A Nine-week Afterschool School Curriculum for Introducing High School Students to the Principles of Computer Science

Nick Goel | CPSC 490 | Advisor: Jason Hirschhorn | Spring 2016

Motivation and Goals

The importance of computer science education for American high school students has been widely recognized. In addition to the value that a strong background in CS holds in today’s job market, the study of CS allows students more generally to develop skills in logical thinking and problem-solving that can be applied to a wide range of subjects. In spite of this, CS education in American high schools remains grossly underdeveloped. Recent estimates suggest that fewer than 3 in 10 high school students in the US attend a school that offers any formal coursework in computer science (1). And in only 29 states do computer science classes even count toward high school graduation requirements (1). Furthermore, glaring underrepresentation of women and racial minorities studying computer science has improved only slightly in the last decade. Only 18% of students taking AP Computer Science Exam in 2013 were women, 3% were black, and 6% were Hispanic (2). A very similar breakdown of demographics is observed both among students pursuing CS degrees in college and in the computer technology workforce (3, 4). Most research points to the importance of early exposure in students developing a lasting interest in the subject (1, 2, 5). Perceptions of CS as difficult, inaccessible, and boring among students not yet exposed perhaps pose the most significant barrier to greater student involvement (1, 5). Indeed, such negative perceptions are particularly prevalent among groups traditionally underrepresented in CS (5).

This project aims to address many of these shortcomings in CS education. Here, we present a nine-week introductory curriculum designed for high school students with no previous CS exposure in an afterschool setting. The principle objectives of the curriculum are as follows: to cover the broadest and most basic principles of CS, to give students the tools they need to write substantive programs in a widely used computer language, and to introduce students to the study of CS in a way that is engaging and accessible. The curriculum is not meant to cover everything we would expect to be taught in AP CS or an introductory college-level course. Nor
is it meant to be a one-off course for students hoping to gain some practical computing skills. The success of this curriculum should be measured by the number of students that it encourages to pursue the study of CS further—beginning with AP CS or an equivalent college-level course (like CPSC 112 at Yale).

**Curriculum Structure and Lesson Design**

The curriculum is composed of nine different weekly lessons, each of which is meant to last for two hours. The first two lessons are taught in Scratch, a visual programming language for beginners developed by the MIT Media Lab. These two lessons represent a soft introduction to programming, where students will explore many of the ideas encountered later in the curriculum within the easy-to-use context of Scratch. The subsequent five lessons—the core of the curriculum—are taught in Python, and each focuses on what broad topic in programming: data and variables, conditionals and Boolean data, loops, functions, and lists. The final two lessons are centered around a cumulative final project in Python: a search engine that scan the web for a list of search queries beginning at a seed web page. The final project ties together ideas from each of the five previous lessons.

Every lesson aside from the final two follows the same basic design. Lessons begin with “Standards” and “Goals.” The “Standards” section draws from the Computer Science Teaching Association (CSTA) revised K-12 standards and lists all of the particular standards that the lesson seeks to address. The “Goals” section involves more specific objectives for the lesson in a presented in a “students will be able to…” format that are assessed by homework and individual exercises later on. These are typically followed by a “Presentation” section which focuses on the core material to be covered in that week’s lesson. Associated with the “Presentation” is a handout for students called the “Reference Sheet” providing a detailed description of the material that students can refer to throughout the lesson. This is typically followed by “Class Exercises,” short code blocks or programming problems that students work on together with help from the instructor. Next are “Individual Exercises,” programming problems that students work on individually before solutions are reviewed by the instructor. Finally, each lesson concludes with three homework problems.
The remainder of these report will offer explanation and justification for most important design decisions that went into the curriculum, based on scholarly work regarding both general curriculum design and computer science education.

Starting in Scratch

Scratch is a visual “media-rich” programming language released by the MIT Media Lab in 2007 designed as an educational tool for beginners (7). Indeed, Scratch was originally motivated by the needs of teenage students in afterschool computer centers, a setting very similar to the one our curriculum is meant for (6). In Scratch students program with the mouse, as lines of code are represented by colorful, Lego-like puzzle pieces that snap into place when syntactically appropriate. Code then influences the behavior of characters, or “sprites,” that visually enact the program a student has written by moving across the “stage.” A growing body of research supports the idea that Scratch can be a highly effective tool for drawing students into the study of computer science (7-13). Most relevant is the work of Malan et. al., demonstrating the efficacy of Scratch as a first exposure to programming for students in a college-level introductory CS class (7). Malan utilizes Scratch in the first four lessons of his 24-lesson curriculum before moving on to a more widely used language like Java. Scratch’s advantage, Malan argues, it that its visual syntax is simple and immediately intuitive, while the syntax of almost any formal programming language can seem at first abstruse or intimidating to the unfamiliar student. Thus, Scratch allows students to “focus on problems of logic,” ie the fundamental principles of programming, instead of worrying about syntax. Furthermore, programming in Scratch maps to activities that are relevant to the interests of a young student—like telling a story or building a game—as opposed to solving math problems or printing out simple text on the screen (ie “Hello World”). This makes initial study of CS much more rewarding for students and helps break down negative perceptions of programming (discussed previously) that may discourage students. Malan found that 76% of students felt that Scratch was a “positive influence” on their experience, especially those with no prior programming background. Other studies have found that middle-schoolers can successfully learn important CS concepts using Scratch alone (8), that Scratch in introductory classes can improve the retention
of CS majors in college (9), and that Scratch can generate early interest in CS among traditionally underrepresented demographics (10, 11, 12).

In this curriculum, Lesson 1 is a basic introduction to Scratch involving many small game-like programming problems. In Lesson 2, students work to implement a game of their own design, with help from classmates and the instructor. This design for Lesson 2 was thoughtfully chosen. Firstly, Malan demonstrates how a similar assignment in his own course generated very positive results. Malan argues, “insofar as our goal was to excite students … we opted to entrust our goal to students’ own senses of curiosity and creativity rather than impose on the experience constraints of our own” (7). A separate study from Malony et. al. further illustrates the remarkable pedagogical value of self-exploration in Scratch (13). Maloney describes how students ages 8 to 18 at an afterschool computer center were offered no instructional intervention whatsoever and yet created projects in Scratch over an 18-month period that included concepts such as user interaction, loops, and conditional statements at a very high rate. These study shows how Scratch makes the fundamentals of CS extremely intuitive, and how the creative freedom that Scratch offers makes learning CS fun—so much so that students will quite literally teach themselves. We wanted to leverage these positive aspects of Scratch by allowing students to realize a project of their own design in Lesson 2, rather than an assigned project. Lesson 1 will have already provided students with the basic tools they will need, and within Lesson 2 help and guidance will be readily available—helping to guarantee that projects students choose are indeed instructive. Lesson 2 ideally will allow students to reach a deeply rewarding milestone in their study of CS (having built their very own game) before they move on to the much more challenging environment of traditional text-based programming language (Python). Like Malan, one of our primary goals is not just to teach, but to excite and motivate students. The self-directed design of Lesson 2 is critical to that goal.

I do, We do, You do

The core of our curriculum (Lessons 3-7) all follow a very similar design: a presentation from the instructor, followed by class exercises, individual exercises, and homework. This design is based on a popular and highly-touted model of teaching known as “I do, We do, You do.” This model, described in detail by both Lemov (14) and Cunningham (15), goes as follows: the instructor first demonstrates a key skill or concept before the class (“I do”), the instructor and
class then work together to solve problems and exercise this skill (“We do”), and finally students work on problems independently as the instructor evaluates their understanding (“You do”). With each step, increasingly more of the cognitive load associated with the skill being taught is transferred from instructor to student, until students have demonstrated a mastery of the skill on their own. In our lessons, the presentation represents the “I do” portion of the model, the class exercises represent the “We do” portion of the model, and individual exercises and homework represent “You do.” Some careful consideration is required to ensure the efficacy of this model.

First, we focus on the “I do.” As Lemov points out, providing students each with a paper record of the information they are expected to master is critical to their success (16). This policy makes clear to students what information they should focus on most and allows them to frequently and easily consult an “expert” in the form of their reference sheet. This idea informs our decision to produce a reference sheet covering the content of each lesson that is printed out and handed out to students. In the presentation, the instructor merely walks through the content of the reference sheet with the class, explaining concepts and fielding questions. As far as the specific design of each reference sheet, we emphasize instruction through demonstrative examples of code rather than lengthy textual explanations. As Victor argues, for an explanation to be effective it must be embedded in the context that it describes—i.e. actual Python code (17). Indeed, any experienced programmer could intuitively tell you that good programming, like good writing, is something that can only be learned by doing. As such, students are encouraged to create a new Python file during each presentation and to try out for themselves all of the code snippets shown in the reference sheet. This code can further serve as a handy reference as students work on individual and class exercises.

Now we discuss the “We do,” i.e. class exercises. The key here is to manage a careful, gradual shift in cognitive load from teacher to student. One way we achieve this shift is by breaking problems into many small questions and posing these questions to the class (18). This allows students to think through problems in chunks without being overwhelmed or discouraged by the difficulty of the problem as whole. One clear example of this approach in our curriculum is the “Three Questions” method for solving problems with while loops in Lesson 6. For such a method to succeed, it is critical that the entire class be engaged in the problem-solving process at all times, instead of just a few particularly vocal or motivated students. Toward this aim, we employ cold-calling—calling on students by name without asking for a show of hands.
Cunningham emphasizes the importance of cold-calling to enforce attentiveness in the entire class and to allow accurate assessment of each student’s understanding throughout the lesson (19). The last major feature of our class exercises is the “think-pair-share” method, another very popular and highly touted teaching technique. In “think-pair-share,” students are given a few minutes to write down their answers to a particular question, a few minutes to discuss their answers with their immediate neighbors, and a few minutes to share their answers with the rest of the class. As Cunningham explains, “think-pair-share” is such an effective method because even students who do not know how to solve a particular problem must engage with that problem nonetheless during discussion with their neighbor (20). Kothiyal et. al. apply this method specifically to a curriculum in introductory CS and show that levels of student engagement are measurably increased (21). In our curriculum, “think-pair-share” is usually applied to the interpretation of code blocks at the beginning of most lessons.

We now address the “You do”: individual exercises and homework. The goal of both these sections is to allow students to reinforce the skills they have learned thus far and to allow the instructor to effectively measure students’ mastery of the lesson's goals. Lemov emphasizes the importance of repetitive practice to help solidify skills taught earlier in a lesson: “A lesson should end with students getting at bat after at bat after at bat” (22). The volume of practice, or the number of repetitions, is key. Thus, almost every lesson in our curriculum ends with three practice problems in the form of individual exercises and three more in the form of homework. These practice problems are in addition to any class exercises or individual exercises that appear earlier in the lesson. Indeed, the significant majority of all class time is occupied by students working on problems in some way. Thus, students have ample opportunity to hammer down the programming concepts introduced in the Reference Sheets and gain a firm, lasting understanding of computer programming that will stay with them after the course is completed. Furthermore, these exercises allow the instructor to assess each student’s progress toward lesson goals from week to week. This is an idea borrowed Cunningham’s “exit passes” (23). The feedback students provide in the form of their completed homework and individual exercises allows an instructor to make improvements to a lesson for future classes and to know what topics should be prioritized given the opportunity to review later on.

Final Project
Lessons 8 and 9 are centered around a cumulative final project: a search engine written in Python. The program works by taking as input the address of a seed web page, a list of queries (ie strings we are searching for), and a maximum search depth (an integer). The program then executes a “breadth-first-search” following links in the seed page and terminating once it reaches its maximum depth. The program scans every web page it visits against the list of queries and returns a list of “hits,” an address of a web page visited that contains one of the queries. The idea of this for project and some of the code involved are adapted from the “Intro to Computer Science” course at Udacity (24).

The principle objectives of Lessons 8 and 9 are to force students to synthesize all of the programmatic concepts they have learned so far and to translate students’ efforts through the entirety of the course into a substantial and rewarding end product. The topic for this project was very carefully chosen. A search engine is something that all students will certainly be familiar with and, more importantly, something students will recognize as relevant to their own lives. This is in contrast to previous Python assignments that involved tasks like printing out lists of prime numbers or calculating factorials. While such assignments are essential to the mastery of programming fundamentals, they lack the same sense of accomplishment that comes with the completion of a program that has some obvious relevance to the real world. This sense of accomplishment ties in closely with the overall objective of the course: to motivate the further study of CS. A large body research indicates that the principle reason that students choose not to pursue CS is not that they find programming boring or useless, but that they do not believe they will succeed (25-28). Thus, we seek to motivate students by leaving them a piece of tangible evidence that they are highly capable programmers, in the form of a completed final project. While the project itself will certainly be challenging for students, Lessons 8 and 9 break the project into a manageable series of steps. With guidance from their instructor, any student who has met the objectives of the course’s previous lessons should be able to develop a working version of the search engine.

Conclusion

Here we described a nine-week curriculum for teaching high school students the basics of computer science in an after school setting. The main goal of this course is to inspire students to continue to study CS in the future, either at the AP or introductory collegiate level. This is not a
one-off course that, alone, will leave students with a practical skill set in computing. Nor is this a course that can immediately be incorporated into a traditional daily high school curriculum. Indeed, the decision to design a course specifically for the after school setting was a deliberate one. We feel that the after school setting is well-suited for introducing students to CS because it represents a low-pressure, low-risk environment where students may feel more comfortable trying out something completely new. That said, the after school setting also comes with considerable limitations. Attendance at after school classes is rarely mandatory and students will likely have other after school obligations competing for their time (such as a job or other family responsibilities). Thus, we cannot expect every student in the class to attend every lesson, and the course itself must be able to accommodate occasional student absences. Moreover, a student’s regular daily curriculum must necessarily take precedence over a supplementary after school curriculum. This limits the amount of time and effort that our course can demand from students, who are already dealing with a full course load. Thus, this course does not endeavor to cover every fundamental topic in CS that a dedicated student ought to know. A few of the big topics that we skip over include algorithms, object-oriented programming, and recursion. However, we do leave students with the tools they will need to build substantial and rewarding programs in Python (such as a working search engine) and to be exceptionally well-prepared for a more formal AP-level course.

Lastly, this curriculum cannot be considered a finished product until it is tested in the classroom. Like any programming project, it surely contains bugs that need to be ironed out and optimizations that ought to be made. Nonetheless, it represents an important tool for tackling the great deficit we face today in K-12 computer science education.
References:


16. Lemov 83


18. Lemov 88

19. Cunningham 162

20. Cunningham 149


22. Lemov 105

23. Cunningham 112


