierce*
*cis120@cis.upenn.edu*

http://www.seas.upenn.edu/~cis120/

**Knowledge Areas that contain topics and learning outcomes covered in the course**

| Knowledge Area | Total Hours of Coverage |
|---|---|
| PL Programming Languages | 24 |
| SDF Software Development Fundamentals | 13 |
| AL Algorithms and Complexity | 2 |
| DS Discrete Structures | 1 |
| HCI Human-Computer Interaction | 1 |

**Where does the course fit in your curriculum?**

Prerequisites: This is a second course though students with prior programming experience or an AP course often do not take the first course (CIS110, which covers fundamentals of computer programming in Java, with emphasis on applications in science and engineering).

Following courses: Discrete Math for CS students (CIS 160), Data structures (CIS 121), Intro to Computer Systems (CIS 240)

Requirements: The course is required for CIS and related majors, but optional for all other students. Enrollment is currently 160 students per term.

Student level: Most students are in their first or second year, but there are some non-CS exceptions

**What is covered in the course?**

- Programming Design and Testing
- Persistent Data Structures & Functional programming
- Trees & Recursion
- Mutable Data Structures (queues, arrays)
- First-class computation (objects, closures)
- Types, generics, subtyping
- Abstract types and encapsulation
- Functional, OO, and Event-driven programming

**What is the format of the course?**

Three 50-minute lectures per week + one 50 minute lab section (lead by undergraduate TAs)

**How are students assessed?**

Two midterm exams, plus a final exam

A large portion of the grade comes from 10 weekly-ish programming projects:

OCaml Finger Exercises
Computing Human Evolution
Modularity & Abstraction
n-Body Simulation
Mutable Collections
GUI implementation
Image Processing
Adventure Game
Spellcheck
Free-form Game

### Course textbooks and materials

*Programming languages:* OCaml and Java (in Eclipse), each for about half the semester
*Materials:* Lecture slides and instructor-provided course notes (~370 pages)

### Why do you teach the course this way?

The goal of CIS 120 is to introduce students (with some programming experience) to computer science by emphasizing *design* -- the process of turning informal specifications into running code. Students taking CIS120 learn how to design programs, including:

- test-driven development
- data types and data representation
- abstraction, interfaces, and modularity
- programming patterns (recursion, iteration, events, call-backs, collections, map-reduce, GUIs, ...)
- functional programming
- how and when to use mutable state
- inheritance and object-oriented programming

Beyond experience with program design, students who take the class should have increased independence of programming, a firm grasp of CS fundamental principles (recursion, abstraction, invariants, etc.), and fluency in core Java by the end of the semester.

The course was last revised Fall 2010 where we introduced OCaml into the first half of the semester.

The OCaml-then-Java approach has a number of benefits over a single-language course:

- It levels the playing field by presenting almost all the students with an unfamiliar language at the beginning.
- Since we use only a small part of OCaml, we can present enough about the language in a few minutes to dive directly into real programming problems in the very first class (instead of having to spend several class sessions at the beginning of the semester reviewing details of a larger language that will be familiar to many but not all of the students).
- OCaml itself is a functional programming language that encourages the use of immutable data structures; moreover its type system offers a rich vocabulary for describing different kinds of data.
- When we do come to reviewing aspects of Java in the second part of the course, the discussion is more interesting (than if we'd started with this at the beginning) because there is a point of comparison.

**Body of Knowledge coverage**

| KA | Knowledge Unit | Topics Covered | Hours |
|---|---|---|---|
| AL | Fundamental Data Structures and Algorithms | Simple numerical algorithms<br>Sequential and binary search algorithms<br>Binary search trees<br>Common operations on Binary Search Trees | 2* |
| DS | Graphs and Trees | Trees | (with*) |

| | | | |
|---|---|---|---|
| HCI | Programming Interactive Systems | Model-view controller<br>Event management and user interaction<br>Widget classes and libraries | (with**) |
| PL | Object-Oriented Programming | All Core topics (Tier 1 and Tier 2) | 7 |
| PL | Functional Programming | All Core topics (Tier 1 and Tier 2) | 7 |
| PL | Event-Driven and Reactive Programming | All Core topics (Tier 1 and Tier 2) | 4** |
| PL | Basic Type Systems | All Core topics except benefits of dynamic typing | 3 |
| PL | Language Translation and Execution | Run-time representation of core language constructs such as objects (method tables) and first-class functions (closures)<br>Run-time layout of memory: call-stack, heap, static data | 2 |
| PL | Advanced Programming Constructs | Exception Handling | 1 |
| SDF | Algorithms and Design | The concept and properties of algorithms (informal comparison of algorithm efficiency)<br>Iterative and Recursive traversal of data structure<br>Fundamental design concepts and principles (abstraction, program decomposition, encapsulation and information hiding, separation of behavior and implementation) | 3 |
| SDF | Fundamental Programming Concepts | All Core topics (as a review) | 1 |
| SDF | Fundamental Data Structures | All Core topics except priority queues | 5 |
| SDF | Development Methods | All Core topics except secure coding and contracts | 4 |

# SE-2890 Software Engineering Practices


## Milwaukee School of Engineering
**Walter Schilling**
schilling@msoe.edu


**Knowledge Areas that contain topics and learning outcomes covered in the course**

| Knowledge Area | Total Hours of Coverage |
|---|---|
| *Software Engineering (SE)* | *29* |

**Where does the course fit in your curriculum?**

Second-year course for computer engineers covering SE fundamentals.

Prerequisites: one year of Java software development including use and simple analysis of data structures. Students have also had two one-quarter courses in 8-bit microprocessor development with assembly language and C.

**What is covered in the course?**

Week 1 - Introduction to software engineering practices
Week 2 - Requirements and Use Cases
Week 3 - Software Reviews, Version Control, and Configuration Management
Week 4/5 - Design: Object domain analysis, associations, behavior
Week 6 - Design and Design Patterns
Week 7 - Java Review (almost a year since last use)
Week 8/9 - Code reviews and software testing
Week 10 - Applications to embedded systems

**What is the format of the course?**

One-quarter (10-week), two one-hour lectures and one two-hour closed (instructor directed) lab per week.

**How are students assessed?**

Midterm and final exams, two individual lab projects and on 8-week team development project.

**Course textbooks and materials**

Gary McGraw, Real Time UML: Third Edition
Advances in the UML for Real-Time Systems Bruce Powel Douglass, Addison- Wesley, 2004.

**Why do you teach the course this way?**

The major goal is to prepare computer engineering students (not SE majors) to work in a small team on a small project, and to gain an introduction to software engineering practices.

**Body of Knowledge coverage**

| KA | Knowledge Unit | Topics Covered | Hours |
|---|---|---|---|
| *SE* | *Software Processes* | | *4* |
| SE | Software Project Management | | 2 |
| SE | Tools and Environments | | 3 |
| SE | Requirements Engineering | | 6 |

| | | | |
|---|---|---|---|
| SE | Software Design | | 10 |
| SE | Software Verification & Validation | | 4 |

# Software Engineering Practices

## Embry Riddle Aeronautical University, Daytona Beach, Florida

**Salamah Salamah**
**salamahs@erau.edu**

**Knowledge Areas that contain topics and learning outcomes covered in the course**

| **Knowledge Area** | **Total Hours of Coverage** |
|---|---|
| Software Engineering | 42 |

### Where does the course fit in your curriculum?

This is a junior level course required for students majoring in software engineering, computer engineering, or computer science. The course is also required by those students seeking a minor in computer science.

The course has a an introductory to computer science course as a prerequisite.

The typical population of students in the course is between 30-35 students.

### What is covered in the course?

Typical outline of course topics includes:

- Introduction to Software Engineering
- Models of Software Process
- Project Planning and Organization
- Software Requirements and Specifications
- Software Design Techniques
- Software Quality Assurance
- Software Testing
- Software Tools and Environments

### What is the format of the course?

The course meets twice a week for two hours each day. The course is a mixture of lecture (about 1.5 hours a week) and group project work. The course is structured around the project development where the students are constantly producing artefacts related to software development life cycle.

### How are students assessed?

Students are assessed through multiple means. This includes

- Individual programming assignments (about 3 a semester)
- In class quizzes
- Homework assignments
- Two midterms
- Semester long team project

Students peer evaluation is also part of the assessment process.

### Course textbooks and materials

Watts Humphrey's Introduction to the Team Software Process is the primary book for the course, but this is also complemented with multiple reading assignments including journals and other book chapters.

### Why do you teach the course this way?

The course is taught as a mini capstone course. It has been taught this way for the last 7 years at least. Students' comments indicate that the course is challenging in the sense that it drives them away from the perceived notion that software engineering is mostly about programming. Course is only reviewed annually as part of the department assessment and accreditation process.

I believe teaching the course based on a semester project is the easiest way to force students to apply the concepts and get familiar with the artefacts associated with a typical software development process.

**Body of Knowledge coverage**

| KA[2] | Knowledge Unit | Topics Covered | Hours |
|---|---|---|---|
| SP | System Level Consideration | Relation of software engineering to Systems Engineering<br>Software systems' use in different domains<br>Outcome: Core-Tier1 # 1 | *1* |
| SP | Software Process Models | Waterfall model<br>Incremental model<br>Prototyping<br>V model<br>Agile methodology<br>Outcome: Core-Tier1 # 2<br>Outcome: Core-Tier1 # 3<br>Outcome: Core-Tier2 # 1<br>Outcome: Core-Tier2 # 2 | 2 |
| SP | Software Quality Concepts | Outcome: Elective # 1<br>Outcome: Elective # 4<br>Outcome: Elective # 6<br>Outcome: Elective # 7 | 4 |
| | | | |
| PM | Team Participation | Outcome: Core-Tier2 # 7<br>Outcome: Core-Tier2 # 8<br>Outcome: Core-Tier2 # 9<br>Outcome: Core-Tier2 # 11 | 2 |
| PM | Effort Estimation | Outcome: Core-Tier2 # 12 | 2 |
| PM | Team Management | Outcome: Elective # 2<br>Outcome: Elective # 4<br>Outcome: Elective # 5 | 1 |
| PM | Project Management | Outcome: Elective # 6<br>Outcome: Elective # 7 | 2 |
| | | | |
| RE | Fundamentals of software requirements elicitation and modelling | Outcome: Core-Tier1 # 1 | 1 |
| RE | Properties of requirements | Outcome: Core-Tier2 # 1 | 1 |

[2] Abbreviation of Knowledge areas is available in the table at the end of the document.

| | | | |
|---|---|---|---|
| RE | Software Requirement Elicitation | Outcome: Core-Tier2 # 2 | 1 |
| RE | Describing functional Requirements using use cases | Outcome: Core-Tier2 # 2 | 1 |
| RE | Non-Functional Requirements | Outcome: Core-Tier2 # 4 | 1 |
| RE | Requirements Specifications | Outcome: Elective # 1<br>Outcome: Elective # 2 | 2 |
| RE | Requirements validation | Outcome: Elective # 5 | 1 |
| RE | Requirements Tracing | Outcome: Elective # 5 | 1 |
| | | | |
| SD | Overview of Design Paradigms | Outcome: Core-Tier1 # 1 | 1 |
| SD | Systems Design Principles | Outcome: Core-Tier1 # 2<br>Outcome: Core-Tier1 # 3 | 1 |
| SD | Design Paradigms (OO analysis) | Outcome: Core-Tier2 # 1 | 1 |
| SD | Measurement and analysis of design qualities | Outcome: Elective # 3 | 1 |
| | | | |
| SC | Coding Standards | Outcome: Core-Tier2 # 4 | 2 |
| SC | Integration strategies | Outcome: Core-Tier2 # 5 | 1 |
| | | | |
| VV | V&V Concepts | Outcome: Core-Tier2 # 1 | 1 |
| VV | Inspections, Reviews and Audits | Outcome: Core-Tier2 # 3 | 3 |
| VV | Testing Fundamentals | Outcome: Core-Tier2 # 4<br>Outcome: Core-Tier2 # 5 | 2 |
| VV | Defect Tracking | Outcome: Core-Tier2 # 6 | 1 |
| VV | Static and Dynamic Testing | Outcome: Elective # 1 | 2 |
| VV | Test Driven Development | Test Driven Development Programming Assignment<br>No available outcome | 1 |
| | | | |
| SE | Characteristics of maintainable software | Lecture on software maintenance and the different types of maintenance<br>No available outcome | 1 |

| | | | |
|---|---|---|---|
| SE | Reengineering Systems | Lecture on reverse engineering<br>No available outcome | 1 |
| | | | |
| FM | Role of formal specifications in software development cycle | Outcome 1<br>Outcome 2<br>Outcome 3 | 2 |
| | | | |
| SR | None | | 0 |

**Additional topics**

Ethics

**Other comments**

**Knowledge Areas Abbreviations**

| **Knowledge Area** | **Acronym** |
|---|---|
| Software Process | SP |
| Software Project Management | PM |
| Tools and Environment | TE |
| Requirements Engineering | RE |
| Software Design | SD |
| Software Construction | SC |
| Software Validation and Verification | VV |
| Software Evolution | SE |
| Formal Methods | FM |
| Software Reliability | SR |

# Technology, Ethics, and Global Society (CSE 262)

**Miami University, Oxford, OH**
**Public research university**
**Bo Brinkman**
[Bo.Brinkman@muohio.edu](mailto:Bo.Brinkman@muohio.edu)

http://ethicsinacomputingculture.com

**Knowledge Areas that contain topics and learning outcomes covered in the course**

| Knowledge Area | Total Hours of Coverage |
|---|---|
| *Social Issues and Professional Practice (SP)* | *20* |
| *Human Computer Interaction (HCI)* | *10* |
| *Graphics and Visualization (GV)* | *5* |
| *Intelligent Systems (IS)* | *elective* |

**Where does the course fit in your curriculum?**

Open to students of any major, but required for all computer science and/or software engineering majors. The only pre-requisite is one semester of college writing/composition. Traditional humanities-style course based primarily on reading and reflecting before class, discussing during class, formal writing after class. Meets three hours per week for 15 weeks.

**What is covered in the course?**

Students that successfully complete the course will be able to:

1. Formulate and defend a position on an ethical question related to technology.
2. Describe the main ethical challenges currently posed by technology.
3. Describe the results of group discussion on ethical issues as a consensus position or mutually acceptable differences of opinion.
4. Analyse a proposed course of action in the context of various cultures, communities, and countries.
5. Demonstrate effective oral and written communication methods to explain a position on the social responsibilities of software developers and IT workers.

Course topics (Coverage is in lecture hours, out of a total of 45 lecture hours.)
All of the following (27 hours):

1. Moral theories and reasoning. Includes applying utilitarianism, deontological ethics, and virtue ethics. Discussion of relativism and religious ethics. 3 hours.
2. Professional ethics. Includes definitions of "profession," codes of ethics, and ACM-IEEE Software Engineering Code of Ethics and Professional Practice. 3 hours.
3. Privacy. Definitions of privacy, the role of computing in contemporary privacy dilemmas. 6 hours.
4. Intellectual and intangible property. Definitions of copyright, trademark, and patent, especially as they apply to computer applications and products. Fair use and other limitations to the rights of creators. Intangible property that is not "creative" in nature. 6 hours.
5. Trust, safety, and reliability. Causes of computer failure, case studies (including Therac-25). 3 hours.
6. Review and exams. 3 hours.
7. Public presentations of independent research projects. 3 hours.

Selection from the following, at instructor discretion (18 hours):

1. Effects of computing on society and personal identity. Social network analysis, Marshall McLuhan, bullying and trolling, crowd-sourced knowledge, cybernetics. 6 hours.
2. Democracy, freedom of speech, and computing. The First Amendment, protection of children, state censorship, corporate censorship, case studies. 6 hours.
3. Computing and vulnerable groups. Case studies of effects of computing on prisoners, the elderly, the young, racial and ethnic minorities, religious minorities, people with disabilities, people with chronic diseases, developing countries, and so on. 6 hours.
4. Autonomous and pervasive technologies. Cases related to data surveillance, moral responsibility for autonomous systems, robots, and systems that function with little human oversight. 6 hours.

### What is the format of the course?

Face to face, primarily based on discussion. 45 contact hours, 90 hours of out-of-class work. Students read the textbook outside of class, and in-class time is spent on applying ideas from the textbook to cases or problems.

### How are students assessed?

30% based on 4 formal papers, 20% for essay-based exams (one midterm and a final), 35% for class participation and informal writing assignments, 15% for a public presentation (usually in Pecha Kucha format).

### Course textbooks and materials

Readings selected from Brinkman and Sanders, Ethics in a Computing Culture, from Cengage Learning, 2012. Supplementary readings, videos, and so on selected from the "recommended readings" listings in the book.

### Why do you teach the course this way?

Many of the topics of this course are incredibly complicated, and do not have clear right or wrong answers. The course is designed to encourage students to reflect on the course material, develop their ideas through engagement with each other, and then document their thinking.

Many schools are successful with distributing SP topics throughout the curriculum, but we found that this made it very difficult to assess whether or not the material was actually delivered and mastered. By creating a required course, we ensure that every student gets the material. Opening the course to non-computing majors has significantly increased the diversity of the course's audience. This benefits the students of the course, because it allows the instructor to demonstrate and highlight ethical clashes that arise when people from different academic disciplines try to work together.

**Body of Knowledge coverage**

| KA | Knowledge Unit | Topics Covered | Hours |
|---|---|---|---|
| *SP* | *Social Context* | *All* | *Worked throughout course* |
| *SP* | *Analytical Tools* | *All* | *3* |
| *SP* | *Professional Ethics* | *All except for "role of professional in public policy" and "ergonomics and healthy computing environments"* | *3* |
| *SP* | *Intellectual Property* | *All except for "foundations of the open source movement"* | *6* |
| *SP* | *Privacy and Civil Liberties* | *All, though "Freedom of expression and its limitations" is at instructor discretion* | *6-7* |
| *SP* | *Professional Communication* | *These topics covered across the curriculum, particular in our required software engineering course and capstone experience* | *0* |

| | | | |
|---|---|---|---|
| *SP* | *Sustainability* | *All topics in elective material of course. Covered in about ¾ of offerings of the course.* | *0-2* |
| *SP* | *History* | *Not covered. This material is covered in an introductory course.* | *0* |
| *SP* | *Economies of Computing* | *Not covered, except for "effect of skilled labor supply and demand on the quality of computing products," "the phenomenon of outsourcing and off-shoring; impacts on employment and on economics," and "differences in access to computing resources and the possible effects thereof." These are elective topics, covered in about ¾ of offerings of the course.* | *0-2* |
| *SP* | *Security Policies, Laws and Computer Crimes* | *These topics are covered in our computer security course.* | *0* |
| *HCI* | *Human Factors and Security* | *Vulnerable groups and computing* | *5* |
| *HCI* | *Collaboration and Communication* | *Psychology and social psychology of computing* | *5* |
| *GV* | *Fundamental Concepts* | *Media theory and computing* | *5* |

**Additional topics**

Autonomous computing and its dangers – elective topic in the Intelligent Systems KA

# COMP 412: Topics in Compiler Construction

## Rice University, Houston, TX

**Keith Cooper**

**keith@rice.edu**

**http://www.clear.rice.edu/comp412**

**Knowledge Areas that contain topics and learning outcomes covered in the course**

| Knowledge Area | Total Hours of Coverage |
|---|---|
| PL — Programming Languages | 21 |
| AL — Algorithms and Complexity | 5 |

### Where does the course fit in your curriculum?

COMP 412 is an optional course. It is typically taken by fourth-year undergraduate students and first-year graduate students in either the Professional Masters' degree program or the PhD program. Some advanced third year students also take the course.

Enrollment ranges from 40 to 70 students.

### What is covered in the course?

Scanning, parsing, semantic elaboration, intermediate representation, implementation of the procedure as an abstraction, implementation of expressions, assignments, and control-flow constructs, brief overview of optimization, instruction selection, instruction scheduling, register allocation. (Full syllabus is posted on the website, listed above.)

### What is the format of the course?

The course operates as a face-to-face lecture course, with three contact hours per week. The course includes three significant programming assignments (a local register allocator, an LL(1) parser generator, and a local instruction scheduler). The course relies on an online discussion forum (Piazza) to provide assistance on the programming assignments.

### How are students assessed?

Students are assessed on their performance on three exams, spaced roughly five weeks apart, and on the code that they submit for the programming assignments. Exams count for 50% of the grade, with the other 50% derived from the programming assignments.

The programming assignments take students two to three weeks to complete.

### Course textbooks and materials

The course uses the textbook *Engineering a Compiler* by Cooper and Torczon. (The textbook was written from the course.) Full lecture notes are available online (see course web site).

Students may use any programming language, except Perl, in their programming assignments. Assignments are evaluated based on a combination of the written report and examination of the code.

### Why do you teach the course this way?

This course has evolved, in its topics, coverage, and programming exercises, over the last twenty years. Students generally consider the course to be challenging—both in terms of the number and breadth of the concepts presented and in terms of the issues raised in the programming assignments. We ask students to approximate the solutions to truly hard problems, such as instruction scheduling; the problems are designed to have a high ratio of thought to programming.

**Body of Knowledge coverage**

| KA | Knowledge Unit | Topics Covered | Hours |
|---|---|---|---|
| AL | Basic Automata Computability and Complexity | Finite state machines, regular expressions, and context-free grammars | 2 |
| AL | Advanced Automata Theory and Computability | DFA & NFAs (Thompson's construction, the subset construction, Hopcroft's DFA minimization algorithm, Brzozowski's DFA minimization algorithm), regular expressions, context-free grammars (designing CFGs and parsing them — recursive descent, LL(1), and LR(1) techniques | 3 |
| PL | Program Representation | Syntax trees, abstract syntax trees, linear forms (3 address code), plus lexically scoped symbol tables | 2 |
| PL | Language Translation & Execution | Interpretation, compilation, representations of procedures, methods, and objects, memory layout (stack, heap, static),<br>Automatic collection versus manual deallocation | 4 |
| PL | Syntax Analysis | Scanner construction<br>Parsing: top-down recursive descent parsers, LL(1) parsers and parser generators, LR(1) parsers and parser generators | 6 |
| PL | Compiler Semantic Analysis | Construction of intermediate representations, simple type checking, lexical scoping, binding, name resolutions; attribute grammar terms, evaluation techniques, and strengths and weaknesses | 3 |
| PL | Code Generation | How to implement specific programming language constructs, as well as algorithms for instruction selection (both tree pattern matching and peephole-based schemes),<br>Instruction scheduling, register allocation | 6 |

## Other comments

The undergraduate compiler course provides an important opportunity to address three of the expected characteristics of Computer Science graduates:

*Appreciation of the Interplay Between Theory and Practice:* The automation of scanners and parsers is one of the best examples of theory put into practical application. Ideas developed in the 1960s became commonplace tools in the 1970s. The same basic ideas and tools are (still) in widespread use in the 2010's.

At the same time, compilers routinely compute and use approximate solutions to intractable problems, such as instruction scheduling and register allocation which are NP-complete in almost any realistic formulation, or constant propagation which runs into computability issues in its more general forms. In theory classes, students learn to discern the difference between the tractable and the intractable; in a good compiler class, they learn to approximate the solution to these problems and to use the results of such approximations.

*System-level Perspective:* The compiler course enhances the students' understanding of systems in two quite different ways. As part of learning to implement procedure calls, students come to understand how an agreement on system-wide linkage conventions creates the necessary conditions for interoperability between application code and system code and for code written in different languages and compiled with different compilers. In many ways, separate compilation is the key feature that allows us to build large systems; the compiler course immerses students in the details of how compilation and linking work together to make large systems possible.

The second critical aspect of system design that the course exposes is the sometimes subtle relationship between events that occur at different times. For example, in a compiler, events occur at design time (we pick algorithms and strategies), at compiler build time (the parser generator constructs tables), at compile time (code is parsed, optimized, and new code is emitted), and runtime (activation records are created and destroyed, closures are built and executed, etc.). Experience shows that the distinction between these various times and the ways in which activities occurring at one time either enable or complicate activities at another time is one of the most difficult concepts in the course to convey.

*Project experience:* In this particular course, the project s are distinguished by their intellectual complexity rather than their implementation complexity. Other courses in our curriculum provide the students with experience in large-scale implementation and project management. In this course, the focus is on courses with a high ratio of thought time to coding time. In particular, the students solve abstracted versions of difficult problems: they write a register allocator for straight-line code; they build a program that parses grammars written in a modified Backus-Naur Form and generates LL(1) parse tables; and they write an instruction scheduler for straight-line code. Students work in their choice of programming language. They typically reuse significant amount of code between the labs.

# CS103: Mathematical Foundations of Computer Science
# and
# CS109: Probability Theory for Computer Scientists
## Stanford University
## Stanford, CA, USA

**Keith Schwarz and Mehran Sahami**
`{htiek, sahami}@cs.stanford.edu`

**Course URLs:**
`cs103.stanford.edu`
`cs109.stanford.edu`

Knowledge Areas that contain topics and learning outcomes covered in the course

| Knowledge Area | Total Hours of Coverage |
|---|---|
| DS-Discrete Structures | 30 |
| AL-Algorithms and Complexity | 6 |
| IS-Intelligent Systems | 2 |

**Where does the course fit in your curriculum?**

CS103 and CS109 make up the first two courses in the required introductory CS theory core at Stanford. The prerequisites for CS103 are CS2 and mathematical maturity (e.g., comfortable with algebra, but calculus is not a requirement). The prerequisites for CS109 are CS2, CS103, and calculus. However, calculus is only used for topics beyond the CS2013 Discrete Structures guidelines, such as working with continuous probability density functions. Approximately 400 students take each course each year. The majority of students taking the courses are Sophomores, although students at all levels (from freshman to graduate students) enroll in these courses.

**What is covered in the course?**

CS103 covers:

- Sets
- Functions and Relations
- Proof techniques (including direct, contradiction, diagonalization and induction)
- Graphs
- Logic (proposition and predicate)
- Finite Automata (DFAs, NFAs, PDAs)
- Regular and Context-Free Languages
- Turing Machines
- Complexity Classes (P, NP, Exp)
- NP-Completeness

CS109 covers:

- Counting
- Combinations and Permutations
- Probability (including conditional probability, independence, and conditional independence)
- Expectation and Variance
- Covariance and Correlation

- Discrete distributions (including Binomial, Negative Binomial, Poisson, and Hypergeometric)
- Continuous distributions (including Uniform, Normal, Exponential, and Beta)
- Limit/Concentration results (including Central Limit Theorem, Markov/Chebyshev bounds)
- Parameter estimation (including maximum likelihood and Bayesian estimation)
- Classification (including Naive Bayes Classifier and Logistic Regression)
- Simulation

### What is the format of the course?

Both CS103 and CS109 use a lecture format, but also include interactive class demonstrations. Each course meets three times per week for 75 minutes per class meeting. CS103 also offers an optional 75 minute discussion session. The courses each run for 10 weeks (Stanford is on the quarter system).

### How are students assessed?

CS103 currently requires nine problem sets (approximately one every week), with an expectation that students spend roughly 10 hours per week on the assignments. The problem sets are comprised of rigorous exercises (e.g., proofs, constructions, etc.) that cover the material from class during the just completed week.

CS109 currently requires five problem sets and one programming assignment (one assignment due every 1.5 weeks), with an expectation that students spend roughly 10 hours per week on the assignments. The problem sets present problems in probability (both applied and theoretical) with a bent toward applications in computer science. The programming assignment requires students to implement various probabilistic classification techniques, apply them to real data, and analyze the results.

### Course textbooks and materials

CS103 uses two texts (in addition to a number of instructor-written course notes):

1. Chapter One of Discrete Mathematics and Its Applications, by Kenneth Rosen. This chapter (not the whole text) covers mathematical logic.
2. Introduction to the Theory of Computation by Michael Sipser.

CS109 uses the text A First Course in Probability Theory by Sheldon Ross for the first two-thirds of the course. The last third of the course relies on an instructor-written set of notes/slides that cover parameter estimation and provide an introduction to machine learning. Those slides are available here: http://ai.stanford.edu/users/sahami/cs109/

### Why do you teach the course this way?

As the result of a department-wide curriculum revision, we created this two course sequence to capture the foundations we expected students to have in discrete math and probability with more advanced topics, such as automata, complexity, and machine learning. This obviated the need for later full course requirements in automata/complexity and an introduction to AI (from which search-based SAT solving and machine learning were thought to be the most critical aspects). Students do generally find these courses to be challenging.

**Body of Knowledge coverage**

| KA | Knowledge Unit | Topics Covered | Hours |
|---|---|---|---|
| DS | Proof Techniques | All | 7 |
| DS | Basic Logic | All | 6 |
| DS | Discrete Probability | All | 6 |
| AL | Basic Automata, Computability and Complexity | All | 6 |

| | | | |
|---|---|---|---|
| DS | Basics of Counting | All | 5 |
| DS | Sets, Relations, Functions | All | 4 |
| DS | Graphs and Trees | All Core-Tier1 | 2 |
| IS | Basic Machine Learning | All | 2 |

**Additional topics**

CS103 covers some elective material from:

- AL/Advanced Computational Complexity
- AL/Advanced Automata Theory and Computability

CS109 provides expanded coverage of probability, including:

- Continuous distributions (including Uniform, Normal, Exponential, and Beta)
- Covariance and Correlation
- Limit/Concentration results (including Central Limit Theorem, Markov/Chebyshev bounds)
- Parameter estimation (including maximum likelihood and Bayesian estimation)
- Simulation of probability distributions by computer

CS109 also includes some elective material from:

- IS/Reasoning Under Uncertainty
- IS/Advanced Machine Learning

**Other comments**

Both these courses lectures move quite rapidly. As a result, we often cover the full set of topics in one of the CS2013 Knowledge Units in less time than proscribed in the guidelines. The “Hours” column in the Body of Knowledge coverage table reflects the number of hours we spend in lecture covering those topics, not the number suggested in CS2013 (which is always greater than or equal to the number we report).