

The Beauty and Joy of Computing

• Dan Garcia • Brian Harvey • Tiffany Barnes •

We last presented our Beauty and Joy of Computing (BJC) course in a special issue of ACM Inroads in June 2012 [13]. At the time, we taught BJC in two of the first five CS Principles national pilots at UC Berkeley and University of North Carolina, Charlotte.

Many things have changed since those early days, with more than two hundred high school teachers offered BJC professional development (PD) through four NSF grants, a transition to our blocks-based online software platform Snap! (based on Scratch) with cloud support [18], and a partnership with professional high school curriculum developers at EDC (Education Development Center), who are working with us to further refine our curriculum. Through partnerships with EDC, the New York City Department of Education, and CSNYC, our NSF-funded BJC4NYC project will bring BJC to 100 high school teachers in New York City, the largest and one of the most diverse school districts in the country. Finally, after two years of preparation, we launched our Massive Open Online Course (MOOC) BJCx via edX on Labor Day 2015 [4]. We are simultaneously offering it as an edX Small Private Online Course (SPOC), allowing high school teachers to use it as an e-book, complete with auto-grading and a class dashboard. More than sixteen thousand learners from all over the world signed up!



In this article, we share our philosophy, an update on our course design principles, a general flow through our curriculum, the impact BJC has had, and conclude with lessons learned.

PHILOSOPHY

Beauty and Joy of Computing (BJC) is a CS Principles course whose guiding philosophy is to meet students where they are, but not to leave them there. It covers the big ideas and computational thinking practices required in the AP CSP curriculum framework and powerful computer science ideas like recursion and higher-order functions. The programming part of the course uses Snap!, an easy-to-learn blocks-based programming language based on Scratch. Through the course, students learn to create beautiful images, and realize that *code itself* can be beautiful. Having fun is an explicit course goal. We take a “lab-centric” approach, and much of the learning occurs through guided programming labs that ask students to explore and play. When approaching a new topic, we use the maxim of “experience before formality”—we ask them to tinker around with a new idea (perhaps by exploring a new set of blocks in Snap!) before formally explaining how everything works. Formative assessments inform instruction through daily *For You To Do* activities in the lab that are shown to partners, groups, and the instructor. *Take It Further* activities in the labs (sometimes open-ended) and the opportunity for students to work at their own pace through the labs provide differentiated learning. We suggest that students use pair programming throughout the semester, with

partners that rotate regularly. We also encourage students to ask for, and give, help to one another—a powerful tool for bringing them together and developing a learning community.

We begin *every day* with a brief discussion of *Computing in the News*, connecting computing to students' everyday lives. When discussing the AP CS Principles *Global Impact* Big Idea, we try to balance optimism about technology with a critical stance toward any particular technology. For each new technology or innovation, we challenge students to think of its potential positive and negative impacts, its relationship to data, and how abstraction/algorithms/programming/ the internet are used. We also ask students to consider what biases or hidden agendas might have motivated an innovation. While the main textbook in the course is the outstanding book *Blown to Bits* [1], we supplement it with interesting web articles and videos.

BJC highlights creativity, collaboration, and communication through end-of-project celebrations, where students present their projects to the class, and then play-test each other's projects. Unlike competitions, this egalitarian model allows everyone to feel they have something unique to contribute to the group. Giving students the opportunity to demonstrate their projects in front of the whole school builds confidence and at the same time serves as an outstanding strategy for recruiting students the following year.

Course Design (Update)

We continue to refine our course every semester, driven by university and high school student and teacher feedback. We have the same course design principles we outlined in [13], with the following significant updates, in no particular order:

- Instead of the venerable Build Your Own Blocks (BYOB, based on Scratch), we now use Snap! as our programming environment (see Figure 1). Snap! is a full rewrite of BYOB in JavaScript by Jens Mönig, and the move to a browser-based programming environment has brought many benefits:

- The most obvious advantage of a browser-based development environment is that there is no download; one only needs to open a browser and go to the Snap! URL to get rolling. This helps teachers who do not have admin access to their Windows-based computers. (Another related challenge is that many schools block YouTube, so teachers must download video resources at home and bring them in.) However, programming in the browser introduces other challenges: not all browsers are equally compliant with the latest updates of JavaScript, not all computers have the latest versions of a particular browser, and some schools have bandwidth and/or connection issues. Fortunately, once Snap! is loaded successfully, (unless the user explicitly uses the http block or wants to store their projects online) it does not need an internet connection to run and can load/save files locally.
- Users can now use the MIOsoft-hosted cloud [18] to store their projects. This means students can work at school, save their project to the cloud, and pick it up later (home, at the library, a friend's house, etc.) without having to worry about copying it to (and not losing!) their USB flash drive.
- Once saved to the cloud, projects can be “shared” with others via an unencrypted URL that points to the project, and indicates the author and project name. Similar to a read-only link to a Google Doc, once another user opens the URL, a *copy* of the project is loaded into their browser—any changes do not affect the original. This feature is very useful for partners working on projects together (say, as part of the AP CS Principles *Create* performance task), since they can exchange projects as easily as they can send each other a link. It's also an invaluable feature for teachers who might be coding live in front of the class and wish to have the entire class catch up to where they are. The teacher would simply save the project to the cloud, click the “Share” button in the open or save dialog box (see Figure 2), and copy the resulting URL to a URL shortener, like *TinyURL*, *bitly*

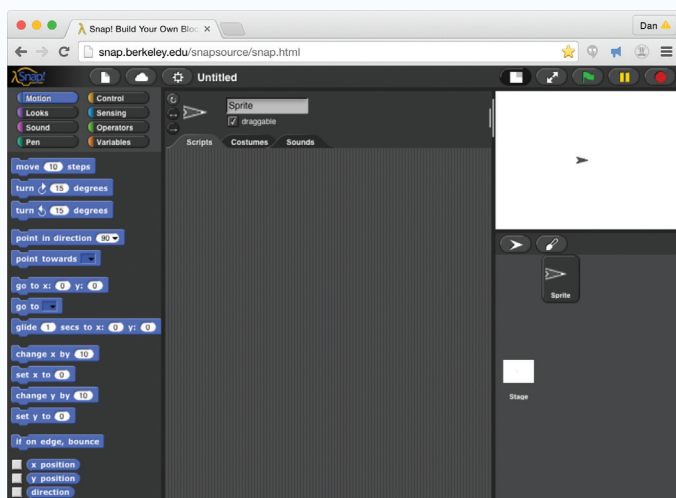


Figure 1: The browser-based Snap! programming environment, a rewrite of BYOB (Build Your Own Blocks, based on Scratch) in JavaScript.

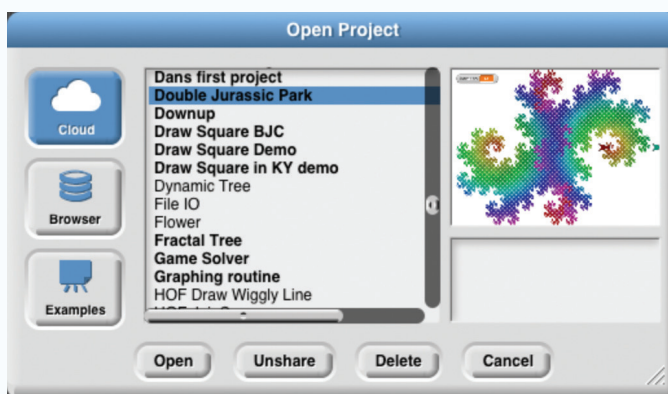



Figure 2: Choosing “Double Jurassic Park” in the Snap! “Open Project” dialog box. All shared projects appear in bold font, and users can easily make a project private again by choosing it and clicking the “Unshare” button.

or *ShoutKey*. We particularly like latter for spontaneously generated URLs because its shortened URLs (though temporary) are just a single English word appended to the end of “shoutkey.com/”, like “shoutkey.com/rice”—so the teacher need only say “Shoutkey rice” and all students know where to go [20]. Another possibility is a Google Chrome extension called “Tone” that introduces a small speaker icon near the URL field  that generates R2D2-like sounds out of your speakers for a few seconds when it is clicked, and all computers within “earshot” (with the extension installed in Chrome) are sent the link [14].

- The combination of the in-the-browser development environment, easily shared cloud-hosted projects, and the fact that all shared projects (by default) start in “play” mode (full-screen, “green flag”) clicked—a common way for projects to start—means that we have a recipe for easy mobile apps. Users just go their mobile device, open their mobile browser, and go to the Snap! project URL (hopefully shortened). Both iOS and Android allow users to save a URL into a virtual link with a square “app icon,” so these web-hosted projects look like native apps. We were able to recreate the Snap! version of the App Inventor tutorial “Whack a Mole” in 90 seconds! (It takes 60 seconds to build the code, seen in Figure 3, and 30 seconds to share it and load it onto a mobile device.)



Figure 3: The Snap! code for the “Whack Alonzo” mobile app, which can be written in 60 seconds, and loaded on a mobile device in only 30 more seconds.

- Snap! is written in JavaScript, and recently added a “JavaScript Function” block that lets (advanced) users write raw JavaScript but hide the details by wrapping it up and presenting it as a regular Snap! block. This is the beauty of abstraction! For example, Figure 4 shows how to add the ability to speak text with a one-line call to JavaScript.

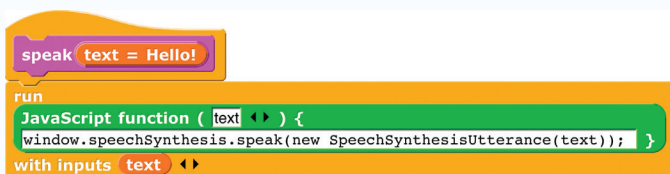



Figure 4: A speak command block can be added to Snap! by using the JavaScript function block.

- Snap! has an http block that allows the easy access to Internet APIs. Unfortunately, these APIs often speak in a cryptic JSON (JavaScript Object Notation) text-based format that can be difficult to parse. However, we provide  a block that can transform that JSON to a key-value pair dictionary / association-list (really just a Snap! list of lists), which can be queried. Figures 5 and 6 show how we call the “genderize” API that can tell (with data-driven probability) whether a particular name is probably a male or female name.

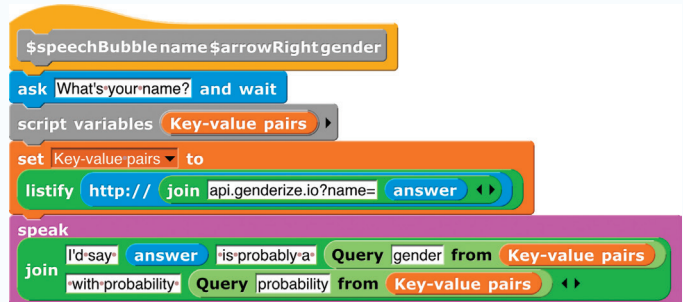






Figure 5: The definition of the  command block shows how easy it is to call an internet API, process the results, and use it in a project. If the user types “Dan” as a response to the “What’s your name?” prompt, the system responds: “I’d say Dan is a probably a male with probability 0.98.”  is a fairly complicated block written in JavaScript that transforms the plaintext JSON (JavaScript Object Notation) result from the API call to a key-value dictionary (aka association list, represented by a Snap! list of lists) that  knows how to traverse.



Figure 6: This demonstrates how you might call the  command block. All the details of the API call are hidden from the user.

- Snap! has four new key usability features that can be significant for some users. Users can resize the size of blocks and the stage, can find blocks using a keyboard command, and can input blocks via the keyboard (see Figure 7). When typing in the names of blocks, Snap! uses “context-based live search,” so as you type, the blocks (that make sense in that context) matching the letters you’ve entered so far instantly show up. Mouse-free coding is one small step toward supporting visually disabled users with our curriculum and software.

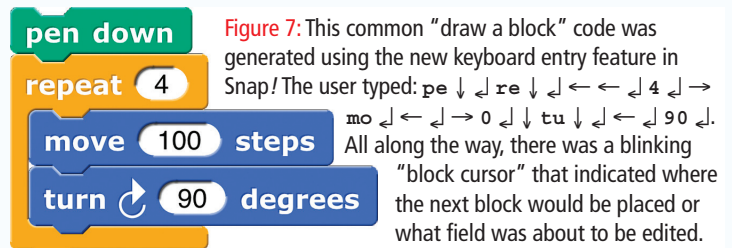














Figure 7: This common “draw a block” code was generated using the new keyboard entry feature in Snap! The user typed: `pe`  `re`  `l`  `l`  `4`  `mo`  `l`  `0`  `l`  `tu`  `l`  `90`  `l`. All along the way, there was a blinking “block cursor” that indicated where the next block would be placed or what field was about to be edited.

- Finally, in preparation for our edX MOOC, we added “auto feedback” to Snap! that allows a user to check whether their code for a particular coding challenge has the correct components (blocks, values, order), input/output properties (for reporter blocks), and/or side-effects (for drawing and interactive commands), and can provide helpful hints. Currently, this only exists in the edX course BJCx, but we hope to pull it out and integrate it with EDC’s curriculum next year.
- We removed Scratch from our first-weeks curriculum, and added Python “Beyond Blocks” lectures and labs at the end to help our local students’ impedance match with our first course for majors (taught in Python). These are not part of our EDC eight-unit high school curriculum, but are available as optional units. Instead, we begin with a Snap! “Hour of Code” activity that takes students through making the “Whack Alonzo” mobile app by the end of the first day.
- We have included a few Hour of Code introductory videos in the labs.
- We integrated the use of Higher-Order Functions (HOFs) into early labs (Unit 3) when students are learning about algorithms and lists. These include the functions map, keep, and combine—that students learn how to build at the end of the course in Unit 8.
- We promote the use of pair programming all year, instead of just for the projects.
- We moved the “Computing in Industry” and “Distributed Computing” material into the “optional units” category to allow more time for lecture, lab, discussion and unplugged activities to address the *Abstraction*, *Creativity*, *Internet*, and *Data* big ideas.
- Overall, labs have been redesigned to have more “experience before formality” and differential learning.

CURRICULUM OVERVIEW

At a very high level, the first quarter of the course (two units) takes students through the *Abstraction* and *Creativity* big ideas, gets them started with programming in Snap!, and ends with a fun programming project of the students’ choice completed in teams. This is so they get practice with collaboration, and feel a sense of accomplishment after the first quarter with a completed project they can share with their friends and family. The second quarter (two units) takes them through the algorithms and programming big ideas (in Snap! they are learning about how to store data in lists), as well as most of the remaining *Global Impact* big ideas. At the close of this quarter, they are ready for their *Explore Performance Task*, since they have been discussing computing innovations on a daily basis through our *Computing in the News* bell-ringer activities. The third quarter (two units) allows them to complete all remaining big ideas (*Data* and *Internet*). At the close of this unit, students have finished all required elements in the Curriculum Framework, complete the *Create Performance Task*, and prepare and take the AP CS Principles

Exam. The final quarter is wonderful “BJC secret sauce” material that students learn *after* the AP exam. Students will explore their creativity learning about fractals and recursive and higher-order functions. They will also hear outstanding lectures on computing research, including Artificial Intelligence and Human-Computer Interfaces. EDC will continue to host the most recent copies of the labs as they refine them over the next several years [10].

■ **Meta: Computing in the News (daily)**

- On a daily basis, in advance of class teachers pull computing news from ACM TechNews [2], the NY Times Technology section [19], the Electronic Frontier Foundation [11], and the Electronic Privacy Information Center [12]. These serve as launching points for bell-ringer discussions about computing innovations, how they interact with data and the CSP big ideas, and their social implications. Sometimes students lead whole-class conversations; other times teachers promote small-group discussions. The teacher improvises to allow for a topic that is engaging the entire class to continue. Big Ideas: *Data* and *Global Impact*

■ **Unit 1: Introduction to Computational Thinking**

- We engage the students immediately by showing some of the best final projects done by students the year before. We start with an introduction to computing with Snap! by building a mobile app with a partner. We designed the course so that students do all work in rotating pairs throughout the year. After that, students dive right into using Snap! by exploring its features and creating visual projects using important programming ideas such as loops, randomness, and building blocks. As much as possible, students experiment with Snap!, playing with input parameters, using blocks before they have been formally introduced, and modifying and extending the assigned tasks. Creativity and Abstraction are not only AP CS Principles Big Ideas, but also essential to students’ experience of the course, and we summarize them through discussions with the students. Students will also discuss social implications of computing, including issues such as cyber bullying, the prevalence of social media, and the ethics of illegally downloading music. Students are likely to be familiar with these issues, and grappling with them at the beginning of the course helps students understand the relevance and necessity of understanding the world of computing. Big Ideas: *Abstraction*, *Creativity*, *Programming*, and *Global Impact*

■ **Unit 2: Developing Complex Programs**

- Although the unit continues to introduce new programming concepts and Snap! features, the major programming focus is on structure and abstraction. Starting with the challenging task of teaching the computer to generate the plurals of nouns (e.g., butterfly → butterflies, moth → moths, bush → bushes), students begin thinking about the *structure of programs* and even, in a very preliminary way, the *structure of data*. They

learn about conditional statements, and think about when to use a sequence of conditionals, when to combine the conditions (Boolean values) first and have fewer conditional statements, and when to create “specialist” blocks, relegating lower level details to sub-procedures so that the main top-level block shows the structure of the logic un-camouflaged by the details. This structure allows the specialist blocks to be refined without requiring revision to the overall program. All of these are at the heart of Abstraction and are core to mathematical thinking as well as computer science. Students also begin to think about what makes correctly-working code “good” code—is it the brevity, the clarity, or some combination? (They do not yet encounter situations in which the speed-efficiency of the code can be a criterion.) This, too, helps them begin to attend to structure. Moreover, they begin to think about debugging by deliberately looking for ways to make a program fail, and then finding ways to avert failures. We also have engaging *CS Unplugged* activities that teach students how to convert numbers back and forth from a decimal representation to binary and hex representations. Big Ideas: *Abstraction, Programming, Algorithms, and Global Impact*

■ Fun Programming Project

- This is a chance, over two weeks, to allow students to work with a partner on a Snap! project of their choice. At the close of this activity, students present their projects to the class, and get a chance to play with each other’s projects.

■ Unit 3: Lists and Algorithms

- At this point, students are comfortable with Snap! but have found cases when they wanted to store aggregate information but didn’t know how—such as high scores for a game. We introduce lists (the only primitive data structure element in Snap!), and explain that there are two ways of thinking of them, as mutable or immutable. The latter follows the functional programming paradigm that underlies the course. We introduce some initial algorithms for working with lists, and continue with hierarchical lists (lists of lists used to draw points on the screen). We explore our first taste of functions-as-data as we experiment with map, keep and combine and use them to write an acronym generator. We end the programming activities with a program to detect magic squares. On the social implications side, we continue reading *Blown to Bits* [1], learning about and discussing the internet, search, encryption, and security. Big Ideas: *Abstraction, Programming, Algorithms, and Global Impact*

■ Unit 4: Algorithmic Complexity

- This is somewhat of a lighter-weight programming unit, but has some important and powerful computer science ideas. We continue with the discussion of algorithms and complexity, starting with a number finding activity. We look at timing activities to compare the difference

between constant, logarithmic, linear, quadratic, cubic and exponential running times. We discuss what kinds of problems are and are not computable, and show a failed attempt at solving the Halting Problem. On the social implications side, students read *Blown to Bits* chapters 6-8 [1] that discuss copyright, intellectual property, censorship, regulation, computing in war, and computing innovations. Big Ideas: *Abstraction, Programming, Algorithms, and Global Impact*

■ CSP Explore PT

- Students complete the Explore PT in eight classroom hours. The Explore PT has students channeling half a year’s worth of *Computing in the News* discussions. At the close of this activity, students share the research on their innovation with the class.

■ Unit 5: Data and Information

- In a rare case of alignment, this unit matches up perfectly with the CSP Big Idea of Data. As is usual for our take on topics, we use a programming lens. Therefore, rather than load data into Excel and massage it there, we pose problems whose solution requires students to write Snap! code to investigate. We start with text-based data: the case of Spam vs. Ham text messages, and ask them to consider writing a filter that could detect (and filter out) Spam. We then look at numerically-based data and the challenge of determining where a criminal is located who is “leaking” identifying GPS data from their tweets. We end with the first five of the ten teaching modules from the UC Berkeley *Teaching Privacy* group: *You’re Leaving Footprints, There’s No Anonymity, Information is Valuable, Someone Could Listen, and Sharing Releases Control* [21]. Big Ideas: *Data and Global Impact*.

■ Unit 6: The Internet

- Again, there is close alignment between our curriculum and a CSP Big Idea. This unit starts with the students reading the outstanding Appendix of *Blown to Bits* [1], and discussing it with the class. We have activities that expose and reconcile student misconceptions about how the internet works, supplemented by Code.org [6] and other popular illustrative videos. For programming activities, we have students explore internet APIs in Snap! and build a mini-project around them, as in Figure 5. We have a scavenger hunt activity involving internet tools *traceroute*, *whois*, *telnet*, *ping* and *Speedtest*. We end with the remaining five teaching modules from the UC Berkeley *Teaching Privacy* group: *Search is Improving, Online is Real, Identity Isn’t Guaranteed, You Can’t Escape, and Privacy Requires Work*. Big Ideas: *The Internet and Global Impact*.

■ CSP Create PT & AP CSP Exam

- The Create PT takes 12 classroom hours, with students in pairs for some of that time and alone for some of that time. At the close of this activity, students demo their projects to the class, and play with each other’s programs.

■ Unit 7: Trees and other Fractals

- Teaching experience shows us that students find recursive commands easier to understand than recursive reporters (functions that return values). Therefore, we split our exploration of recursion into two units, starting with the easier idea. Generating fractals is a vivid and engaging application. We start with the usual fractal tree, using the *combining method* of teaching recursion: Write a procedure, TREE1, that draws just the trunk of a tree, then write a procedure, TREE2, that draws simple branches by calling TREE1 twice. This is not a recursion, and since the students already wrote and debugged TREE1, there is nothing magical about it. We continue with TREE3, TREE4, and so on until the students complain that all these procedures are the same. We take them up on it by writing a recursive TREE with an extra LEVEL input replacing the numbers in the procedure names. (Since the first unit, students have seen many examples of generalizing a pattern by adding an input that captures the differences among similar examples.) However, it doesn't work, because there is no base case. The students have forgotten that TREE1 is different from the others. By correcting this, we reach a recursive solution that feels doable.
- The tree fractal provides plenty of opportunity to debug students' understanding by adding constraints: make the branches brown and the leaves green; make the trunk thicker; change the angles and lengths of branches randomly. (It's tricky to do that and still end up back at the bottom of the one tree trunk!) We then present several other fractals, with a lot of scaffolding in the early ones and progressively less in later ones. We do *not* use tail-recursive examples; we do not present recursion as a more complicated way to do iteration. If we can write code to do something iteratively, we should—or, even better, we should solve it using higher order functions! It's important to avoid tail recursion so that students aren't encouraged to develop the defective “go back” model of recursion, where they believe the report (i.e., return) statement means “jump back to the top of the block and start again.”

■ Unit 8: Recursive and Higher-Order Functions

- Recursive functions are both practically useful, especially in these days of massive parallelism, and of importance in theoretical computer science. They're also beautiful, once you understand them. What makes them harder for students than recursive commands is that the recursive call(s) aren't separate instructions, but rather have to be part of the (possibly quite complicated) compound sub-expression returned by the function. We start by addressing that difference explicitly, with simple examples using different combiner functions. Then we jump straight into branched recursion, to ensure that students create the correct conceptualizations of recursive calls. (Note that recursive functions are hardly ever truly tail

recursive, but if there is only one recursive call, it is easy for students to miss that subtlety.) Pascal's Triangle is a great example because it's a branched recursion if you simply implement the usual definition in terms of adding earlier numbers to get a later one, and yet there are (much) more efficient ways to solve the problem. We introduce the idea of *memoization* as a way to have our (simple definition) cake and eat it (efficiently) too.

- We reconnect with our earlier introduction to binary and hex notation, again taking the opportunity for students to write programs. Conversion between radices is a linear recursion, not a branched one, but it's too elegant to miss. We extend the algorithm to other bases, and in a Take It Further, we demonstrate the arbitrariness of representation by teaching about biquinary. (This also lets us sneak in a mention of the potential unreliability of hardware.)
- Progressively harder examples include mergesort (in which the simple recursive definition *helps* time efficiency, compared with simple iterative quadratic-time algorithms) and finding the subsets of a set (giving more practice with lists of lists). Then we return to simple examples, writing several recursive functions over lists that could more easily be done using MAP, then we do our usual trick of generalizing patterns by adding an input, and voilà, the students have written their first higher order function. They then quickly write KEEP and COMBINE. This is the climax and culmination of the course.

IMPACT

By the end of the summer 2015, we had offered professional development to 245 teachers, and worked with 20 master teachers who returned for a second summer PD and were ready to lead sessions on their own; four master teachers have already led sessions. We have had rather large, week long, face-to-face workshops (see Figure 8), as well as smaller, more intimate ones. Our PD format has evolved our model from 1-8-1 (one week face-to-face, eight weeks online, one week face-to-face) to 1-4-1.



Figure 8: The teachers who attended the first week of the 2014 BJC PD at UC Berkeley.

We have partnered with TEALS, a Microsoft initiative that partners software engineers with high school teachers to offer AP Computer Science A and “INTRO CS”—either a one-semester or full-year version of our BJC course [22]. We have welcomed TEALS teachers to our PD sessions, and have had dedicated one-day PD for engineers. In the 2014-15 school year, TEALS had 131 partner schools and 162 classes in those schools. Of those 162, 51 were semester-long and 16 were full-year long INTRO CS courses, both of which used the BJC curriculum.

In terms of directly connecting the course with students ourselves, we are consulting with the Level Playing Field Institute to offer BJC to their rising-junior SMASH scholars over the summer and through their fall academic-year program [16]—one model to reach students who do not have any computer science in their high school [17]. We worked feverishly to finish preparations to launch BJCx on Labor Day 2015 to 16K+ students, coming from 175 countries (the top three are US at ~32%, India at ~12%, and UK at ~4%) with self-reported ages from 10 to 70 (see Figure 9).

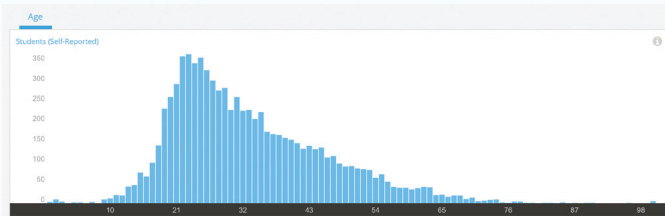


Figure 9: A histogram of the self-reported ages of more than 16,000 students enrolled in the first BJCx MOOClet, which launched Labor Day 2015.

Other universities, including UNC Charlotte, UNC Greensboro, Winona State, University of Oklahoma, NC State, and IUPUI, have taught BJC. At UC Berkeley, the BJC course has been transformational, bringing in record numbers of underrepresented students into the computer science major stream. We hit a crescendo in 2013, when BJC became the first UC Berkeley introductory computing class in recorded history to have more women than men in it [5]. Finally, we were humbled when both the College Board and Code.org endorsed BJC as a CS Principles course [7,8], and delighted to see components of our Creative-Commons BY-NC-SA-4.0 curriculum show up in the curricula of others in the CS10K family (e.g., Snap!, computing-in-the-news discussions, etc.) and among educators internationally. At the recent Scratch 2015 conference, we saw related initiatives from Korea, Spain, Germany, France and Austria [24]!

LESSONS LEARNED ABOUT OUR PROFESSIONAL DEVELOPMENT

- In our 1-N-1 model, we had only been paying teachers for their bookend face-to-face time. We believe teachers need stipends for the online part too, so they treat it as their full time job for the entire time. In a perfect world, we would sequester

the teachers away in an uninteresting locale with no nightlife, to allow time for daily homework and projects after dinner.

- In the intro week, we would have the teachers sample the curriculum as pair programming teams, model examples of class activities (e.g., *CS Unplugged*, *Computing in the News*, discussions, etc.), and have the teachers work in teams to complete and present a micro-project (see Figure 10).



Figure 10: NYC teachers enjoying a particularly amusing end-of-PD-week presentation.

- Initially, university professors led the PD. Now, two high school teachers, one senior and one junior, lead together. That has been great for scalability, but participants have said there are times when it's nice to have one of the university faculty leads in the room, to answer a tough question or provide an explanation why things (e.g., curriculum, software) were designed the way they were. Our leadership model is as follows (over consecutive summers, hopefully teaching BJC every year after the first summer): attend BJC PD as a regular teacher, attend a “master teacher” workshop that brings together veteran BJC teachers to share best practices, co-lead a BJC PD alongside a senior master teacher, and finally lead a BJC PD as the senior master teacher.
- As much as we have tried to coalesce them in one place, we still have no answer to the expanding variety of resources and websites necessary for summer PD: Snap! for programming, Wiki or Google Drive for sharing materials, Piazza *BJC TEACHERS* site for questions and answers, video-conferencing (Skype, Google Hangout) for online office hours, and the EDC site for the curriculum. If you add to that the main BJC website [3], our Facebook and Twitter pages, our CS10K community [9] area, and the edX course pages, it sometimes feels like a cacophony.

LESSONS LEARNED ABOUT OUR CURRICULUM

- The “How fast can N people return a shuffled deck to its sorted order?” (with varying values of N) is easily the most engaging *CS Unplugged* activity we have in the

curriculum (see Figure 11). Given the constraint that the unique sorted order needs to be announced in advance, the students themselves come up with parallel algorithms. The material requirements are minimal, and everyone is usually fully participating. The teacher can increase the value of N slowly, allow newly formed groups to determine their strategies, time them with a stopwatch, clicking the “lap” button whenever another group finishes, and plot the best times on a big graph to motivate a discussion of Amdahl’s Law. We normally use this later in the year to introduce our concurrency material, but our New York City teachers advocated that we should introduce it early (even on the first day), since it could serve as an icebreaker and team-building activity.



Figure 11: It’s hard to beat the “ N people sort a deck of cards” activity for engagement, team-building, student-driven learning and motivation for learning about algorithms, running time and concurrency. Here, NYC teachers are attempting to set the 13-person world record—they sorted them in 26 seconds!

- Learning about higher-order functions is easier if students first use them (and we now cover in Unit 3) and separately write them (now in Unit 8).
- Some people found the name of the guess-a-word game “Hangman” culturally insensitive, so we changed it to “Wheel of Fortune.” We typically launch it as a three-part problem: word guessing basics, full Wheel of Fortune, Evil Wheel of Fortune, the last one based off of Keith Schwartz’ Nifty Assignment [23].
- Teachers want more directed instruction and resources around building interactive projects (e.g., how to manage multiple sprites, how to scroll the background, how to test for sprite collisions). One way to work with multiple sprites in Snap! is to think of one sprite as the “director” and the others as “actors.” The director sends commands to the other sprites via the “tell” block (available when you “Import Tools”), as in Figure 12.

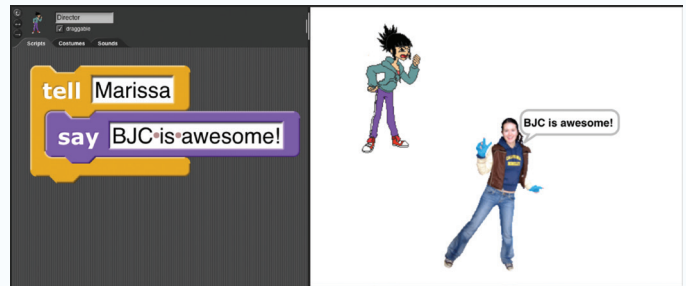


Figure 12: The Director sprite in the upper-left of the stage “tells” the Marissa sprite to say something. This “Director-Actors” pattern is extremely useful when working with multiple sprites in a simulation or game.

- A fertile area for us to grow is rule-based problems—with multiple sprites interacting, and each sprite following a very small set of rules. Like recursion, in which a few lines of code can reveal complex results, these projects sometimes yield quite surprising, emergent behavior. Some good examples are the simulated N -body physics problems, Conway’s Game of Life, and other continuous and discrete-space cellular automata [25].
- There is untapped potential in the sharing of resources and ideas with others leading CS10K efforts. Most of us awarded NSF CE21 funds to offer CSP teacher PD are wedded to our particular programming environment, but there are a host of other activities we employ to teach the non-programming aspects of CSP that we should be sharing. We have already incorporated several of Code.org’s videos, and are exploring the inclusion of some of their online activities, like the data compression and networking simulations.
- It is very time-consuming to transform a curriculum into an online course; many have said the lift is comparable to writing a book. Lots of “I”s to dot and “T”s to cross; it takes a village of folks to help, especially if there is a fair bit of development to be done (e.g., auto-grading). Leveraging top undergraduates can be quite effective but sometimes a mixed bag; they do amazing work, but they can also flake. It helps to have a solid core of lead students who do work but also manage other students. The most important two are the technical lead and the program manager, and we been fortunate to have terrific students in those roles.

CONCLUSION

Developing, teaching, and offering PD for BJC has been a six-year labor of love for us. It has allowed us to work with many incredible people—faculty and thought-leaders involved in CS10K, hundreds of high school teachers, and thousands of students. Many of these students have shared their poetry and artwork with us describing what the course has meant to them and how it has changed their lives (see Figure 13). We look forward to the coming years, which will bring even more polishing of the curriculum by the EDC team,

one hundred more teachers in New York City offering the course, and hopefully many more crowd-funded through Let's Teach CS [15]. All of that will allow us to continue to broaden participation, and as part of the larger CS10K effort, bring impactful, rigorous, and engaging introductory computer science to the world. **lr**



Figure 13: Artwork created from a UC Berkeley BJC student (who wished to go unnamed) to summarize what the course meant to her.

Acknowledgements

NSF grants 1143566, 1138596, 1443699 and 1441075, and a development grant from edX have supported this work. We owe a tremendous amount of thanks to the students and fellow instructors who have helped develop, refine and teach this course.

References

- [1] Abelson, Hal, Ken Ledeen, and Harry Lewis, *Blown to Bits: Your Life, Liberty, and Happiness After the Digital Explosion*. Addison-Wesley Professional, 2008.
- [2] ACM TechNews; <http://acm.org/technews>. Accessed 2015 September 16.
- [3] BJC website; <http://bjc.berkeley.edu>. Accessed 2015 September 16.
- [4] BJC on edX; https://www.edx.org/course?search_query=BJC. Accessed 2015 September 16.
- [5] Brown, K. "Tech shift: More women in computer science classes." *sfgate.com*. February 18, 2014; <http://www.sfgate.com/education/article/Revamped-computer-science-classes-attracting-more-5243026.php>. Accessed 2015 September 16.
- [6] Code.org website; <http://code.org>. Accessed 2015 September 16.
- [7] Code.org. "Expanding computer science through partnerships." *Anybody Can Learn* (blog), June 3, 2015; <http://blog.code.org/post/120601224166/3rdparty-partnerships>. Accessed 2015 September 16.

- [8] College Board. "College Board and NSF Expand Partnership to Bring Computer Science Classes to High Schools Across the U.S." June 4, 2015; <https://www.collegeboard.org/releases/2015/college-board-and-nsf-to-bring-computer-science-classes-to-high-schools>. Accessed 2015 September 16.
- [9] CS10K community website; <http://cs10kcommunity.org>. Accessed 2015 September 16.
- [10] EDC. "Beauty and Joy of Computing, 2015-2016;" <http://bjc.edc.org/>. Accessed 2015 September 16.
- [11] Electronic Frontier Foundation; <http://eff.org/>. Accessed 2015 September 16.
- [12] EPIC – Electronic Privacy Information Center; <http://epic.org/>. Accessed 2015 September 16.
- [13] Garcia, D. D.; Harvey, B; and Segars, L. 2012. CS principles pilot at University of California, Berkeley. *ACM Inroads* 3, 2 (June 2012), 58-60; DOI=<http://dx.doi.org/10.1145/2189835.2189853>
- [14] Google Tone; <http://googleresearch.blogspot.com/2015/05/tone-experimental-chrome-extension-for.html>. Accessed 2015 September 16.
- [15] Let's Teach CS; <https://www.facebook.com/LetsTeachCS>. Accessed 2015 September 16.
- [16] Level Playing Field Institute. "LPFI and UC Berkeley's "Beauty and Joy of Computing" Collaborate to Bring CS Principles to SMASH Scholars." (blog) September 1, 2015; <http://www.lpfi.org/cs-initiatives/cs-principles-course/>. Accessed 2015 September 16.
- [17] Level Playing Field Institute. "Path Not Found: Disparities in Computer Science Course Access in California High Schools." May 7, 2015; <http://www.lpfi.org/path-not-found-disparities-in-computer-science-course-access-in-california-high-schools/>. Accessed 2015 September 16.
- [18] Miosoft. "Snap! Programming for everyone." (blog) February 25, 2015; <https://blog.miosoft.com/2015/02/snap-programming-for-everyone/>. See also <http://snap.berkeley.edu/>. Accessed 2015 October 15.
- [19] New York Times: Technology; <http://www.nytimes.com/pages/technology/>. Accessed 2015 September 16.
- [20] Shoutkey; <http://shoutkey.com>. Accessed 2015 September 16.
- [21] Teaching Privacy Research Group. "Teacher's Portal;" <http://teachingprivacy.org/teachers-portal/>. Accessed 2015 September 16.
- [22] TEALS | Computer Science in Every High School; <http://www.tealsk12.org/about/>. Accessed 2015 September 16.
- [23] Schwarz, K. "Evil Hangman." Nifty Assignments. March, 2011; <http://nifty.stanford.edu/2011/schwarz-evil-hangman/>. Accessed 2015 September 16.
- [24] Scratch 2015 Program; <http://www.scratch2015ams.org/wp-content/uploads/2015/01/Scratch2015programa-web.pdf>. Accessed 2015 September 16.
- [25] Wolfram, S. *A New Kind of Science* Champaign IL, Wolfram Media Inc., 2002.

DAN GARCIA

777 Soda Hall #1776

UC Berkeley, Berkeley, California 94720-1776 USA
ddgarcia@cs.berkeley.edu

BRIAN HARVEY

784 Soda Hall #1776

UC Berkeley, Berkeley, California 94720-1776 USA
ddgarcia@cs.berkeley.edu

TIFFANY BARNES

Engineering Building III (EB3) 2401, Box 8206
 NCSU Campus, Raleigh, North Carolina 27695 USA
tmbarnes@ncsu.edu

DOI: 10.1145/2835184

© 2015 ACM 2153-2184/15/12 \$15.00