Computer Science with Theatricality

Creating Memorable Moments in CS50 with the American Repertory Theater during COVID-19

David J. Malan
Harvard University
malan@harvard.edu

ABSTRACT

In Fall 2020, Harvard University transitioned entirely from on-campus instruction to Zoom online. But a silver lining of that time was unprecedented availability of space on campus, including the university's own repertory theater. In healthier times, that theater would be brimming with talented artisans and weekly performances, without any computer science in sight. But with that theater's artisans otherwise idled during COVID-19, our introductory course, CS50, had an unusual opportunity to collaborate with the same. Albeit subject to rigorous protocols, including face masks and face shields for all but the course's instructor, along with significant social distancing, that moment in time allowed us an opportunity to experiment with lights, cameras, and action on an actual stage, bringing computer science to life in ways not traditionally possible in the course's own classroom. Equipped with an actual prop shop in back, the team of artisans was able to actualize ideas that might otherwise only exist in slides and code. And students' experience proved the better for it, with a supermajority of students attesting at term's end to the efficacy of almost all of the semester's demonstrations. We present in this work the design and implementation of the course's theatricality along with the motivation therefor and results thereof. And we discuss how we have adapted, and others can adapt, these same moments more modestly in healthier times to more traditional classrooms, large and small.

CCS CONCEPTS

• **Social and professional topics** → **Computational thinking**; **CS1**; **Computer science education**.

KEYWORDS

analogies, demonstrations, demos, memorable moments, metaphors, pedagogy, props, sets, teachable moments

ACM Reference Format:
David J. Malan. 2023. Computer Science with Theatricality: Creating Memorable Moments in CS50 with the American Repertory Theater during COVID-19. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1 (SIGCSE 2023), March 15–18, 2023, Toronto, ON, Canada.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3545945.3569859

SIGCSE 2023, March 15–18, 2023, Toronto, ON, Canada.

ACM ISBN 978-1-4503-9431-4/23/03…$15.00
https://doi.org/10.1145/3545945.3569859

1 INTRODUCTION

CS50 is Harvard University's introductory course in computer science in which we have long used metaphors and props to bring computer science to life. But COVID-19 offered us an unusual opportunity to do so with lights, cameras, and action on stage as well, creating all the more memorable moments for students, even via Zoom.

With classes slated to be entirely online in Fall 2020, we discussed in Summer 2020 how best to proceed. CS50 had long been part of the university's program in distance education as well as, more recently, a massive open online course (MOOC). So we initially considered reusing those videos rather than create from home, on short notice, what we worried would be a lesser experience for students via Zoom.

We soon realized, though, that not only was campus itself to be unusually vacant, so was the university's drama center just down the road, home to the American Repertory Theater (A.R.T.) [10]. Moreover, the A.R.T.'s team of talented artisans, including experts in lighting, sound, props, and sets was otherwise idled, with no shows on stage. We therefore reached out to see if the theater could instead be home to a course in computer science for a term. And we asked the university if we could indeed return to campus to film classes live from the A.R.T., albeit sans audience, with those present on staff subject to COVID-19 protocols.

After much discussion and iteration on (pre-vaccine) protocols especially, both answers came back as yes. A few full-time staff were invited, but not required, to return to campus to help film the class. (At the time, all were eager to return to work in some form.) Only the instructor could be unmasked in the theater, with everyone else wearing both face masks and face shields, maintaining a distance of 15 feet from each other at all times. The theater's HVAC system was upgraded to MERV 13 filtration, with at least 400 CFM of airflow available per person.

And so we moved CS50 itself into the theater. Ironically, the course's usual classroom was another theater on campus, a historic and beautiful one at that. Though that space lacked a back stage in which to ready demonstrations and also lacked a "prop shop" in which to build them from scratch. In healthier times, too, that theater was so heavily scheduled that there just wasn't time to do much more than get in and get out.

In the A.R.T., then, we had an unprecedented opportunity to collaborate with a team of artisans with materials and talents typically unavailable to classes. And so we spent that summer and fall bringing computer science to life more theatrically than ever, creating with props and sets all the more memorable moments for students.

With that said, theatricality does not require an actual theater. (Though it did make it more fun, during that first year of COVID-19 especially.) Indeed, some of our demonstrations that year were simply higher-end alternatives to homemade equivalents that we ourselves had made in years past, often with paper and tape.

We present in this work our repertoire of theatrical demonstrations, each of which is designed not only to introduce some concept but memorably so. Each analogy and metaphor aims to allow students to use what they already know to understand some new subject [6–8], particularly one that might be outside their own comfort zone. Our own analogies and metaphors tend to be higher-level than those of Waguespack [11], which focus more on programming primitives. Ours tend to be more algorithmic, too, simpler than those of Forišek and Steinová [2] but with some overlap in domains studied by Sanford et al. [9].

For us, this work was an opportunity to reflect on how and why we do what we do, particularly when we don't have a whole theater and crew. We present each memorable moment, roughly one per week, on a stage as a metaphor itself for any classroom, large or small. Each demonstration is "unplugged," requiring neither code nor computer.

All lectures were live-streamed via Zoom to more than 500 students off campus, though students could also watch asynchronously on demand after, particularly in far-away time zones. The instructor could see, and interact with, as many as 150 students at once across three large displays. In practice, just enough students (optionally) kept their cameras on for the instructor to feel like an audience was present. Many more students asked questions live via chat of the course's teaching staff during each lecture, some of which were relayed to the instructor to answer live on camera as well.

We begin in Section 2 with these lectures' most memorable moments, including discussion of each. In Section 3, we summarize some of our less-helpful moments that we ourselves plan to re-tool. In Section 4, we summarize our results and own takeaways. And in Section 5, we conclude.

2 MEMORABLE MOMENTS

In this section, we present some of the course's most helpful moments, along with discussion of their motivations and pitfalls. We order them roughly chronologicaly, per the course's own syllabus, focusing primarily on those that a majority of students found "very helpful," per Figure 1. Students' evaluations of each demonstration are from an end-of-term survey, with an approximate response rate of 500.

The course itself is taught primarily in C toward term's start and primarily in Python toward term's end, with brief introductions to SQL and JavaScript too. But most every demonstration is language-agnostic, accompanied often by pseudocode.

In Fall 2020, with all students off campus, we enacted all demonstrations ourselves, sometimes with virtual volunteers. (See Appendix for videos thereof.) In healthier times, students themselves volunteer to come up on stage. In anticipation of healthier times ahead, we emphasize the latter versions herein, to facilitate adoption and adaptation by others. Evaluations thereof are from Fall 2020.

We claim only to have enacted the incarnations herein, the demonstrations themselves undoubtedly originated and inspired by our own teachers and colleagues. Italicized throughout are the key concepts introduced.

2.1 Tearing a phone book to explain binary search

The course's first, and perhaps most memorable, moment, is meant to introduce students to *algorithms* by way of a phone book with hundreds of pages. (We estimate 1,024 sheets.) We ask students rhetorically how we might find "David," for instance, in that phone book, assuming it's sorted (for the sake of discussion) by first name. We proceed immediately to search for him from the first page toward the last, one sheet at a time. We pause after a few page turns to ask whether the algorithm is *correct*. We confirm that it is, because if David is in the phone book, we'll eventually reach him. We then ask whether the algorithm is *efficient*. We admit that it isn't, because if he (or whoever we're calling) is toward the end, it might take us nearly a thousand steps to reach. We then restart the search, this time flipping two sheets at a time. We again ask whether the algorithm is correct. It now isn't, because David might end up "sandwiched" between two sheets. We explain there's a *bug*. We ask whether the bug is fixable. We acknowledge that, if we find that we're alphabetically beyond David, we can just double-back one or a few pages. Whereas the first algorithm might take as many as one thousand steps, we explain, the second algorithm might take only five hundred, give or take.

We ask how students themselves might search the same phone book. We posit that most would open the phone book to, roughly, its middle. We demonstrate such and claim that we're in the "M" section. We ask what we now know about the pages to the left and to the right of that section. We confirm that David must be to the left. We dramatically tear the phone book in half (down the spine) and throw (the right) half of the problem away. We continue to *divide and conquer* just as dramatically until we're left with one sheet. We ask how many steps that third algorithm might take. We confirm roughly ten (i.e., $\log_2 1{,}024$). We pretend to call David. We assure students that they have the intuition already to be successful in computer science.

We then give the first two algorithms a name, *linear search*, likening them to searching along a line, from left to right, albeit at different rates. We then give the third algorithm a name, *binary search*, noting that a prefix of "bi" implies two, just as we split the phone book in two. We emphasize just how much faster that third algorithm is: if the phone book were to double in size, the first algorithm might take one thousand more steps, the second algorithm might take five hundred more, but the third algorithm would take only one more. We present Figure 2, explaining that the third algorithm takes not *linear* but *logarithmic* time.

We translate the algorithm to pseudocode, introducing verbs as *functions*, decisions as *conditionals* with *Boolean expressions*, and repetitions as *loops*.

2.1.1 Discussion. Not only do 88% of students report this moment to be helpful, per Figure 1, students anecdotally report this moment to be the course's most memorable as well, even years later. The demonstration is not without pitfalls, though. If we start by asking

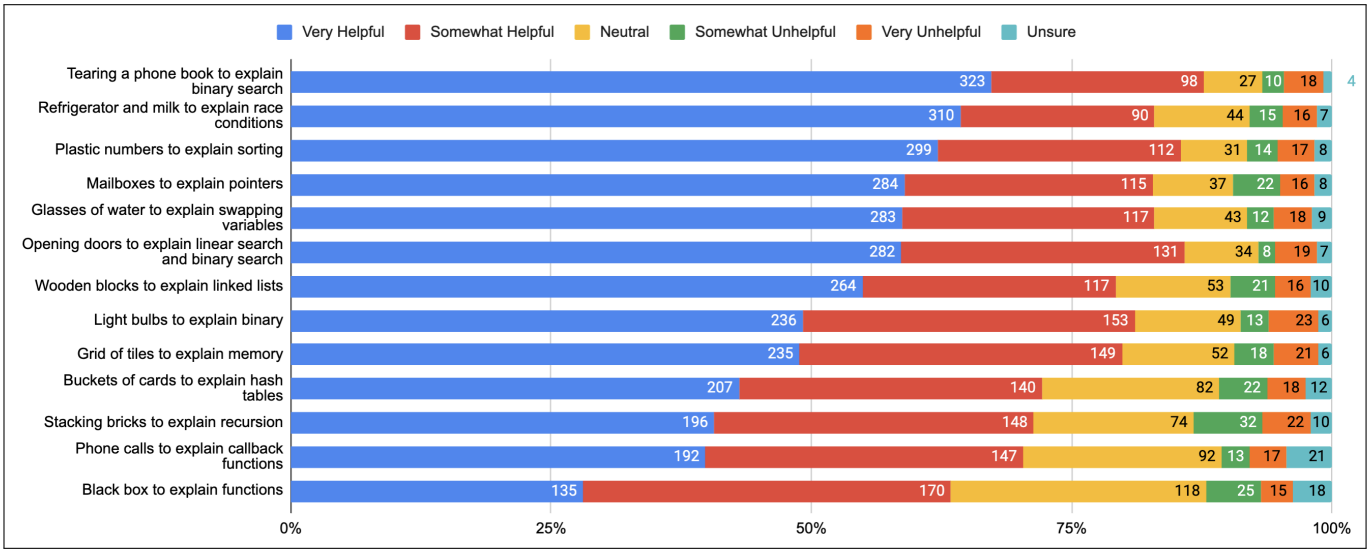

Figure 1: At term's end, the course's students were asked to evaluate the course's demonstrations. A majority of students found more than half of the demonstrations "very helpful." And a majority of students found all of the demonstrations "very helpful" or "somewhat helpful."

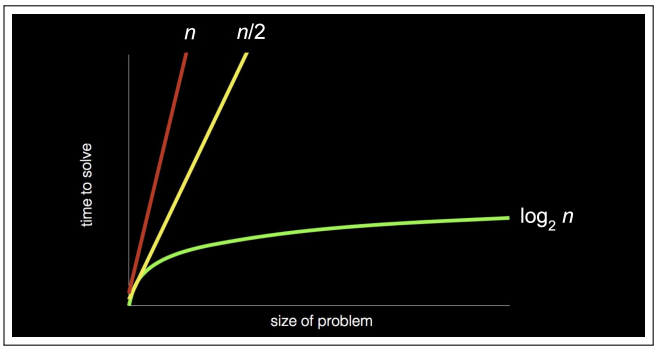

Figure 2: We introduce students to algorithms in the course's first lecture, including high-level previews of linear search and binary search. Depicted here is the time required to search a phone book, using linear search (one or two sheets at a time) or binary search, as a function of the size, *n*, of the phone book.

students how we could search for David, instead of diving immediately into a linear search ourselves, someone invariably shouts that we should use binary search, using terminology that most of their classmates would not yet know. When asked whether our first algorithm is correct, students frequently answer no, conflating inefficiency with incorrectness, though that itself is a teachable moment. When asked how else to search for David, students frequently propose starting with the "D" section, sometimes assuming incorrectly that the phone book has an index along its edge (which, admittedly, some do), sometimes not realizing that finding that, would itself, take some number of steps. As for the demonstrations themselves, it's physically difficult to flip two sheets at a time at a uniform speed, to convey that it's (theoretically) twice as fast as just one. It's easier to find thick (commercial) yellow pages than (residential) white pages, so we tend to pretend that David is in yellow. Though it's increasingly difficult to find any phone books at all. And, as of 2021, at least one student reported not knowing what a phone book even is. Though we now present a screenshot of a mobile phone's contacts, to liken our search to now-familiar autocomplete.

2.2 Opening doors to explain linear and binary search

We later revisit linear and binary search at a lower level. We erect an *array* of seven doors in standalone door frames on stage, all of them closed, side by side. Hiding behind each is a different life-sized number (some with fur, some with googly eyes). We invite a volunteer to try to find a particular number. We emphasize that the doors are closed so that they must methodically *index* into them, opening one at a time. Once the student finds the number, we ask them to explain what their algorithm was.

We then close all the doors, rearranging the numbers behind them, this time arranging them in ascending order, left to right. We invite another volunteer to find a different number, disclosing that the numbers are now *sorted*. Once the student finds the number, we ask them to explain what their algorithm was and how the added assumption helped them, if it did.

2.2.1 Discussion. 86% of students find this demonstration helpful, per Figure 1. Though this demonstration is certainly possible without all the doors. Without the luxury of a prop shop, we sometimes cart with us seven small lockers instead (which could alternatively be those in a hallway for small classes). And in smaller classrooms, we typically use seven sheets of paper taped to a blackboard, with numbers written in chalk behind them. For the first volunteer, we

find it helpful to select a small number like 0, hidden behind the rightmost door, as many students start from the left and move right, thereby demonstrating that linear search is in $O(n)$. Some students open doors randomly and even get lucky, finding the number immediately by chance. Hilarity tends to ensue, all the more memorably, if suboptimally, for a discussion of O, but opportunely for a discussion of Ω as well. Almost all volunteers decide to apply binary search to the second set of doors, so we find it helpful to prescribe a number that we expect will be behind the last door they open. We always use seven doors so that each subset has a well-defined middle, without rounding. We daresay that fifteen or more might be better for more repetition but might be unwieldy to set up. Ideally, we use a different set of numbers for each search (one unordered, one ordered), lest students remember the first search's numbers and deduce, in constant time, where some sorted number will be.

2.3 Plastic numbers to explain sorting

But how much time does it take to sort those numbers before we can use binary search? To introduce students to *comparison sorts*, we place an *array* of eight plastic numbers (each of which lights up with a switch) on a table or shelf, initially *unsorted*. We ask a volunteer to come up and *sort* them in ascending *order*. We then ask the volunteer what their algorithm was. Invariably, they describe some variant (or amalgam) of *selection sort* and *bubble sort*, though not by name.

We then reset the numbers to their original, unsorted positions. We propose to sort the array by selecting the smallest number first. We emphasize that we must step through the array, left to right, to determine which number is smallest; we can't decide, at a glance all at once, like the audience can. (We don't bother with closed doors, for time's sake.) As we step through the array (literally, walking a step at a time, left to right), we make clear that we're making mental note, as with a *variable*, of the *index* of the smallest number we've seen yet. Once at the end of the array, we pick up the number at that index and ask students where we should put it instead. A student invariably proposes to put it at the beginning. We remind them that we can't just make room for it there on the edge of the table or shelf; the array has a fixed length. A student typically proposes that we shift everything to the right, in which case we respond that it seems like a lot of (unnecessary) work. We counter-propose swapping the smallest number with the leftmost number. After all, if the latter was there randomly anyway, we're not necessarily making the problem any worse. We then light up that smallest number, now in its final location. And we repeat for the next-smallest numbers in turn. We observe that the whole list is *sorted* once all numbers are lit. We reveal the algorithm's name to be selection sort. And we demonstrate it once more, this time using digital animation [3] with vertical bars of short and tall heights representing small and large numbers, respectively, to make the selection of each smallest number more visually clear. We conclude with an analysis of our total number of steps. We observe that our search for the smallest number took n steps (or $n-1$ comparisons); our search for the next-smallest took $n-1$ steps; and so forth. We ask students to trust us that the sum thereof is $(n^2+n)/2$, which is in $O(n^2)$. We explain why the algorithm, as defined, is also in $\Omega(n^2)$ and, in turn, $\Theta(n^2)$.

We then reset the numbers again. We propose to do better. We ask students to explain in what sense the array is unsorted. A student typically cites an example of two (adjacent) numbers that are out of order. We proceed to swap the two numbers, followed by any other such pairs, left to right, again and again. As larger numbers "bubble" their way toward the end of the list, we light them up once in their rightmost positions. We again observe that the whole list is sorted once all numbers are lit. We reveal the algorithm's name to be bubble sort. We demonstrate it once more, again using digital animation to make the pairwise swapping and bubbling more clear. And we again analyze our number of steps, comparing $n-1$ pairs of numbers as many as $n-1$ times, totaling $(n-1)^2$ steps, which is again in $O(n^2)$. But we note that, if we short-circuit this algorithm after a pass with no swaps, we can achieve $\Omega(n)$ this time instead.

We reset the numbers one final time, ideally on a shelf with one or more empty shelves below (representing additional space). We then introduce students to *merge sort* by way of pseudocode, which we then enact, merging sorted halves from one shelf to another. We explain why merge sort is in $\Theta(n \log n)$. We conclude class itself with a digital animation showing selection sort, bubble sort, and merge sort in parallel, with the last dramatically finishing an order of magnitude faster [1].

2.3.1 Discussion. 85% of students find these moments helpful as well, per Figure 1. But we have learned not to leave too much of the narrative to chance, as by asking too many questions. On occasion, students have responded with algorithmic suggestions that don't quite map to the algorithms (and order thereof) that we hope to discuss. Pedagogically, we prefer the progression from good, to better, and to best, asymptotically speaking. More so than other demonstrations, then, we tend to steer this one's narrative. These algorithms' enactments can be messy as well without practice, as it's all too easy to stand in front of the numbers you're swapping or merging, blocking students' own view. (And standing behind the numbers instead otherwise involves doing everything backwards.) When lacking for shelf space, we sometimes have students hold the numbers themselves (or printouts thereof) and implement selection sort and bubble sort physically, which tends to be fun but less visually clear. The digital animations are good supplements, though, as they present the same algorithms graphically, at a uniform pace.

2.4 Glasses of water to explain swapping variables

To help students understand and implement in-place sorting algorithms (or swap values more generally), we take out two glasses of water, each colored differently via drops of food coloring. (We've used milk and orange juice too.) We explain that each glass represents a *variable*, with the colored water therein its *value*. (By this point in the course, students have used variables to store values but not necessarily swap.) We ask for a student to volunteer to *swap* the two liquids, somehow pouring the water from one glass into the other and vice versa. The volunteer typically hesitates, at which point we offer them an empty glass as a *temporary variable* with which to perform the swap in three steps. We then translate the steps to three lines of code for which students then have a mental model.

2.4.1 Discussion. Per Figure 1, 83% of students find this demonstration helpful. It later lends itself, too, to our discussion of *scope*, whereby those same lines of code seem to fail if implemented within a function, in which the swap is local. We then introduce *pointers* as a solution thereto, *passing by reference* rather than *passing by vaule*. Among more comfortable students, we sometimes demonstrate how to swap values without a temporary variable using bitwise XOR operations instead. We have even tried to enact such (using just two glasses) with water and oil, which theoretically don't mix, but not visibly enough for a good metaphor. On at least one occasion, too, when asked to swap two liquids (without an empty glass), a student simply swapped the positions of the two glasses, so we have since clarified our instruction.

2.5 Mailboxes to explain pointers

To help students understand memory and *pointers* during our first several weeks in C, we have long drawn pictures with arrows. More helpful, perhaps, has been a pair of traditional mailboxes (on posts) that we picked up from Home Depot. One has a name, *P*, clearly labeled as such with a sticker, akin to a family's name on a mailbox. When we open that mailbox, we find a value, a sheet of paper with an *address* (e.g., `0x123`) or, alternatively, a map wherein X marks a spot. At that address or X across stage is another mailbox, labeled as such, to which we point using an oversized foam finger (as you might find in a stadium). Inside of that mailbox is a (non-pointer) value or, better yet, treasure.

2.5.1 Discussion. 83% of students find this demonstration helpful, per Figure 1. Better still, though, might be a whole wall of mailboxes, as you might find in an apartment instead, which would resemble a bank of memory more closely, albeit more challenging to set up.

2.6 Wooden blocks to explain linked lists

To help students visualize the data structures that they can then build with those pointers, we build a *linked list* of students on stage. We `malloc` one student at a time, asking each volunteer to represent a *node*, holding some value in one hand and a foam finger in the other, pointing at the next node in the list (or at the floor if `NULL`). We emphasize that the nodes need not be contiguous in memory; we deliberately spread volunteers out. And we ask the audience how we might insert additional nodes at the start, end, and middle of the list, enacting each in turn. Invariably, some nodes are accidentally orphaned during insertions, at which point we discuss *memory leaks* too.

2.6.1 Discussion. 79% of students find this demonstration helpful as well, per Figure 1. With no students in person during COVID-19, though, we temporarily replaced volunteers with large wooden blocks, built by the prop shop to represent nodes, each connected to another via an orange extension cord and (unpowered) receptacle. Though we underappreciated just how heavy 3-feet-tall wooden blocks would be to move on stage, an accidental metaphor, perhaps, for how difficult memory can be to manage.

2.7 Refrigerator and milk to explain race conditions

Toward term's end, we introduce students to real-world issues in computing like *race conditions* in the context of databases and SQL specifically. We invite students to consider what could go wrong if two users happen to "like" the most-liked egg on Instagram [4] at the same time, if updating that post's counter isn't *atomic*. And we enact that same issue with an old-time refrigerator on stage. We propose that one roommate arrives home to discover the refrigerator out of milk. And so they head out to buy more. In the meantime, another roommate returns home, only to discover the same. They, too, head out to buy more (without crossing paths with the other). The end result is more milk than they can both drink before it goes sour. We observe that the problem arises because, after one roommate makes a decision based on the *state* of the refrigerator, itself a *variable* of sorts, the other roommate makes a similar decision while that variable's value is in the midst of an update. We ask students how to prevent such a "race." Often, a student proposes that the first roommate leave a note. We counter-propose, more dramatically, that they instead *lock* the refrigerator outright (as with a chain and padlock we have on stage), unlocking it only once the value is updated. We then mention finer-grained *transactions* as well.

2.7.1 Discussion. 83% of students report this demonstration to be helpful, per Figure 1. The need for a refrigerator, though, limits its reenactment. We ourselves sometimes resort to a verbal narration instead, an oral allegory of sorts [5], or to a plastic miniature thereof.

3 FUTURE WORK

We present more succinctly in this section the demonstrations that fewer than half of students found "very helpful," per Figure 1. (See Appendix for videos thereof.) A majority of students still found them, at least, "somewhat helpful," so we plan to solicit additional feedback in future terms in order to refine or rethink each.

3.1 Light bulbs to explain binary

We first introduce introduce students to *binary* by way of light bulbs, each of which has a battery as well as a switch, akin to a *transistor*. We start with just one such *bit*, to represent 0 and 1. We then upgrade to three, to count in (unsigned) binary from 0 to 7. We later point out, after introducing *Unicode*, that the 64 light bulbs along the edge of the theater's own stage might actually be encoding a message (e.g., `HI MOM`). We suspect we might be spending too much time on this "bit."

3.2 Grid of tiles to explain memory

Thanks to the theater's prop show, we have an 8-by-8 grid of wooden tiles that represents a bank of *memory*, with each tile a *byte*. The tiles are dry erase-friendly, allowing us to draw values atop them. Beneath each tile is an icon of Oscar the Grouch representing a *garbage value* as well. We use the grid to depict *stack frames* especially, to clarify why swapping two values inside in a function has no effect on the caller's copies thereof (i.e., other tiles), unless the values are passed in by reference. Moving the (magnetic) tiles tends to be clumsy, though, so we suspect we can improve this demonstration through more practice or digitization thereof.

3.3 Buckets of cards to explain hash tables

When discussing *hash tables* in the context of data structures, we introduce students to *hashing* by way of oversized playing cards, hashing each card into one of four buckets (also from Home Depot) according to its suit. Because this particular demonstration only indirectly relates to how we later hash strings into actual hash tables, we might replace this one altogether.

3.4 Stacking bricks to explain recursion

When discussing *recursion*, we observe that a pyramid of "bricks" from Super Mario Bros. is itself a recursive structure whereby a pyramid of height n is but a pyramid of height $n - 1$ plus another layer of bricks. Because of gravity, though, it's difficult to build such (with cardboard blocks) on stage, as by lifting $n - 1$ layers to add the other. We might revert to presenting this structure graphically.

3.5 Phone calls to explain callback functions

When surveying paradigms in other languages at term's end, we introduce students to *asynchronous* functions and *callbacks* in JavaScript by calling a colleague during class on the phone. Upon picking up, they explain that they'll need to call us back with the answer to some question. Class is later interrupted with an asynchronous callback. We spend relatively little time on JavaScript itself, so we suspect the returns of this demonstration are simply low in terms of opportunities for application thereof.

3.6 Black box to explain functions

When first introducing students to *functions*, we take out an actual *black box*, inserting into it one or more *inputs* (e.g., two slips of paper with numbers) and taking out some *output* (e.g., another slip of paper, prepared in advance with the sum of those numbers). We explain that we don't (need to) know how the box works inside; its *implementation details* are, for now, *abstracted* away. Insofar as we have tended to present this demonstration before implementing actual functions with code, we suspect it's not obvious during the demonstration itself what we are actually abstracting away. We might try presenting it afterward instead, so that it's clearer in retrospect what the abstraction represents.

4 RESULTS

When asked at term's end what they thought of the course's use of physical props during lectures to help explain topics, nearly all students reported loving (65%) or liking (27%) the same, per Figure 3. They were "engaging" and "fun," reported some students. "It really helped me understand at a different level and helped me remember the material better," reported one student. And "they made me comprehend something instead of just recalling it," explained another. "It helped to visualize all of the code and ideas that would otherwise just be letters, symbols, numbers, and indents," elaborated another.

That said, not all students felt the same, with time spent a concern. For instance, one found that "they were often unnecessary. Helpful, but unnecessary. For me, an animated diagram on the slides would have been sufficient, and often felt that these demonstrations took up much more time than necessary." Another distinguished between engagement and learning: "I liked them because they were engaging, but I don't think they helped a lot with my learning." But

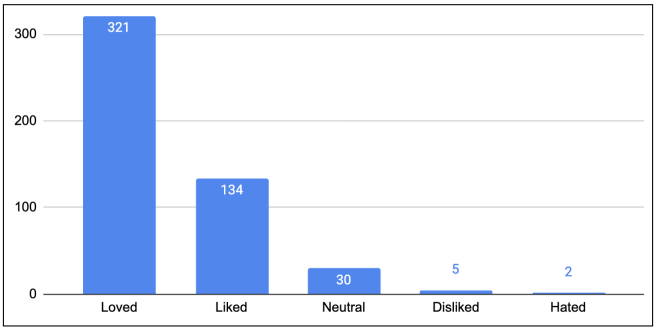

Figure 3: When surveyed at term's end, 321 students (65%) reported that they "loved" the course's use of physical props during lectures to help explain topics, and 134 students (27%) reported that they "liked." Students were separately asked to elaborate why in prose as well.

engagement did recur as a theme: "it really helped me stay engaged, especially in the virtual nature of this semester." And as another student acknowledged, "I don't really get value from stuff like that but I know it helps others."

To be fair, we have not run a controlled experiment, creating memorable moments for some students but not others. Nor have we ourselves tracked students' performance beyond course's end. But that so many students reported those moments as still memorable at course's end is encouraging for long-term retention and comfort as students move on to higher-level courses next with those foundations in place. And we do plan to reduce time spent on some demonstrations in order to strike a better balance.

5 CONCLUSION

CS50's flair for theatricality predates COVID-19 itself. But a silver lining of that particular moment in time was an opportunity for us to collaborate with a team of artisans to bring computer science all the more to life on a stage, for an audience, no less, that could not be there in person. The collaboration, too, proved an opportunity for us to reflect on how we might introduce students more effectively to that which is not familiar by way of that which already is, via analogies, metaphors, and theatrical props. Ultimately, the collaboration did not enable pedagogical techniques that aren't already available to us and others off-stage as well, equipped as we more often are with just paper and tape. But it did come at just the right time for so many students who were otherwise isolated at home.

APPENDIX

See https://www.youtube.com/cs50 for videos of every demonstration herein.

ACKNOWLEDGEMENTS

Many thanks to the author's own teachers who originated or inspired these moments and so many more, including Brian Kernighan, Henry Leitner, Margo Seltzer, et al. And to CS50's own team, including Doug Lloyd, Brian Yu, Carter Zenke, et al.

And many thanks to the American Repertory Theater's team of artisans for bringing to life so many of these moments on stage.

REFERENCES

[1] Viktor Bohush. 2022. SortingAlgorithmAnimations. https://github.com/vbohush/SortingAlgorithmAnimations

[2] Michal Forišek and Monika Steinová. 2012. Metaphors and Analogies for Teaching Algorithms. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education* (Raleigh, North Carolina, USA) *(SIGCSE '12)*. Association for Computing Machinery, New York, NY, USA, 15–20. https://doi.org/10.1145/2157136.2157147

[3] David Galles. 2022. Comparison Sorting Algorithms. https://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html

[4] Egg Gang. 2022. https://www.instagram.com/world_record_egg/

[5] Jeisson Hidalgo-Céspedes, Gabriela Marín-Raventós, Vladimir Lara-Villagrán, and Luis Villalobos-Fernández. 2018. Effects of oral metaphors and allegories on programming problem solving. *Computer Applications in Engineering Education* 26, 4 (2018), 852 – 871.

[6] D. Hofstadter and E. Sander. 2013. *Surfaces and Essences: Analogy as the Fuel and Fire of Thinking*. Basic Books.

[7] Yu-chen Hsu. 2006. The Effects of Metaphors on Novice and Expert Learners' Performance and Mental-Model Development. *Interact. Comput.* 18, 4 (July 2006), 770–792. https://doi.org/10.1016/j.intcom.2005.10.008

[8] George Lakoff and Mark Johnson. 1980. *Metaphors we Live by*. University of Chicago Press, Chicago.

[9] Joseph P. Sanford, Aaron Tietz, Saad Farooq, Samuel Guyer, and R. Benjamin Shapiro. 2014. Metaphors We Teach By. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education* (Atlanta, Georgia, USA) *(SIGCSE '14)*. Association for Computing Machinery, New York, NY, USA, 585–590. https://doi.org/10.1145/2538862.2538945

[10] American Repertory Theater. 2022. https://americanrepertorytheater.org/

[11] Leslie J. Waguespack. 1989. Visual Metaphors for Teaching Programming Concepts. In *Proceedings of the Twentieth SIGCSE Technical Symposium on Computer Science Education* (Louisville, Kentucky, USA) *(SIGCSE '89)*. Association for Computing Machinery, New York, NY, USA, 141–145. https://doi.org/10.1145/65293.71203