Teaching Statement: Raghav Malik

Philosophy

Having thoughtfully designed classes with careful scaffolding, opportunities for engagement, and many avenues of success is vital to getting the next generation of students excited about computer science and training them to become CS researchers. So, my role as an educator is not only to *teach students computer science*, but to *teach them how to learn computer science*. My priorities in the classroom are focused around this goal: knowing an answer matters less than knowing how to get to the answer; a deep understanding of the boundaries of current knowledge is a crucial prerequisite to doing original research that pushes these boundaries; and not every attempt at solving a problem will be successful, but effort should be rewarded.

These priorities are reflected in my approach to lesson design. I start by getting students familiar with the material at a basic level, usually by giving a lecture. Of course, merely listening to a lecture is not sufficient for getting a truly deep understanding of the material, so I like to augment my lectures with guided exploration in the form of activities. By giving students specific prompts to think about, I encourage them to deeply explore the material on their own and ensure that they retain a better understanding than they would from just staring at my slides. Finally, I firmly believe that learning should not occur only within the walls of the classroom: when designing lectures and activities, I try to also give students the tools for engaging with course material even after class, e.g., in the form of problem sets that encourage further exploration.

Experience

I have had many opportunities throughout my time at Purdue to engage with this teaching philosophy. When I was a teaching assistant for our Discrete Mathematics course in Spring 2023, I wrote a small proof assistant for verifying Gentzen-style propositional- and predicate-logic proofs, and I got to see firsthand the positive impact this had on students' learning. Encouraged by the near-immediate feedback they could now get for their homework proofs, some students began exploring more "creative" methods of proving the same statements. They found that, for example, in the absence of a constructive logic, one could "derive" the *modus ponens* rule using disjunctive syllogisms and the definition of implication. Anecdotally, the students who were experimenting with the proof assistant tended to demonstrate a better grasp of propositional and predicate logic, e.g. in office hours.

My duties teaching assistant duties expanded significantly in Fall 2024, when I started TA'ing for Object Oriented Programming in C++. Over the course of the next few semesters, I got to put my lecturing theories to the test as I designed and delivered several lectures on my own. Fortunately for me, for the first few lectures that I gave, the class (or at least the subset of the class that regularly attended lectures) was relatively small, so I was able to make my lectures highly interactive. I would lecture for a slide or two about a topic—for instance, "mutable" vs. "const" references—then pause to ask a few questions to facilitate a discussion ("How would the semantics of this program change if I passed by const or mutable reference instead of by value?"), encouraging students to talk about their responses with each other as well before continuing. Several students later confessed to me how much more engaging they found this form of lecture. Although these interactive lectures did not scale well to the larger class sizes in later semesters, we were able to replace the interactive portions with extra credit "activities", and found that students who regularly completed the activities consistently performed much better on exams.

Finally, in the Summer of 2024, I was given sole-instructorship over a Data Structures and Algorithms course and given free reign to simulate my teaching philosophy on a much larger scale. Since the course was taught virtually and the students were in different time zones, I decided to implement "attendance polls" in lieu of a proper attendance policy. After most lectures, I would post an open-ended question ("What is your favorite application of a topological sort?", "Why would you use Bellman-Ford instead of Dijkstra's algorithm?", etc.) and allow students to submit responses for 24 hours. These were meant to encourage actually engaging with the class material; a certain threshold of responses was required to pass and credit was awarded on the basis of effort rather than correctness. Although I was unable to provide individual feedback on every response to every poll, I would still read most of the responses to get a sense of what material students were still struggling with; I would then make sure to reinforce this in the next lecture before moving on.

The course was project-based, and while I inherited some programming assignments that had been given in past semesters, I decided to overhaul them to make them more open-ended as well. For example, one of the projects covered spatial indexing techniques. Rather than requiring them to implement, e.g., a k-d tree, the project description simply asked them to write an application that could efficiently find collisions between large sets of points. The students were then free to choose the data structure that they felt best fit the problem. Although many students did, in fact, end up just implementing k-d trees, there were still many more "creative" solutions. One student cleverly combined k-d trees and quadtrees to write a program that, while painful to debug, technically worked and certainly demonstrated a strong grasp of the material. Several others who had not yet internalized the difference between k-d trees and binary trees wrote code that always split their tree along the same axis; while this was a technique I had cautioned against in class, trying it for themselves definitely helped them better understand why long- and narrow-space partitions are nonideal.

When designing bi-weekly problem sets for the course, I sometimes included multipart problems that would incrementally build up to discussing something not covered in class. For example, one problem might start by asking students to analyze some simple hashing algorithms, then apply one of the hashing algorithms to a string-matching problem, and eventually "reinvent" the Rabin-Karp algorithm from first principles. Structuring the problem this way allowed me to reinforce course material (e.g., hashing algorithms) by requiring students to think creatively about applying the algorithms they learned about in class rather than rote-memorizing some formulas. It also provided a starting point for students who were interested in exploring a topic further than what we could cover in class.

Courses

At the undergraduate level, I can teach courses on data structures and algorithms, discrete mathematics, and programming languages and compilers. At the graduate level, I can also teach courses on programming languages and compilers, and cryptography. Additionally, I am interested in developing a course that surveys some more advanced compilation techniques, with a focus on building domain-specific compilers.