

My central goal when teaching is to improve the way students think about computer science. Regardless of course level, students should leave my classes with a more expert-like approach to answering questions and solving problems than when they entered. This goal forms the foundation for my teaching strategies and objectives.

Technical expertise is an essential part of computer science, but students are often overwhelmed by what appears to be a collection of disconnected facts, and they miss the coherent structure of knowledge that an expert sees. To combat this problem, I start with an overview of the material and with general problem-solving strategies before delving into specifics. This strategy reduces cognitive load on students and improves learning. My objective when teaching technical material is to cover the skills that students will use in the real world (e.g., logic, testing, and mental debugging). These skills apply not only to computer science but also to other disciplines: while I may not cover how to write a biology simulation in MATLAB, my students learn the skills needed to understand the program's general structure and how to test and to debug a simulation that isn't working correctly. An old-fashioned blackboard is my preferred visual aid to teach this material. When I hand-write source code, I tend to make mistakes similar to those of the students. When finished, I stop to review the solution and ask students questions about it. My review actively demonstrates the key practices of mental debugging and "expert thinking." Is my solution correct? How do I know? Does my implementation follow the strategy I outlined before writing code? My questions to students (e.g., I ask everyone to answer using an electronic student response system) give me instant feedback on how well the class understands the material. The questions force all students to think about the material and answering anonymously removes social barriers that might otherwise prevent their response. When a large number of students answer a question incorrectly, I stop to review the concepts and then give the students time to discuss the question and reach consensus regarding the answer. My approach builds students' confidence in their ability to solve bigger problems on their own as they realize they can answer small questions correctly.

Students learn better when they understand why the material is important. I motivate new material by starting with a question or problem that students cannot solve using their existing paradigms. When I was an undergraduate lab assistant, I introduced arrays by asking students how they would store the characters in a string. Their previous experience using single-valued variables quickly led them to realize that they needed a more sophisticated data structure—they essentially reinvented arrays with little more than an initial question from me. In larger classrooms, I'll split students into "geographical groups" of 3–5 students that work collaboratively to solve the initial problem or to discuss a prompt. I circulate among the groups to hear the discussion so I can gauge how students are thinking about the problem; after the discussion period, I call on several of the groups to present their solution to the entire class. Both of these strategies force students to engage with their peers in the learning process. When I explain material, I also incorporate analogies so students understand the practical application of general concepts to everyday situations. While a single explanation may not suffice for every student, different analogies and examples help all students to learn better.

I believe that more than technical expertise is necessary for success in computer science. Abstraction is an essential part of understanding real-world software systems. The design of software systems is critical because an elegant design is one of the best methods engineers have for dealing with complex problems. My teaching style—that is, starting at a high level and then moving on to smaller details and using analogies—illustrates how to use abstraction effectively. I also try to give students homework assignments with large, complex systems. For example, my undergraduate operating systems course required modifying the Windows kernel, a task that would not be possible without first understanding the existing abstractions. Experience with

applying abstractions to complex problems will help my students succeed in the workforce, but today's graduates also need good communication skills as developers increasingly work in large, multi-national teams. Effective communication becomes essential when no single developer can complete a complex implementation task alone. Communication is also important when interacting with non-technical audiences—for example, managers who are overseeing the work. I try to build good verbal communication skills via in-class conversations and through group projects that necessitate communication among team members; students practice written communication via free-response questions on homeworks and exams.

Research shows that learning assessments should be frequent to maximize long-term retention of material. I mix short- and long-term assignments with in-class assessments of students' learning. Programming activities in labs and homeworks focus on practical skills; homeworks also incorporate more conceptual questions to guide students toward a good implementation rather than the application of memorized "recipes" with no understanding of the higher-order concepts. My homework assignments reiterate the importance of good design by building upon each other. To keep the learning environment enjoyable, I have students implement games and complete challenges. For example, I use Sudoku to teach backtracking; students are encouraged to optimize their implementation with the fastest solver receiving extra credit. Competitions within the classroom such as keeping track of how many questions the left and right halves answer correctly can even make reviewing material an entertaining process. Exams bring together material from different parts of the course to ensure students synthesize the many concepts that we cover.

During both my undergraduate and my graduate education, I've served as a teaching assistant and also presented guest lectures on course material. My teaching experience extends from introductory to senior-level courses with enrollments ranging from a dozen to sixty students. I am passionate about the students in my classes and go the extra mile to improve their learning experience by holding extra office hours, helping them review for exams, or even coming back to work to help them meet a midnight deadline. I enjoy working closely with students, challenging them to learn new things and allowing them to challenge me to learn new things myself. I've found that success in computer science requires thinking deeply as well as logical problem solving, creative design, and practical application of the material I teach. Students from all backgrounds possess these qualities, and I hope to instill within them a love for learning that will continue throughout their life.