How My Students and I (Re)Discovered the Joy of Computing in CS2

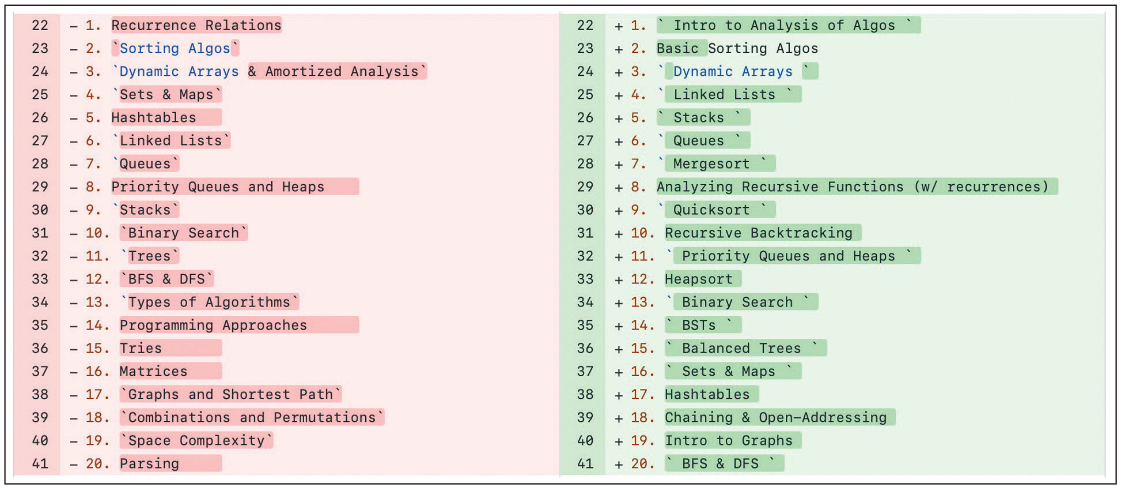
By Victoria C. Chávez, *Northwestern University*

Ask students what the hardest CS courses are, and they will likely mention data structures and algorithms (DSA) amongst the top three. In fact, DSA courses are included thrice in the current top 5 video results for "hardest CS classes" on YouTube. Students often bemoan about their nightmarish experiences and the high failure rates. How can we make CS2-DSA courses more joyful? How can we build on the incredible work being done in AP CS and CS0 courses through efforts such as "Passion, Beauty, Joy and Awe" (PBJA) of computing [4,5] and "Beauty and Joy of Computing" (BJC) [5]? In this piece, I reflect on my own experiences teaching a DSA course for the first and second time and highlight some of the ways my students and I were able to (re)-discover the joy of computing in CS2.

During Spring 2021 I had the opportunity to teach my department's CS2 course for the first time. At the time, I was working at a medium-sized public university and we were still remote due to the pandemic. Having heard so many of my students bemoan about their experiences taking and re-taking the course, I was committed to changing the course's bad rap and lowering the DFW (drop, fail, withdrawal) rate.

As I started course planning, I noticed a familiar issue: Course Creep. Course creep, a term I learned from Dr. John F. Hughes, is used to describe the phenomenon of courses having topics added in, often by various instructors across a course's lifetime, with few to no topics ever being removed. Instructors who have inherited courses may recognize its familiarity and how difficult it is to resolve. Case in point: for at least two decades we have struggled to establish some sort of "intellectual focus" for CS2 [8] but with the proliferation of computer science across academic institutions, this challenge has only grown greater. Scholars have called attention to the inconsistencies of CS2 courses and expectations across institutions [6,9] and even the varying CS curriculum definitions and recommendations over the years from ACM [10]. While I knew there was too much being covered in the course, as a new instructor to it, I struggled to identify what I could remove. I consulted some of the professors who would be teaching my students' subsequent courses and inquired about what they deemed essential pre-requisite knowledge for students to succeed in their courses. While they agreed there was too much trying to be covered in my class, when I asked what needed to stay versus what could be cut out, I was left with essentially just as many topics (see Figure 1). As foundational as all these topics are, trying to teach 20 distinct topics over the course of 10-15 weeks, especially if we expect students to understand them in-depth, is unreasonable.

While some of these topics should remain in a DSA course, many could be introduced earlier. For example, CS1 could have a formal introduction to arrays rather than only using them as a use-case for loops. CS0 or CS1 could dive into binary search as a concrete example of an algorithm when the term 'algorithm' is first introduced. Students can also be exposed to the use-cases of sets and maps in early programming projects. Essentially, topics that can be scaffolded from an understanding of arrays or lists could be integrated into CS1 as relevant applications of what students are learning. Similarly, if we introduce early ideas of efficiency in CS1, the formal mathematical introduction in CS2 will be more cohesive. To be clear, I am not proposing these be incorporated as new topics in another too-heavily packed course, rather, I am suggesting earlier exposure and more formal scaffolding across introductory courses. These topics can and should still have formal introductions in CS2, but prior use and exposure to them will make CS2 less daunting. In my own undergraduate years, I experienced how an integrated approach [3] that introduced programming alongside theoretical fundamentals—including DSA topics—helped reinforce my programming skills while reducing the cognitive load of trying to understand several ADTs (abstract data types) over two courses instead of just one. Relatedly, Porter et al. [9] have discussed the difference between 'basic' data structures and 'advanced' data structures, concluding that data structures



```
- 1. Recurrence Relations
- 2. `Sorting Algos`
- 3. `Dynamic Arrays & Amortized Analysis`
- 4. `Sets & Maps`
- 5. Hashtables
- 6. `Linked Lists`
- 7. `Queues`
- 8. Priority Queues and Heaps
- 9. `Stacks`
- 10. `Binary Search`
- 11. `Trees`
- 12. `BFS & DFS`
- 13. `Types of Algorithms`
- 14. Programming Approaches
- 15. Tries
- 16. Matrices
- 17. `Graphs and Shortest Path`
- 18. `Combinations and Permutations`
- 19. `Space Complexity`
- 20. Parsing
```

```
+ 1. ` Intro to Analysis of Algos `
+ 2. Basic Sorting Algos
+ 3. ` Dynamic Arrays `
+ 4. ` Linked Lists `
+ 5. ` Stacks `
+ 6. ` Queues `
+ 7. ` Mergesort `
+ 8. Analyzing Recursive Functions (w/ recurrences)
+ 9. ` Quicksort `
+ 10. Recursive Backtracking
+ 11. ` Priority Queues and Heaps `
+ 12. Heapsort
+ 13. ` Binary Search `
+ 14. ` BSTs `
+ 15. ` Balanced Trees `
+ 16. ` Sets & Maps `
+ 17. Hashtables
+ 18. Chaining & Open-Addressing
+ 19. Intro to Graphs
+ 20. ` BFS & DFS `
```

Figure 1: Before and after commit history of the course topic list

such as graphs, heaps, and others may not belong in a 'basic' data structures course.

Additionally, given that many departments require students to take a separate algorithms course, time-complexity need not be another major focus of CS2. If students will encounter the material in a subsequent required course, CS2 can omit proofs and formal notation, instead only introducing naive/informal/intuitive ways of analyzing data structures and algorithmic choices to give students more breathing room.

Of note, scaffolding topics across distinct courses only works if the professors of said courses communicate expectations (and realities) consistently. Simply setting course descriptions and objectives once a decade and letting courses move independent of each other is harmful to our students' learning. As students come in with different experiences[1], we should adjust our courses and ripple the effects onto subsequent courses. Doing so will not only prevent students from being met with the unpleasant surprise of being un- or under-prepared but will also ensure professors have realistic expectations of their incoming students' knowledge.

JOY BEYOND REDUCING OVERWHELMINGNESS

Unfortunately, attempting to resolve course creep alone was not enough to ensure my students had a joyful CS2 experience. While I felt that my first iteration of CS2 was still rushed and drab, overall student evaluations were positive. Students—especially those who had taken the class previously and had found it lacking—were appreciative of the flexibility of assignment deadlines and resubmissions [1], the sense of community despite/while being online, and the joy I tried to bring to the class amidst a raging pandemic we still knew too little about. I taught the course again over the summer and made some additional adjustments including the following.

- Removing exams
- Using a combination of soft and hard deadlines
- Making resubmissions more accessible, keeping resubmissions unlimited
- Updating the automated grading test suites to provide better feedback
- Removing extended (two-week) homework assignments and only assigning weekly labs
- Making assignment grading complete/incomplete
- Reorganizing topics

Of these changes, I would like to expand on the last one: topic reorganization. The ordering of topics as they are introduced to students is more important than we credit it. However, when it comes to CS2, given that there is not enough agreement about what the course is supposed to teach, topic ordering is a conversation the community has not yet broached. One of the few papers on DSA topic ordering [7] suggests how topic ordering has the potential to help or hinder students' understanding and differentiation of ADTs versus data structures. As I reorganized the topics of my course, I wanted to do so in a way that allowed students to have some 'early wins' while also better scaffolding more difficult topics (see Figure 2). While I leveraged my experience in computing and as an educator to scaffold topics and assignments, I centered students' experiences and feedback to determine topic difficulty.

For example, given that most students were familiar with arrays from CS1, I moved the topic to the beginning of the course and leveraged students' understanding of arrays to scaffold binary search and maps. Relatedly, since students seemed to enjoy and more quickly understand binary search and maps, those two topics were also placed much earlier in the term. On the other hand, while linked lists were a tremendous pain point for students, I placed them early enough in the course so that we could revisit the topic quite often, but after students had already gained some confidence working with arrays, sets, and maps. To reinforce how linked lists work, students completed linked-

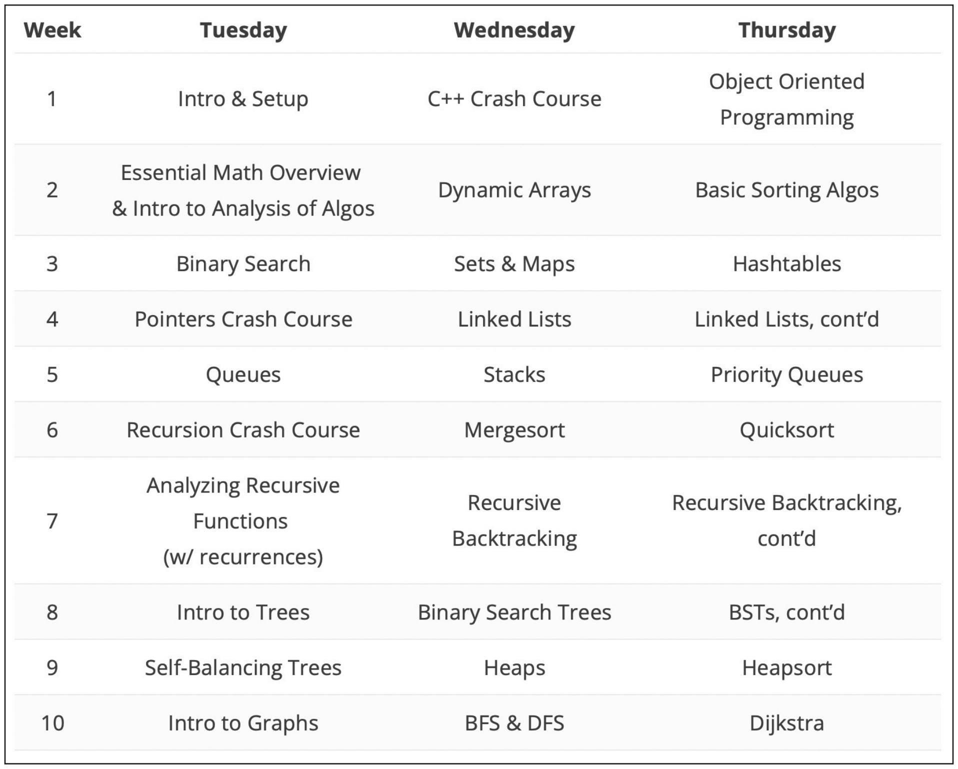
[1] For example, did you know Gen Z students' concept of a computer's file system is analogous to an unsorted and unseparated laundry basket rather than a filing cabinet? [2]

list implementations of stacks and queues by modifying their original linked list implementations. We also revisited linked lists with every tree-type structure we learned about, drawing differences and comparisons across them, and reiterating how trees could be understood as modified linked lists.

Regarding the "A" in DSA, unlike other DSA courses I was familiar with, I decided to place sorting algorithms halfway through the course rather than early on. Given that a lot of students struggled with the analysis of algorithms, I did not want to them to feel too intimidated at the outset. Additionally, this placement allowed students to have a break from data structures between linked lists and trees—this 'extra' time allowed students to revisit their assignments and resubmit as needed, solidifying their understanding of linked lists before moving on to trees.

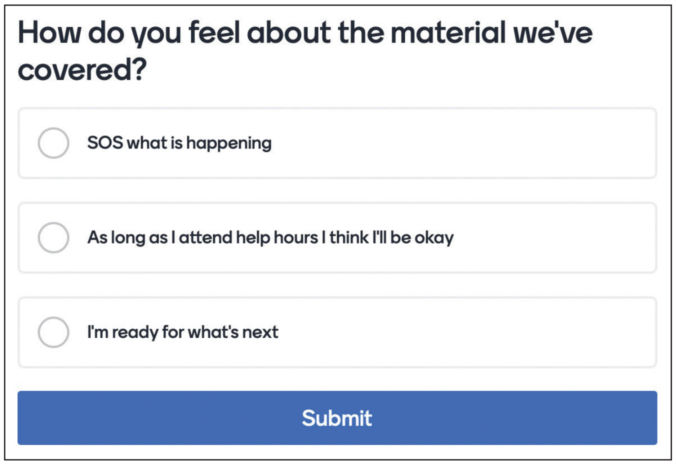
Organizing the topics in a way that made sense to students—such that they were able to see how and why topics built on each other—helped the course feel more cohesive. Despite the summer timeline being shortened by three weeks, students in the second version of the course were able to keep up just as much, if not more, than students enrolled in the first iteration of the course. I even noticed a decrease in the number of in-class questions and auto-grader resubmissions with no decrease in overall student engagement or performance.

Beyond reorganizing topics and implementing logistical changes to the course, I found it important to bring in a sense of lightheartedness to the class. Trying to learn amidst an ongoing pandemic—while there were still so many unknowns surrounding covid, the future of learning, and future of our lives—meant students and I were constantly stressed, grieving, worried, overworked, and depressed. The world was on fire all around us but despite the all-too-common programming frustrations, we were able to find and enact joy in our class. Simple things like including funny and mildly relevant GIFs to set a light mood at the beginning of class, acknowledging students' humanity and encouraging them to bring their full selves to class though informal, often silly but sometimes serious check-ins (like asking for rainy day recommendations or doing 'temperature checks' on material covered in class, see Figure 3 for an example), celebrating birthdays, and sharing cute or funny social media posts, helped students feel cared for. As an example, through these check-ins, a couple of students realized our shared love for SpongeBob. The iconic yellow sponge ended up becoming a running joke throughout the class as I often included SpongeBob themed GIFs throughout my slides and allowed time for us to reminisce on old episodes dug up from our memories by way of the GIFs. One student, Kevin, even committed to having a new SpongeBob virtual background each class! While very few of these check-ins and opening conversations were directly related to the course material and could easily be dismissed as irrelevant or wastes of time, they were crucial to us building community, students becoming more comfortable asking questions, and all of us finding reasons to laugh through the pain of the pandemic and a notoriously stressful course. At the end of the term, Kevin crafted his final project aligned with our unofficial SpongeBob theme and made a rhythm game [11] using a song from The SpongeBob SquarePants Movie.



Week	Tuesday	Wednesday	Thursday
1	Intro & Setup	C++ Crash Course	Object Oriented Programming
2	Essential Math Overview & Intro to Analysis of Algos	Dynamic Arrays	Basic Sorting Algos
3	Binary Search	Sets & Maps	Hashtables
4	Pointers Crash Course	Linked Lists	Linked Lists, cont'd
5	Queues	Stacks	Priority Queues
6	Recursion Crash Course	Mergesort	Quicksort
7	Analyzing Recursive Functions (w/ recurrences)	Recursive Backtracking	Recursive Backtracking, cont'd
8	Intro to Trees	Binary Search Trees	BSTs, cont'd
9	Self-Balancing Trees	Heaps	Heapsort
10	Intro to Graphs	BFS & DFS	Dijkstra

Figure 2: Updated topic list and schedule



How do you feel about the material we've covered?

SOS what is happening

As long as I attend help hours I think I'll be okay

I'm ready for what's next

Figure 3: Example of temperature check check-in used at the beginning of class

Admittedly, being a younger and more relatable instructor allowed my efforts to feel as genuine as they were, but I was also comfortable admitting (and often poking fun at myself) when I didn't understand a reference my students made. Asking them about these references allowed me to learn and incorporate their interests and experiences into my repertoire and provided me with material to make analogies that students found relevant and relatable. For example, one student explained how linked lists might be understood through Terraria's Stardust Dragon, an analogy that resonated with many of their peers.

CONCLUDING THOUGHTS

DSA courses need not be the intimidating monsters with obscenely high DFW rates that they currently are. While I know my approach [1] was not a perfect one and I would love to continue iterating and improving, I can confidently say that many students enjoyed taking the course and were able to take away fond memories from a class they were terrified of taking.

Pedagogically speaking, I think a flipped classroom environment worked well for this course, but I also would want to rethink the flipped aspect to maximize the time students have with the instructor. Course creep is a tremendous issue that, as I mentioned before, cannot be solved through a single course's redesign alone. Assignments should be better aligned with student interest, as that can be a great source of joy and passion for students [5], and ethics and accessibility should be a core through-line in every topic, lecture, and assignment. I found auto-grading to be an invaluable tool, especially given the large number of students, but students should also have the opportunity to explain their thought process,[2] and to receive feedback from a human. Collaboration should be encouraged, not discouraged—so much learning comes from conversations with peers and not just from listening to someone talk at them.

Most importantly, students' wellbeing and learning must be prioritized. We need to devalue grades, exams, 'easily-quantifiable' ways of measuring learning, and problematic ideas surrounding 'rigor,' instead focusing on sparking joy and having students (re)discover why they're passionate about and interested in technology. We need to have students truly experience the joy of computing, beyond year one. ❖

[2] Metacognition is a fantastic learning tool and I personally enjoyed watching my students' recorded videos as they explained their programming choices and walked me through their challenges and thought processes.

Acknowledgements
I am eternally grateful to my students, who were the life of the course. I am also grateful to my course staff, whose work and dedication made the course a success. Much gratitude to my support emotional support peers who supported all the late nights of course redesign and grading, and to Drs. Marcelo Worsley and Vishesh Kumar for their feedback on early drafts of this write-up. A special thank you and shout-out to Kevin Suggs for allowing me to share his final project video.

References
1. Chávez, V. *URI CSC 212*. https://vcc-csc212.github.io. Accessed 2023 May 3.
2. Chin, M. *File not found*. September 22, 2021; https://www.theverge.com/22684730/students-file-folder-directory-structure-education-gen-z. Accessed 2021 November 3.
3. Computer Science at Brown University. *Which Introductory Course Should I Take?* https://cs.brown.edu/degrees/undergrad/whatcourse.html. Accessed 2022 December 27.
4. Garcia, D., Caldwell, J., Fox, P. and Keeshin, J. Rediscovering the Passion, Beauty, Joy, and Awe: Making Computing Fun Again, part 8. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education (SIGCSE '16)*. Association for Computing Machinery, New York, NY, (2016), 80–81. https://doi.org/10.1145/2839509.2844659.
5. Garcia, D., Harvey, B., and Segars, L. CS principles pilot at University of California, Berkeley. *ACM Inroads* 3,2 (2012), 58–60. https://doi.org/10.1145/2189835.2189853,
6. Hertz, M. (2010). What do "CS1" and "CS2" mean? investigating differences in the early courses, in *Proceedings of the 41st ACM technical symposium on Computer science education (SIGCSE '10)*. Association for Computing Machinery, New York, NY, 199–203. DOI: https://doi.org/10.1145/1734263.1734335.
7. Lang, J. E. and Maruyama, R. K. Teaching the abstract data type in CS2. *Proceedings of the twentieth SIGCSE technical symposium on Computer science education (SIGCSE '89)*. Association for Computing Machinery, New York, NY, (1989), 71–73. DOI: https://doi.org/10.1145/65293.65301.
8. Long, T. J., Weide, B. W., Bucci, P., Gibson, D. S., Hollingsworth, J., Sitaraman, M., and Edwards, S. Providing intellectual focus to CS1/CS2. *Proceedings of the twenty-ninth SIGCSE technical symposium on Computer science education (SIGCSE '98)*. Association for Computing Machinery, New York, NY, (1998), 252–256. DOI: https://doi.org/10.1145/273133.274307.
9. Porter, L., Zingaro, D., Lee, C., Taylor, C., Webb, K. C., and Clancy, M. (2018). Developing course-level learning goals for basic data structures in CS2. in *Proceedings of the 49th ACM Technical Symposium on Computer Science Education (SIGCSE '18)*. Association for Computing Machinery, New York, NY, 858–863. DOI: https://doi.org/10.1145/3159450.3159457.
10. Sobral, S.R. CS1 and CS2 curriculum recommendations: Learning from the past to try not to rediscover the wheel again, in: Á. Rocha, H. Adeli, L. Reis, S. Costanzo, I. Orovic, F. Moreira (Eds.) *Trends and Innovations in Information Systems and Technologies*. Springer, Cham, 2020. https://doi.org/10.1007/978-3-030-45697-9_18
11. Suggs, K. CSC 212 *Final Project KS*. 24 July 2021; https://www.youtube.com/watch?v=SzK3LYhqMkI. Accessed 2021 July 30.

Victoria C. Chávez
Computer Science & Learning Sciences
Northwestern University
2120 Campus Drive
Evanston, IL 60208
vcchavez@u.northwestern.edu

DOI: 10.1145/3596918