CSCE155N Introduction to Computer Science I (MATLAB) ONLINE

Class Syllabus
Spring 2018

Instructor

Leen-Kiat Soh, Professor
Office: Room 122E, Avery Hall  Tel: 402-472-6738  E-mail: lksoh@cse.unl.edu
Skype: profsoh
Homepage: On http://canvas.unl.edu
Office Hours: 9:30-11:00 AM MW and Open Door Policy

Teaching Assistants

Graduate Teaching Assistants:
- Zion Schell (E-mail: zschell@cse.unl.edu) (Lab)
- Venkata Sunkara (E-mail: vsunkara@cse.unl.edu)
- Yang Liu (E-mail: yang@cse.unl.edu)

Office Hours:
Zion Schell:  F 2:00-3:30 PM, 5:30-7:00 PM
Venkata Sunkara: WF 12:30 PM-2:00 PM
Yang Liu:  M 4:00-7:00 PM

Undergraduate TAs:
- Isaac Logsdon:  M 10:30 AM-12:00 noon
- Vinay Singh:  W 1:00-2:00 PM  R 1:00-3:00 PM
- Trevor Thomsen:  MW 11:30 AM – 1:00 PM
- Tucker Loosbrock:  M 3:30-5:00 PM  W 4:30-7:00 PM

Room:  Student Resource Center, Avery Hall 12

Lectures & Labs

<table>
<thead>
<tr>
<th>CLASS</th>
<th>TYPE</th>
<th>DAY</th>
<th>TIME</th>
<th>LOCATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSCE-155N-210</td>
<td>LEC</td>
<td>MW</td>
<td>08:00:AM-09:15:AM</td>
<td>AVH-115CITY</td>
</tr>
<tr>
<td>CSCE-155N-221</td>
<td>LAB</td>
<td>T</td>
<td>06:30:PM-07:45:PM</td>
<td>AVH-20CITY</td>
</tr>
<tr>
<td>CSCE-155N-222</td>
<td>LAB</td>
<td>T</td>
<td>08:00:AM-09:15:AM</td>
<td>AVH-20CITY</td>
</tr>
<tr>
<td>CSCE-155N-223</td>
<td>LAB</td>
<td>T</td>
<td>05:00:PM-06:15:PM</td>
<td>AVH-20CITY</td>
</tr>
<tr>
<td>CSCE-155N-224</td>
<td>LAB</td>
<td>T</td>
<td>11:00:AM-12:15:PM</td>
<td>AVH-20CITY</td>
</tr>
<tr>
<td>CSCE-155N-225</td>
<td>LAB</td>
<td>T</td>
<td>03:30:PM-04:45:PM</td>
<td>AVH-20CITY</td>
</tr>
<tr>
<td>CSCE-155N-700</td>
<td>LEC</td>
<td>TBA</td>
<td>12:00:AM-12:00:AM</td>
<td>ONLINEDISTANCE</td>
</tr>
</tbody>
</table>
Computational Thinking

What is computational thinking? Jeannette Wing’s article in a 2006 issue of the Communications of the ACM has a definition. AP Computer Science Principles has one. ACM/CSTA K-12 CS Standards has one. Google’s Exploring Computational Thinking site has one. I see computational thinking as a way of thinking for logically and methodically solving problems that is purposeful, describable, and replicable. It encompasses skills or abilities such as problem decomposition, pattern recognition, abstraction, generalization, algorithmic design, and evaluation. These are all fundamental to problem solving.

Is computational thinking then something to do with computer science (CS)? Yes. But is it exclusive to computer science? No! It underlies many (if not all) disciplines. Now, how exactly is computational thinking then related to CS? I see that CS as a discipline that allows us to practice computational thinking more effectively, more efficiently, more reliably. Computational thinking manifests in CS largely due to CS’s emphasis in algorithms and their properties, such as complexity and extensibility. Also, in CS, practicing or applying computational thinking often involves considering digital artifacts—numbers, texts, sounds, videos, pictures, etc.—and producing information that leads to knowledge or decisions, or other digital artifacts.

Catalog Listing

Recommended for students interested in numerical and graphical applications in engineering and science, such as applied physics, working with time-sequence data, and matrix applications. Introduction to problem solving with computers. Topics include problem solving methods, software development principles, computer programming, and computing in society.

Course Objectives

1. Mastery of the fundamentals of programming in a high-level language, including data types and rudimentary data structures, control flow, repetition, selection, input/output, and procedures and functions.
2. Familiarity with problem solving methods, including computational thinking, problem analysis, requirements and specifications, design, decomposition and step-wise refinement, and algorithm development (including recursion).
3. Familiarity with software development principles and practices, including data and operation abstraction, encapsulation, modularity, reuse, prototyping, iterative development, exception handling, documentation, coding conventions, and testing.
4. Exposure to computing topics, including algorithms for searching and other problems, graphical user interfaces, event-driven programming, and database access.
5. Exposure to the history of computing.

Topics Covered

These topics are not purely sequential nor fully separate. Earlier topics continue to be reinforced and amplified in the context of other topics.
1. **Problem-solving methods**, including computational thinking skills, problem analysis, requirements and specifications, design (including top-down, bottom-up, object-oriented, and case-based design), decomposition and step-wise refinement, algorithm development.

2. **Software development principles and practices**, including use of an integrated development environment (IDE), tracing and debugging, data and operation abstraction, encapsulation, modularity, reuse, object-oriented programming, prototyping, iterative development, exception handling, documentation, coding conventions, and testing.

3. **Data constructs**, including data types, constants, identifiers, variables, assignment, operators, expressions, strings, input/output, arrays, abstract data types, and object properties.

4. **Control structures**, including selection, repetition, and exception handling.

5. **Modular elements**, including procedures, functions, variable scope, parameters, parameter passing, and object methods.

6. **Algorithms**, including searching (sequential and binary), recursion, case studies.

7. **Graphical user interface (GUI) programming**, GUI APIs, GUI event-driven programming.

8. **Programming database** access.

9. **History of computing**.

---

**Prerequisites**

1. Mastery of basic mathematical problem solving as demonstrated by satisfactory completion of mathematics through college algebra, trigonometry, and pre-calculus. (MATH 103 or equivalent).

2. Familiarity with the use of computers and software applications.

---

**Text Book**

*MATLAB: A Practical Introduction to Programming and Problem Solving*

---

**Homework Assignments (Individual & Group)**

There will be about 5-6 Homework Assignments to help you understand and master the basic concepts and train you on problem solving—to ultimately be proficient in developing software programs. The MATLAB programming language must be used for all the assignments. The assignments are due at class time on the indicated dates. Grading and hand-in requirements will be provided at a later time during the semester.

Homework assignments in the beginning of the semester will be individual and in smaller scale. After the first few assignments, group leaders will be selected based on student performance in class (exams and all assignments), and groups will be formed accordingly. After that, larger and more complex and open-ended homework assignments will be group-based.

---

**Laboratory Assignments (Randomized Pair)**

The laboratories are designed to supplement the lectures and provide hands-on experiences on topics that need additional attention. It is a significant part of the course and you are strongly encouraged to fully take advantage of this opportunity. You are required to read the lab handouts and the reading assignments before coming to the lab. There are about 12 laboratories.

---

**Examinations (Individual)**
We will have two mid-term exams and a final exam. The tentative dates for the exams are listed in the schedule. There will be no makeup tests.

<table>
<thead>
<tr>
<th>Final Grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>Your final course grade is based on the following: (1) programming projects (35%), (2) laboratory assignments (15%), (4) midterms (30%), and (5) final exam (20%). Final grades in this class will be assigned based on the following scale.</td>
</tr>
<tr>
<td>A+: ≥ 97  A: ≥ 93  A-: ≥ 90  B+: ≥ 87  B: ≥ 83  B-: ≥ 80</td>
</tr>
<tr>
<td>C+: ≥ 77  C: ≥ 73  C-: ≥ 70  D+: ≥ 67  D: ≥ 63  D-: ≥ 60</td>
</tr>
<tr>
<td>F: &lt; 60</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Disabilities</th>
</tr>
</thead>
<tbody>
<tr>
<td>Students with disabilities are encouraged to contact Christy Horn for a confidential discussion of their individual needs for academic accommodation. It is the policy of the University of Nebraska-Lincoln to provide flexible and individualized accommodation to students with documented disabilities that may affect their ability to fully participate in course activities or to meet course requirements. To receive accommodation services, students must be registered with the Services for Students with Disabilities (SSD) office, 132 Canfield Administration, 472-3787 voice or TTY.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Academic Misconduct</th>
</tr>
</thead>
<tbody>
<tr>
<td>Violations of academic integrity will result in automatic failure of the class and referral to the proper university officials. The work a student submits in a class is expected to be the student’s own work and must be work completed for that particular class and assignment. Students wishing to build on an old project or work on a similar topic in two classes must discuss this with both professors. Academic dishonesty includes: handling in another’s work or part of another’s work as your own, turning in one of your old papers for a current class, or turning in the same or similar paper for two different classes. Using notes or other study aids or otherwise obtaining another’s answers for an examination also represents a breach of academic integrity. Those who share their code and those who copy other’s code will be penalized in the same way; both parties will be considered to have plagiarized. Sanctions are applied whether the violation was intentional or not.</td>
</tr>
<tr>
<td>Academic dishonesty of any kind will be dealt with in a manner consistent with the CSE Department's Policy on Academic Integrity (<a href="http://cse.unl.edu/undergrads/academic_integrity.php">http://cse.unl.edu/undergrads/academic_integrity.php</a>). You are expected to know and abide by this policy.</td>
</tr>
<tr>
<td>To help avoid these problems, please start assignments early and seek help when you need it.</td>
</tr>
<tr>
<td><strong>PLAGIARISM OF ANY KIND IN THIS COURSE WILL RESULT IN A GRADE OF F.</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Relationship to ACE Learning Outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>This course is approved for ACE Student Learning Outcome 3, according to the following criteria:</td>
</tr>
<tr>
<td><strong>SLO3:</strong> Use mathematical, computational, statistical, or formal reasoning (including reasoning based on principles of logic) to solve problems, draw inferences, and determine reasonableness.</td>
</tr>
</tbody>
</table>
We will provide the following opportunities for learning the above ACE Learning Outcome. This course not only teaches students about how to design an algorithmic solution to solve a problem, but also teaches students about how to engineer the design into a working piece of program. Furthermore, the engineering process of implementing a program involves significant debugging, testing, and refining code. These activities teach and reinforces inferencing: a student has to be able to draw inference when diagnosing why a program crashes or does not compile or generate incorrect output; after making fixes, a student will have to re-evaluate the design to see if the outcome meets his or her expectation, and further draw inferences on how to proceed. Finally, an algorithm is fundamentally a logical sequence of steps that, given a set of input, generates definitely a set of output. The correct derivation of the output provides the decidability of the algorithm, which in turn determines reasonableness.

We will assess your achievement of the outcome through three primary tools, among others: exams, programming homework assignments, and structured laboratory assignments. Note that the artifacts of the programming homework assignments inherently embed the results of problem solving, inferencing, and reasonableness reasoning. This is because in order to produce a working program that compiles, runs, and computes the correct output, a student must devise an algorithmic solution and then implement it. For the structured laboratory assignments, worksheets—where students submit their findings from solving the problems given—are graded, and pre- and post-tests are given to students to test how they have learned.

Reinforced Skills

- Critical Thinking. Critical thinking is key in the development of algorithms and during the debugging process of implementing a program. The course provides numerous opportunities for critical thinking in lectures, programming assignments, and laboratories. The laboratories and assignments are problem-based and students are tasked to apply critical thinking to solve problems.

- Problem Solving. The development of algorithms and the implementation of programs are inherently problem solving. The course provides numerous opportunities for problem solving in programming assignment and laboratories. The laboratories and assignments are problem-based and students are tasked to solve problems.

Other Matters

The CSE Department has an anonymous suggestion box (http://cse.unl.edu/department/suggestion.php) that you may use to voice your concerns about any problems in the course or department if you do not wish to be identified.

It is CSE Department policy that all students in CSE courses are expected to regularly check their email so they do not miss important announcements.

Please make use of the Student Resource Center in Avery 13A: http://cse.unl.edu/src

Longer Version on Relationship with ACE Learning Outcome

The course presents many opportunities to learn computational and formal reasoning methodologies and skills to solve problems, draw inferences, and determine reasonableness.

Specifically, the lectures, together with the programming assignments and the weekly laboratory sessions, teach students both algorithms and the implementation of those algorithms to solve problems. That is, the course not only teaches students about how to design algorithmic solutions to solve problems, but how to engineer designs into working programs. This engineering process involves
significant debugging, testing, and refining code. These activities teach and reinforce inferencing: a student must draw inferences when diagnosing why a program crashes, does not compile, or generates incorrect output; after making fixes, a student must re-evaluate the design to see if the outcome meets expectations, and further draw inferences on how to proceed.

Finally, an algorithm is fundamentally a logical sequence of steps that, given a set of input, generates output. Specifications for the output determine reasonableness. Through algorithmic development, with top-down design, problem analysis and specification, step-wise refinement, and modularization, the students, when programming, are trained to determine the reasonableness of their solution. For example, students are trained to examine how their algorithms handle exceptions (which could terminate an algorithm prematurely if handled incorrectly), deal with boundary conditions (to prevent their programs from crashing), and prevent infinite loops (which could prevent reaching an outcome).

The course has approximately 45 hours of lectures each designed to explore concepts and paradigms that are central to the field of computer science. Students will master control flow, repetition, selection, input and output processes, and procedure and function design and invocation. Students will learn fundamental problem solving paradigms, including abstraction, encapsulation, exception handling, and event-driven programming. Through lectures, laboratory sessions, and programming assignments, students learn about problem analysis and specification, top-down design, algorithm development (including recursion), step-wise refinement, and modularization. Students will also be exposed to various algorithms, such as for searching and sorting.

The course has approximately 14 hours of laboratory sessions, each designed to train students to apply what they learn in the lectures to actual implementation and analysis of algorithms and software programs. Laboratory sessions require students to solve problems, to debug or revise programs, to analyze programs. These activities reinforce the students on problem solving, drawing inferences from their design and implementation, and determining reasonableness of a solution.

The course includes several programming assignments designed to help students learn about designing algorithmic solutions and the practice of implementing solutions as correct software programs, involving key steps such as program analysis, solution identification and evaluation, solution-to-algorithm mapping, initial feasibility analysis, coding, debugging, testing, and refinement. These steps provide ample opportunities for students to apply their computational and formal reasoning skills to solve problems, draw inferences, and determine reasonableness.