Teaching theoretical computer science courses requires keeping a close eye on students' cognitive load. Theory courses are particularly susceptible to confusing students on what the learning objectives are – are the students learning the theorem itself, the specific proof of the theorem, the proof techniques, or problem-solving skills? Often the answer is all of the above, but the wide variety of potential objectives adds stress to the student experience. How can they learn if it's not clear what they are supposed to focus on? It is not possible for an instructor to perfectly manage student focus – our classes are too large to manage the attention of each student, and besides we often are intentionally covering multiple objectives at once. However, by deeply understanding the mathematics, having students focus on what they need, and communicating the problem-solving process explicitly, I work to reduce unnecessary cognitive load on students.

Understand the mathematics

The first step for me in teaching theoretical CS courses is deeply understanding the mathematics in the course. A deep understanding improves my ability to communicate with students. As an example, consider finding closed forms of recurrences (to find the big-$\mathcal{O}$ running time of recursive code). At UW this skill is taught in our data structures courses. We have three different techniques: "unrolling" (i.e. simply plugging the recursive definition into itself until a pattern appears), the "tree method" (which uses a tree of recursive calls to organize the analysis), and applying "Master Theorem." When preparing for theoretical courses, I find significant benefit in deep preparation – from doing research in theoretical computer science, I bring extra domain knowledge that can improve my presentation. When I first was a TA for CSE 373 (our "data structures for non-majors" course) tree method and unrolling were both presented as ways of finding closed forms of recurrences. There was no extra context given as to why one would use one or the other, nor why we really needed to cover both. When I TAed again for the same instructor in Winter 2019, we were able to make improvements. A close examination of the two methods (and experience using both) will reveal that (at least on the kind of recurrences one runs into in data structures) the two methods are doing the same algebra, just in a different order. With that deeper knowledge, Kasey and I were able to change the lecture presentation: teaching them as two versions of the same idea, rather than as seemingly unrelated techniques. The new presentation gives students more context and reduced their confusion about the relationship in lecture.

Beyond altering the presentation, a deep understanding of the material allows for better decisions on what to cover and what to sweep under the rug. In our big-$\mathcal{O}$ analyses for 373 we always cover best- and worst-case. But worst-case analyses in data structures are frequently misleading. When dealing with array-based data structures, the worst case of `insert` methods is frequently $\mathcal{O}(n)$, but is constant time when the array doesn't have to resize. Similarly `quicksort` has a worst-case running time of $\mathcal{O}(n^2)$, but usually will run in time $\mathcal{O}(n \log n)$. To a theoretician, these statements are very different: the data structure analysis is amortized, while the `quicksort` statement is a high-probability statement about a random variable. To a practitioner, though, they will sound very similar; and in-practice, the distinction is not likely to make a difference for the population of CSE 373. Most resources (including older versions of our materials) would use different words for these two ideas; for Summer 2019, when I was the instructor, I decided to omit this distinction. We just called those running times "in-practice." This change let us skip really defining amortized and average running times (which are subtle, particularly for a group with widely varying math backgrounds). By avoiding the extra vocabulary, students can spend their mental energy where we want them to spend it: understanding the intuition rather than trying to understand a subtle definition.

Know the learning objectives

The second way I work to minimize student cognitive load is to understand exactly what our learning objectives are, and tailor our homework and exam questions to focus only on those objectives. For

courses with TAs, it is critical to get everyone on the same page about what is important. When I TAed Machine Learning, I led the creation of section materials, and would make the "plan" all TAs shared each week. I made sure our quiz section instructions for TAs included not just problems to go through, but also an explanation of what learning objectives we were working toward. Telling the TAs those objectives gave us the chance to make the section experience more uniform for students.

In CSE 373, when teaching the tree method, our learning objective is for students to be able to find the closed-form of a recurrence. Our objectives did not include students being able to modify the method or memorize it. Therefore, as we modified our materials, we ensured that we gave students scaffolding every time they needed to use the tree method, and that the scaffolding was always identical. While instructors and TAs can easily understand "What is the total work in the base case?" "What is the work at the base case level?" and "What is the work at the non-recursive level?" to all be asking the same question, students were confused by these changes in wording. Managing different vocabulary was not a goal for the course in the context of recurrences. We could reduce the cognitive load on the students by ensuring every place we mentioned tree method (lectures, sections, homework, and exams) we used identical scaffolding. After a multi-quarter effort to find the best wording and integrate it everywhere, we saw some positives. The consistent wording reduced confusion for students and it streamlined grading for TAs. My first time TAing we would frequently have to grade recurrence questions as a whole (despite breaking the question into subparts) because students would answer the "wrong question" in our scaffolding, but demonstrate understanding of the problem in their final answer. By Summer 2019, we were able to grade the questions essentially part-by-part, (since answering the "wrong question" was now quite rare). The change was not without its drawbacks – while students were making fewer mistakes interpreting our scaffolding, (anecdotally) the deeper understanding of what they are doing seemed to be missing. We did not find the perfect presentation to balance all of our learning objectives last quarter, but by knowing our objectives and continuing to iterate on how we present them, we can continue to make the course better for our students.

Communicate problem-solving explicitly

A third way to ease cognitive load is to make problem-solving an explicit part of class time. For CSE 373, the last few weeks of the course are dedicated to basic graph algorithms (BFS/DFS, spanning tree algorithms, and path-finding). In that portion of the course, the learning goal I care about the most is *graph modeling*: given a scenario, model that scenario as a graph and decide which algorithm to apply to find the answer you are looking for. To test that objective, we simply ask students to do it: we give them a scenario and ask for how they would define a graph and what algorithm they would run on it. The questions are extremely open-ended, so even with scaffolding they can be intimidating to students. Indeed, graph modeling is a hard skill to acquire, but of critical importance: Ph.D. students in CS have visited my office asking for help modeling their (research) problems. When I taught in Summer 2019, I introduced more emphasis on the problem solving process itself. We took more time to practice this skill in lecture, and when we did it was accompanied by an outline of how I think about the problems. My way of thinking won't work for everyone, but it gives students a starting point. Moreover, we added many more practice problems to give students a chance to practice the skill on their own. A skill like graph modeling is second nature to experts, which can cause them to think of it as "obvious" and not worth explaining. By taking the time to make problem-solving skills explicit in student interactions, the students are better able to learn those skills we care about.

Theoretical CS courses will always put significant cognitive load on students. By deeply understanding the mathematics, we can improve presentation of concepts and better identify whether mathematical complexity is actually necessary. By being very clear on what we do and do not consider a learning objective we can reduce cognitive load on students. By including our problem solving strategies explicitly, we can ensure that we are using that load as effectively as we can.