Emmett Wald

Dr. Lyndsey Nunes

EDUC 540: Exceptional Learners

10 May 2021

Instructional Strategies for Teaching Computational Thinking to K-12 Students with Learning Differences

Over the past decade, the rapid growth of the tech industry has led to greater demand for computer science literacy in the general population [1]. As a result, computer science (CS) is increasingly a part of K-12 education [1], [2]. However, the research on implementing CS in the K-12 curriculum remains sparse. Furthermore, as CS has migrated from elective to core curriculum, students with disabilities have often been left out of the classroom entirely [2], [3]. According to the most recent statistics available (from the 2018-2019 school year), 14% of students in the U.S. receive special education services under IDEA (the Individuals with Disabilities Education Act) [4]. Assuming we don't wish to exclude these 7.1 million students [4] from such a vital subject, it's imperative that we actively pursue pedagogy that will make a place for students with disabilities in the computer science classroom.

This paper aims to take a methodical approach to this issue, examining the cognitive demands of computational thinking (a core component of CS; see section 2 below) in order to extract and distill concrete instructional strategies for teaching students with learning differences, so that these principles can be applied to a variety of learners in a variety of contexts. Though assistive technology is also vital for students with disabilities, it is beyond the scope of this paper.

For the purposes of this paper, "learning differences" will include students with learning disabilities, mild intellectual disabilities, speech/language impairments, and/or ADHD. This follows Gage et al. [5], who found no significant differences in measures of cognitive performance among these categories of learners. However, the instructional strategies put forth in this paper should not be considered exclusively for students with disabilities; per universal design for learning, they are approaches that seek to make learning more accessible to *all* students.

Section 1 reviews existing research about teaching computer science to students with disabilities. Section 2 discusses computer science and computational thinking, explaining why abstraction is a suitable proxy for both. Section 3 delves into an exploration of the cognitive processes involved in abstraction and their connections to learning differences. Section 4 lays out instructional strategies to address the challenges associated with these cognitive processes. Finally, section 5 concludes the paper with a consideration of future directions for research in this area. Appendix A summarizes the instructional strategies in condensed form, and Appendix B offers an example lesson plan following the principles in this paper.

1. Left Out of the Classroom—And the Research

Many researchers (e.g., [2], [6], [7]) have noted the dearth of educational research that addresses teaching CS to students with disabilities. These researchers and others (e.g., [1], [3], [8]) have begun to fill the gap. However, the research often focuses narrowly on a case study of a specific curriculum with specific students, making it difficult to draw broadly applicable conclusions. Furthermore, several researchers utilize a deficit framework that treats students with disabilities as problems to be solved or fixed, or consider the accommodation of disabilities as an afterthought.

For example, Bouck & Yadav [3] state that "students with disabilities are left out" of CT and CS, but their suggested approach for inclusion suggests that teachers go through five steps of designing an instructional activity before, only in the sixth and final step, considering "accommodations or assistive technology" for students with disabilities. This intentionally last-minute, narrow-sighted modification of lesson plans simultaneously reflects and reinforces an implicit belief that the instructional needs of students with disabilities are extraneous and excessive, and likely to take away from the instructional needs of students without disabilities. In contrast, Israel et al. [1], Hansen et al. [2], and Outlier Research [9] emphasize principles such as universal design for learning, differentiated instruction, culturally responsive pedagogy, and other proactive strategies for making CS curriculum more inviting and responsive to all learners, including students with disabilities.

CS education researcher Maya Israel is a strong advocate for the inclusion of students with disabilities in CS. Israel succinctly states the goal of much of her research and of this paper: "If we can anticipate learning barriers in CS education and then proactively build in […] pedagogical approaches to address those barriers, students can experience success alongside their peers" [10]. In order to anticipate these learning barriers, we must first examine what computer science education is all about.

2. Computer Science, Computational Thinking, and Abstraction

When teachers and researchers seek to make CS accessible to all students, the first hurdle is that computer science itself is not always a well-delineated subject. Some seem to perceive CS as synonymous with coding alone (e.g., [2]), while others define it much more broadly as everything from what a motherboard is to the ethical implications of computers in society (e.g., [11]). Clearly, the concepts and cognitive abilities associated with building a computer from parts are as different from those involved in analyzing the impact of AI as physics is from philosophy.

As such, it’s difficult to define an essential set of instructional strategies for computer science as a whole.

Computational thinking (CT), a core component of computer science, is much more narrowly and consistently defined: it is a problem-solving framework whose primary conceptual skills are abstraction, decomposition, pattern recognition, algorithmic thinking, and debugging [3], [12]–[14]. Many educational researchers (e.g., [1], [3], [13]) have pointed out that the value of CT goes beyond CS: as a problem-solving approach that builds critical thinking skills, it can be applied in myriad other contexts, including mathematics, science, and everyday life. Some suggest that computational thinking is the essence of computer science (e.g., [14]). Though reducing CS to such a fine point seems extreme, this paper will focus on instruction in computational thinking—not as the sole essence of CS, but as a foundational skill for CS and other subjects.

In order to develop instructional strategies, it’s important to first identify learning goals and likely stumbling blocks. Drilling deeper into the key concepts of computational thinking, we can define them as follows:

1. *Abstraction* – “Changing the resolution” [14] on a problem to focus on essential details and discard extraneous ones [3], [12], [13].
2. *Decomposition* – Breaking a complex problem into smaller, easier-to-solve parts [3].
3. *Pattern recognition* – Noticing repetition and using it to make predictions or solve problems more efficiently [13].
4. *Algorithmic thinking* – Developing a clearly defined, step-by-step procedure to solve a problem or complete a task [13].
5. *Debugging* – Testing a proposed solution, locating errors, and correcting them [3].

However, according to Statter & Armoni [14], “these ideas can be viewed as manifestations of one fundamental idea, which in a sense is the essence of CS”: abstraction. Each of these skills or tasks involves focusing on certain aspects of a problem or solution while setting aside the rest for the moment. Decomposition can be accomplished by concentrating on one sub-portion of a problem and ignoring the rest; pattern recognition requires finding the correct “resolution” at which the pattern operates; algorithmic thinking involves considering the underlying structure of a task, possibly ignoring finer details; and debugging often involves searching for a needle in a haystack, holding onto the larger structure while focusing in on the finer details, often at the level of programming syntax. As such, in considering challenges and opportunities in teaching computational thinking, we can consider abstraction a proxy for the various sub-tasks, and appoint it the primary learning goal.

3. Cognitive Demands of Abstraction: Challenges and Opportunities

We’ve shifted from the broad, amorphous field of computer science to the better-delineated concept of computational thinking, and then further identified abstraction as the crux of computational thinking. This narrower and more precise context will permit a clearer examination of the cognitive tasks involved, and thus help identify specific instructional strategies.

Abstraction as a computational thinking task is “a difficult concept to teach at all age levels” [14], and it’s understood that abstraction as a general cognitive task is often particularly difficult for students with learning differences [1], [15], [16]. Michal Armoni has done extensive research on conceptualizations of abstraction, as well as strategies and common difficulties in teaching it (e.g. [12], [14]). In proposing a framework for teaching abstraction, Armoni built on the levels of abstraction described by Perrenet et al. [17], [18], labeling their structure the PGK hierarchy [12], [14]. The PGK hierarchy is summarized in Table 1:

	Name	***Description***	***Example***
LEVEL 4 *(highest level)*	Problem	A "black box" object with its own characteristics and attributes.	Finding the maximum value on a list.
LEVEL 3	Object/Algorithm	An algorithm, the complexity of which can be measured or calculated.	Set first element as *max*, then compare each value to *max* and update if current value is larger than *max*.
LEVEL 2	Program/Process	An implementation of an algorithm in a specific programming language.	Writing a Python program using the above algorithm.
LEVEL 1 *(lowest level)*	Execution	A specific run of a specific program on a specific computer.	Using the Python program to find the largest value from the list [17, 9, 42, 23, 18].

Table 1. The PGK hierarchy as laid out by Michal Armoni [12], [17], [18].

Armoni [12] argues that successful computational thinking necessitates relative fluency in understanding and working with these various levels of abstraction. However, both educators and students often demonstrate poor recognition of the importance or the application of abstraction, and CS education experts disagree about how best to approach teaching it [14]. In fact, both teaching and student work, especially at the K-12 level, tend to remain stuck in the lower levels of abstraction, with little time spent in level 3 and only rare (if any) excursions up to level 4 [14]. Part of this struggle, Statter & Armoni [14] contend, stems from the fact that abstraction, being difficult to define, is also difficult to measure; thus, gauging the relative efficacy of different pedagogies is not as straightforward as it might be in, say, reading, which has well-established, evidence-based metrics for success. Accordingly, one of their goals was to develop a means of assessing abstraction abilities. Armoni laid the foundation for this by defining four essential subskills necessary for successful abstraction [12], as summarized in Table 2 below:

Skill	***Example***
1. **Differentiating** between the four levels of abstraction.	Recognizing that pseudocode belongs to level 3, but a program written in Java is level 2.
2. **Moving intentionally** between the levels.	Upon completing an algorithm (level 3), deciding it's now appropriate to move on to writing the code (level 2).
3. Deciding **which level** of abstraction is appropriate at each stage of problem solving.	In debugging a specific run of a program (levels 1-2), recognizing that an error in algorithmic thinking requires moving back up to level 3.
4. Making **refinements** of abstraction, e.g., moving from a general to a more detailed algorithm (within level 3.)	Rewriting an algorithm in detailed pseudocode (both level 3) before moving on to implementation (level 2).

Table 2. Essential subskills for abstraction [12].

Whereas Statter & Armoni move next to developing metrics for assessing abstraction capabilities, for the purposes of this paper it is more important to turn to an analysis of the cognitive demands associated with abstraction as defined above.

The Cattell-Horn-Carroll (CHC) theory of cognitive abilities is "the most comprehensive and empirically supported psychometric theory of the structure of cognitive abilities to date" [19]. It defines 16 "broad" cognitive abilities, with an additional 81 "narrow" cognitive abilities organized beneath them. This framework is often used in cognitive testing of students with learning differences [19], and it provides a helpful structure for identifying the cognitive skills involved in abstraction and computational thinking.

Of the sixteen broad abilities, the one most essential to abstraction is *Fluid Intelligence*, which is used in processes such as "forming and recognizing concepts, perceiving relationships among patterns, drawing inferences, comprehending implications, problem solving, extrapolating, and reorganizing or transforming knowledge" [19]. These processes are a vital part of differentiating, using, and moving between different levels of abstraction, and thus vital to computational thinking. Fluid intelligence comprises three narrow abilities: induction (determining the underlying concept/process in a problem), sequential reasoning (starting with defined rules/conditions and taking steps to find a solution), and quantitative reasoning (dealing with mathematical concepts) [19]. While the latter is not necessarily involved in abstraction, induction's search for the fundamental characteristic of a problem is level 4 abstraction, while the problem-solving steps of sequential reasoning are central to algorithmic thinking (level 3). Some additional broad CHC cognitive abilities associated with abstraction as defined by Armoni:

- *Short-Term Memory*: Particularly important for the detail work at levels 1-3.
- *Long-Term Storage and Retrieval*: Sometimes called "idea production, ideational fluency, or associative fluency" [19], which reveal the ways this ability is important in the creative use of stored knowledge of concepts or structures. Includes meaningful memory, associational fluency, alternative solution fluency, originality/creativity, and ideational fluency [19], all of which relate to generating relevant ideas, responses, or solutions.
- *Processing Speed*: Associated with "cognitive tasks [that] often require maintained focused attention and concentration" [19], such as programming and debugging (abstraction levels 1-2).

In addition, several of the cognitive abilities, while not essential for abstraction itself, support the cognitive processes associated with learning and using abstraction in the computer science classroom:

- *Crystallized Intelligence*: Helpful for receptive and expressive communication. Includes language development, lexical (vocabulary) knowledge, listening ability, and communication ability [19].

- *Reading/Writing*: Helpful for communication as well as comprehension and production of algorithms and code. Includes reading decoding, reading comprehension, writing ability, and others [19].
- *Decision/Reaction Time or Speed*: Helpful for efficient performance on computational thinking tasks. Includes semantic processing speed, mental comparison speed, and inspection time, which are related to mental manipulation and recognition of differences between items [19].

These are the cognitive processes involved with abstraction and its context in computing education; what challenges might these pose to students with learning differences?

As previously stated, this paper uses "learning differences" to mean students with learning disabilities (LD), mild intellectual disabilities (MID), speech/language impairments (SLI), and/or attention deficit-hyperactivity disorders (ADHD). Of course, despite Gage et al.'s conclusion that students in these disability categories have similar cognitive performance, they caution that "interventions and programs should be provided to students *based on individual need*, not disability category" [5; emphasis added]. Within and across disability categories, students' learning styles and needs are far from monolithic. However, it can be helpful to "consider the basic, psychological processes underlying the disorders and subdisorders" [7] when developing instructional strategies. Table 3, below, identifies common challenges associated with each of the seven broad cognitive abilities discussed above.

This paper takes an anti-deficit framework regarding students with disabilities. Rather than seeing a student with disabilities as "lacking inherent skills and attributes that are necessary for success," disability is recognized as a "mismatch between personal competency and environmental demands" [6]. Identifying cognitive challenges experienced by students with learning differences should not be taken as an attempt to point out what these students cannot do, but rather as an attempt to point out the cognitive demands placed on them that may be incompatible with their skills and capabilities. This in turn allows for an examination of those cognitive demands: What supports can be put in place to give students a better chance of success?

CHC Cognitive Ability	***Specific Learning Differences***	***Challenges***
Fluid Intelligence	Dyscalculia, nonverbal learning disorders, MID	Sequencing, cause and effect, pattern recognition, formal logic, understanding and using algorithmic structures, complex reasoning, problem-solving, big-picture thinking
Short-Term Memory	ADHD	Following directions, remembering the order of steps in a solution, holding an algorithm in memory while translating it to code, keeping track of variables and functions
Long-Term Storage and Retrieval	SLI, MID, ADHD	Learning new concepts and terminology, applying existing skills in the novel context of CS, applying newly learned concepts to problems/tasks, generating possible solutions, synthesizing multiple concepts/facts
Processing Speed	Nonverbal learning disorder, auditory processing disorder, SLI, ADHD	Sensory perception, sustained focus/concentration, debugging
Crystallized Intelligence (i.e., language and communication)	SLI, oral expression, listening comprehension	Understanding tasks or instructions, communicating with classmates, expressing confusion or asking for help, formulating an algorithm or explanation, learning technical vocabulary
Reading/Writing	Dyslexia, dysgraphia, reading comprehension, SLI	Understanding written information or instructions, notetaking, comprehending and writing code, interpreting error messages
Decision/Reaction Time and Speed	Nonverbal learning disorders, MID	Comprehending code, debugging

Table 3. The seven broad cognitive abilities associated with abstraction in CS, examples of learning differences associated with those abilities, and the challenges they are likely to pose [15], [16], [19], [20].

In addition to the CHC cognitive abilities, students with learning differences are also likely to struggle with *Executive Functioning* tasks: decision making, planning and organizing, task initiation and completion, self-monitoring, coping with difficulty or frustration, and metacognition (awareness of their own thought processes) [7], [15].

These seven CHC cognitive abilities plus executive functioning encompass the vast majority of cognitive processes associated with computational thinking in general and abstraction in particular. By considering instructional strategies that address common difficulties in these cognitive skills encountered by students with learning differences, we can "minimize the mismatch between the demands of school and the child's personal competence" [6] and give these students a chance to succeed in computer science.

4. Theory to Action: Instructional Strategies

The instructional strategies that follow are rooted in the assumption that "*all* students are capable of excelling academically" [2]. If students with learning differences are adequately supported by the curriculum and pedagogy, they will have a chance to nurture their talents and abilities. It's important to note that the relative newness of computer science in the K-12 setting means that pedagogies for these age groups, disabled and non-disabled, are still under development, and pedagogical "best practices" have yet to be empirically established. As such, when students have trouble with computational thinking, it may be "due to yet-to-be-developed pedagogical practices or to support needs specific to their disability" [6]. To some extent, this distinction is unimportant: universal design for learning tells us that we can find practices that "are essential for some students, beneficial to others, and not detrimental to any" [2]—if a pedagogy is helpful to some and doesn't get in the way of others' learning, it can be considered a good practice, whatever student population it's targeted at. The following instructional strategies are focused on the cognitive abilities they can facilitate. While students with learning differences may be more likely to have more cognitive impairments to a greater degree than their peers, all students have cognitive strengths and weaknesses that can be scaffolded by the appropriate pedagogical approaches.

For practical purposes, I have organized the seven CHC abilities plus executive functioning into four categories with regards to instructional strategies and supports. Section 4.1 addresses memory supports for both short-term and long-term memory. Section 4.2 covers communication supports related to crystallized intelligence and reading/writing abilities. Section 4.3 combines strategies for executive functioning and processing skills, including decision/reaction time and processing speed. Finally, section 4.4 digs into the teaching of fluid intelligence, where the core processes of abstraction take place.

Some of these instructional strategies are common general education or special education practices; some fall under or are offshoots of universal design for learning (UDL) principles; some are derived from CS pedagogical practices; some are novel, or synthesized from multiple sources. All seek to address the specific cognitive challenges associated with computational thinking and abstraction. Framing instructional approaches in relation to the cognitive tasks and disabilities they address allows for a more strategic process of choosing and using these strategies.

4.1 Memory supports

Memory supports can address difficulty with short-term or long-term memory. For vocabulary and technical language, it can be helpful to clearly define new vocabulary, keep a class glossary that is accessible to students at all times, and regularly review terminology to ensure that students understand and remember important terms [1], [7]. Keep technical vocabulary to a minimum, but be precise, and use vocabulary frequently and consistently to help students recognize meanings in context [1]. When teaching new information, list or highlight key points to help students recognize what is most important to remember [9]—color-coding can serve as an additional memory boost.

Maintaining a reference sheet of code/syntax for general use, and encouraging students to make a list to keep track of the variables and functions in each of their programs, can help students who struggle to remember details of code [9]. Anchor charts, posters offering key information in textual and/or visual form, are an additional resource to help students remember important terms, syntax, procedures, or strategies [2]. Glossaries, reference sheets, and anchor charts are a great way to offer students reminders without them needing to ask for help, which helps students maintain autonomy and avoid learned helplessness (as well as freeing up the teacher to focus on more complex needs) [1], [6], [15].

It's also helpful to review important information, including re-teaching as needed [1], [2]. This can be as simple as a quick full-class discussion of terminology or key points at the start of class. Sometimes re-teaching is a whole-class activity, whereas at other times only a small group needs this extra support [2].

Offering reminders, references, and reviews in a variety of formats (per UDL) as discussed above will help ensure that all learners are able to access the information they need in order to engage in class activities.

4.2 Communication supports

Communication challenges are often divided into receptive (incoming) and expressive (outgoing) categories; they can also be categorized by what format—written, spoken, visual, etc.—poses difficulty [15], [19]. Addressing students' communication needs can often be accomplished via UDL, by providing information in multiple formats as well as giving students multiple means of expressing themselves [1], [2], [6], [7]. However, the complexity of computational thinking often creates additional communication barriers for students. The strategies mentioned above for helping with vocabulary are relevant here, and developing class notes or handouts can provide support for students who struggle with notetaking [9].

It can also be helpful to provide students with frameworks for interpreting incoming information or scripts for communicating with others. For example, interpreting error messages is a notoriously challenging task for novice programmers [21], and is even more difficult for students with reading or receptive language disabilities. Providing a diagram with the "anatomy" of an error message, as well as step-by-step instructions for interpretation, would help students develop the ability to decode error messages independently.

Israel et al. [1] share a framework for asking for help developed to facilitate peer collaboration. This series of questions can also be used to ask teachers for help: "What are you trying to do? What have you tried already? What else do you think you can try? What would happen if…?" [22]. An anchor chart bearing this framework can remind students of the script when they need it. Additional scripting tools can be developed as the need arises. This can also be done more informally by "provid[ing] students with prompts and cues" if they are having trouble communicating [3].

4.3 Processing & executive function supports

Supporting students who struggle with processing speed or executive function starts as simple as giving them “more time to process and respond to information” [23]. Allowing for extra time requires planning ahead for how to keep students who work more quickly engaged; offering a bonus exercise or free exploration time can help accommodate differential pacing [2]. These students may also struggle with self-pacing, so monitoring student progress or providing checkpoints during less-structured time can help keep everyone on track [1]. For example, teachers can ask students to come have their algorithm looked over and okayed before they begin coding.

Students with executive functioning or processing difficulties can also be supported by the use of physical, tactile components to a lesson. Manipulatives and other forms of kinesthetic learning can help students with executive dysfunction engage more fully [24], and they can allow students who have visual or auditory processing difficulties take in information kinesthetically, which is the earliest and most intuitive way of interacting with the world [25]. “Unplugged” activities are popular for teaching computational thinking to younger students, and offer a kinesthetic experience that can support students with learning differences [1], [14]. However, research suggests that unplugged exercises can miss the mark and fail to teach students the core principles of computational thinking unless explicit connections are made to the intended concepts [26], [27].

The cognitive effort of maintaining focus or processing stimuli can be exhausting or overwhelming for students with impairments in processing or executive function; allowing students to take voluntary breaks and having a dedicated quiet corner in the classroom gives them a chance to rest and recover [6], [23]. These can also help students who struggle to manage frustration when faced with a difficult task.

Anticipating and scaffolding particularly difficult tasks helps to avoid such frustration and overwhelm in the first place; the scripts and frameworks suggested in the previous section also apply here. Planning, organizing, and decision making can pose particular difficulty for those who struggle with executive function. Examples, templates, or outlines for writing algorithms or code can facilitate these processes [1], [9]. Offer a few options of topics if students have trouble making choices during open-ended tasks [9]. The planning process can also be supported by encouraging students to make sense of the problem and develop an algorithm or outline for their solution before they begin coding, in effect starting at levels 3 and 4 of the PGK hierarchy rather than level 2 [14]—this strategy is discussed further in section 4.4.

Another often-frustrating task, and one that can be particularly challenging for students who struggle executive function or processing, is debugging. Maya Israel offers a series of "debugging detective questions" to help with this: "What happened when I ran my code? What did I want my code to do? Does any part of my code work? Do I know where the problem is in my code?" [8]. These questions prompt students to work on their metacognitive skills, a common weak point for students with executive dysfunction. Teachers can also offer students "options to try when [they run] into trouble" [6], such as checking that variables are spelled correctly, making sure all opening parentheses have matching closing parentheses, and double-checking reference sheets.

It's important to maintain a balance between structured instruction and open inquiry [8]. Scaffolding is crucial to supporting students with learning differences in experiencing success; however, instructors should "fight the urge to over-support," which can lead to overdependence and learned helplessness [6]. Open inquiry gives students the opportunity to engage in productive struggle—an excellent antidote to learned helplessness—while deepening and broadening their understanding of the material [28]. In order for students to engage in productive struggle, they need time to think critically about what they're struggling with [28]. For students who tend to immediately ask for help when they run into problems, refrain from hand-holding and instead direct students to available resources or offer "purposeful questions to help students reflect on the source of their struggle and focus their thinking" [28].

4.4 Fluid intelligence supports

Memory, communication, processing, and executive function are all important cognitive processes in computer science education. At the core of computational thinking and abstraction, however, is fluid intelligence. Fluid intelligence comprises the "mental operations that an individual uses when faced with a relatively novel task that cannot be performed automatically" [19]. Solving a novel computational problem using abstraction is certainly such a task. As previously discussed, abstraction is a notoriously tricky act of cognition, one that students of all ages and abilities struggle with. Armoni's framework for teaching abstraction calls for "an explicit approach" [12], as if accommodating a cognitive limitation most students are known to have—explicit instruction is a common instructional strategy in special education.

In fact, explicit instruction is recommended by many researchers for supporting students with disabilities in computational thinking (e.g., [1], [9], [12], [15], [23]). Computer science instruction often includes considerable time spent on unstructured, open-ended problem solving, which is often challenging for students with learning differences; explicit instruction, in contrast, is "a systematic and direct approach to teaching" and "has been demonstrated as effective for students with learning disabilities and others who struggle with [computational thinking]" [1]. This includes prioritizing big ideas, setting clear goals and expectations, conducting regular reviews of prior learning, modeling with step-by-step demonstrations for students, using simple language and providing definitions for new vocabulary, offering guided practice, closely monitoring student performance, and providing immediate corrective feedback on student work [1]. Several of these strategies have been addressed in previous sections.

Modeling computational thinking practices for students can and should take several forms to provide students with multiple ways of engaging [1], [2]. Running through a "verbal think-aloud [of] coding and decision-making" [3] is perhaps the default form of modeling, but other options include watching video demonstrations or exploring and modifying sample solutions [1], or even physically acting out an algorithm [2]*. Students with memory or processing difficulties benefit from having access to notes and sample work from demonstrations [1]. Modeling offers students an example of how to complete a task, and repeated modeling of similar tasks can help students develop an abstract understanding of the process [9], [14]. Demonstrating the problem-solving process—while offering clear explanations of the reasoning behind and contents of each step—helps students improve their formal logic and complex reasoning capabilities.

Modeling can also support the learning of decomposition [7], a level-3 abstraction task that many students with learning differences find overwhelming. It can be helpful to draw a connection between decomposition and task analysis: the latter is a popular tool in special education, and is effectively decomposition for everyday tasks [3]. Making the connection to a familiar process can encourage students to engage in high-level abstraction. Another useful tool to use when demonstrating decomposition and algorithm design is the humble but mighty flowchart [13]. A flowchart can be used to lay out the parts of a problem in order: this helps bridge decomposition

***A favorite "unplugged" activity is having students write detailed instructions for making a common food item like a peanut butter-and-jelly sandwich or quesadilla, then following up with a live performance of the algorithm(s) [2].*

and algorithms, supports students who struggle with sequencing or cause-and-effect, and provides a visual format for understanding a complex algorithm or solution.

According to Statter & Armoni, modeling should be done “while explicitly demonstrating and referring to” abstraction [14]. It is important to clearly and consistently articulate which level of abstraction is being used and when a move is made from one level to another [12], [14]. This lets students become familiar with the different levels and the roles each level has in the problem-solving process. Using specific language to differentiate between the levels provides students with additional cues—though it is important to keep this language simple and precise, because an excess of technical language can overwhelm rather than clarify [1], [14]. (For example, Armoni uses the term “verbal description” instead of “algorithm” with middle-school students [12].) This is a context where anchor charts can come in handy: a visual diagram of the levels of abstraction, with vocabulary and strategies associated with each, can give students another format to understand the concept and act as a visual cue to actively consider what level they’re working on.

Because students have a “tendency to reduce the level of abstraction,” it is important to start at the highest level of abstraction [12]. Computer science instruction commonly takes place directly in a programming environment (levels 1-2), but starting at the problem and algorithm levels (levels 4 and 3, respectively) helps ensure that students understand the problem more fully before they tackle the details of coding, thus emphasizing the importance of big-picture thinking [14]. Furthermore, returning to the algorithm and problem levels during implementation helps students practice switching between levels of abstraction in ways that serve the problem-solving process [12], [14]. For example, if the code runs without error but doesn’t produce the expected output (level 1), returning to the problem level (level 4) can reveal a misinterpretation of the original question, or returning to the algorithm level (level 3) can reveal a faulty chain of logic.

Making these sorts of practices explicit and regularly modeling them for students helps ensure that students with learning differences have a chance to recognize, understand, and internalize them. Abstraction is “a habit of mind” [12], and as with any new habit, repetition and consistency are key.

Again here, it’s important to avoid the pitfall of over-structuring or over-supporting students: while explicit instruction and modeling are proven instructional strategies, they should be balanced with time for open inquiry, exploration, applications, and productive struggle.

Reflection is a way to reinforce learning and help students understand key takeaways, and also offers instructors a chance to gauge whether students are mastering the material. Armoni’s framework for teaching abstraction recommends ending a computer science course with a reflection of the “major CS principles and methods” covered in the course [12]; explicit instruction and UDL principles suggest more frequent “comprehension checkpoints” to ensure that students recognize the key points of a lesson or unit [1]. These could take the form of full-class discussions or formative assessments, either formal or informal. The instructor then has the opportunity to address challenge points with additional instruction [1], [2].

5. Conclusion

This strategic application of instructional approaches to support students with learning differences in specific cognitive tasks is not novel. For example, Wille et al. developed a set of “guidelines” listing “specific learning and attention deficit disorders and the underlying psychological processes typically associated with them […] that serve as the starting point for adjustments specific to CS instruction and curriculum” [7]. However, this methodical approach is unusual in the research on both CS education in general and CS education for students with learning differences specifically. While some researchers use an iterative curriculum design process to respond to students' difficulties with the material (e.g., [2]), the common approach seems to focus on specific learning or coding platforms or specific curricula. I believe that a methodical, proactive, and responsive curricular design offers students with learning differences the best chance of success—and offers teachers the best chance of avoiding frustration and wasted time. However, empirical research would be required to evaluate the validity of this approach.

The greatest need in research on computing education with students with disabilities is large-scale empirical research comparing different pedagogical strategies. While Armoni's framework for teaching abstraction has shown incredibly promising results in an initial study [14], there are no studies examining its efficacy for students with learning differences. In addition, many of the instructional strategies recommended in section 4 have not been empirically studied in the context of computing instruction. Even though many are known to be effective for supporting students with learning differences in other academic contexts (e.g., see [1], [2], [9]), additional research is needed to determine whether they are similarly effective in supporting computational thinking, as well as to determine whether certain strategies produce better outcomes than others.

Appendix A

Summary of Instructional Strategies

Memory supports (short-term memory, long-term storage and retrieval)

- Vocabulary: define new terms, review terms, maintain a class glossary, use terms frequently and consistently
- List or highlight key points during instruction
- Maintain a reference sheet of code/syntax
- Have students keep a list of variables and functions when coding
- Use anchor charts for important facts, procedures, etc.
- Review and re-teach as needed

Communication supports (crystallized intelligence, reading/writing)

- UDL: provide information in multiple formats, give students multiple means of expression
- Class notes/handouts with key points
- Frameworks for interpreting incoming info, scripts for communicating with others
  - How to make sense of an error message
  - How to ask for help
- Providing prompts and cues if students are struggling to express themselves

Processing (processing speed, decision/reaction time) & executive function supports

- Allow more time for processing and responses (offer bonus exercises or free exploration time to those who work quickly)
- Monitor student progress, build checkpoints into activities

- Physical manipulatives and kinesthetic activities
- Allow students to take breaks
- Offer a quiet area to work or take breaks
- Scaffold difficult tasks
    - Examples, templates, or outlines for the planning/organizing stage
    - Ideas for topics if students get stuck
    - Make sense of the problem and write an algorithm/outline before beginning to code
    - Questions to guide the debugging process
    - A set of options to try when something isn't working
- Maintain a balance with open inquiry: don't over-support, allow productive struggle, ask guiding questions

Fluid intelligence supports

- Explicit instruction
    - Prioritize big ideas, set clear goals and expectations
    - Review prior learning
    - Use modeling and step-by-step demos
    - Monitor student performance and provide immediate corrective feedback
- Modeling with UDL: demos with notes and sample code, video tutorials, exploring or modifying sample code, "unplugged" demonstrations
- Connect decomposition to task analysis
- Use flowcharts for algorithm design
- Levels of abstraction

  - Be clear and explicit
  - Use language to differentiate
  - Anchor chart diagram of different levels
  - Emphasize higher levels of abstraction
  - Be consistent
- Reflection: comprehension checkpoints, class discussion, formative assessments

Appendix B

Sample Lesson Plan

Context

"Thinking Like a Computer Scientist," a computing summer camp program for novice computer science students ages 10-14. The camp will run six hours a day (three in the morning, then three more in the afternoon after an hourlong lunch break) for five days.

Schedule Outline

Monday morning: Intro to CS and CT; Making sense of a problem; 20 Questions activity

Monday afternoon: Algorithms & flowcharts; PB sandwich class activity; Battleship group activity

Sample Activities: Monday afternoon, "Peanut butter sandwich" and "How to win at Battleship"

- *Objectives*: (1) Practice writing detailed steps for a solution; (2) understand how to use flowcharts to outline an algorithm
- *Materials*: Flipchart and flipchart markers, whiteboards and whiteboard markers, scratch paper and writing utensils; sandwich bread, jar of peanut butter[*], plate, butter knife, placemat or butcher paper, and paper towels; Battleship game set
- *Overview*: This lesson has two main parts. First, the instructor will lead the class in a problem-solving activity: writing instructions for making a peanut butter sandwich. The instructor will start on level 4 to make sure students understand the problem before moving to level 3 to outline a solution. As students brainstorm, the instructor will use guiding

[*]*Or almond, sunflower seed, or other butter in the case of a peanut allergy—make sure this is sorted out in advance.*

questions to help them develop a workable solution. The instructor will use a flowchart to outline the students' solution on a flipchart as it comes together, explaining what each structure of the flowchart means. Once the solution is complete, the instructor will ask the students to explain the solution, ensuring that they understand both the solution and the flowchart structure. Finally, the instructor or TA will physically act out the algorithm, mistakes and all! For the second part, the instructor will divide the students into pairs or groups (depending on class size and demeanors) and give them a new problem to solve: writing detailed instructions on how to win at Battleship. Each group will have a large stretch of whiteboard as well as scratch paper to brainstorm on. The assignment is to rephrase the problem and list important considerations, then create a flowchart describing the solution to the problem. At the end, students will share their solutions and give each other feedback on the clarity, functionality, and completeness of their algorithms.

- *Challenge points & instructional strategies*:
    - Many students have not seen or used flowcharts before. This might pose a problem for students with *processing* (especially *visual processing*) impairments, students with *mild intellectual disability*, students with *symbolic or spatial learning disorders*, and others. To address possible confusion, the flowchart will be introduced in the context of solving a problem (modeling), with the solution written out as numbered steps beside the flowchart (multiple formats of presentation), and the instructor will describe and explain what they are doing as they draw the flowchart (explicit instruction). Asking the students to explain the solution acts as a comprehension checkpoint, both giving the instructor a chance to see how well

they understand and giving students a chance to review the concepts and rephrase them.

- Students with *language/communication* or *processing difficulty* may struggle to participate in the full-class problem-solving activity. The problem will be presented aloud, written on the board, and provided on a paper handout (multiple forms of presentation). During discussion, the instructor will intentionally leave some processing time between asking a question and taking responses, and will call on different students to make suggestions. If a TA is available for the class, they could take digital notes during the discussion as a record and additional format for students to absorb the information. In addition, during the group work all groups will work on the same activity, so that students don't need to rapidly make sense of a new problem during the final discussion.
- Some students may not have encountered the game Battleship before, or may not remember exactly how it works. There will be a physical copy of the game for students to look at and refer to during the Battleship activity (multiple formats of information; manipulatives), and if students who aren't familiar with the game are having trouble making sense of it, the instructor or TA can lead a game to demonstrate how it works (modeling).
- Students with *communication impairments* might have trouble with the collaborative structure. The instructor will try to choose pairs/groups of students with similar temperaments to keep quieter or slower-processing students from

being overwhelmed by louder or more exuberant classmates[*]. In addition, information from the morning will be provided on handouts, including scripts to guide communication and problem-solving.

- Students with *executive dysfunction* may find the open-ended structure of the second activity challenging. The instructor will supervise the students during this stage, and if a student or group seems stuck, the instructor will ask guiding questions, provide prompts, and direct the students towards available resources (scaffolding). In addition, if a student seems to be getting frustrated, the instructor will check in with them and either provide additional guidance or encourage a break if it seems warranted.
- Students with *memory impairments* might have trouble remembering the material from the morning; paper and digital handouts and flipchart notes from the morning activity will help serve as reminders. Students who are having trouble keeping track of all the parts of the problem will be encouraged to take notes in whatever form works for them, with some possible notetaking formats suggested for students who don't have a preferred strategy.

[*]*The question of similar-ability versus mixed-ability grouping is a controversial one, and beyond the scope of this paper. For this early stage of the camp, placing students with similar classmates is intended to prevent overwhelm and foster equal participation.*

References

[1] M. Israel, Q. M. Wherfel, J. Pearson, S. Shehab, and T. Tapia, "Empowering K-12 Students With Disabilities to Learn Computational Thinking and Computer Programming," *Teaching Exceptional Children*, vol. 48, no. 1, pp. 45–53, Oct. 2015, doi: 10.1177/0040059915595790.

[2] A. K. Hansen, E. R. Hansen, H. A. Dwyer, D. B. Harlow, and D. Franklin, "Differentiating for Diversity: Using Universal Design for Learning in Computer Science Education," presented at the SIGCSE, Memphis, TN, Mar. 2016, doi: 10.1145/2839509.2844570.

[3] E. C. Bouck and A. Yadav, "Providing Access and Opportunity for Computational Thinking and Computer Science to Support Mathematics for Students With Disabilities," *Journal of Special Education Technology*, pp. 1–10, Dec. 2020, doi: 10.1177/0162643420978564.

[4] "The Condition of Education: Students With Disabilities," National Center for Education Statistics, May 2020. Accessed: Apr. 14, 2021. [Online]. Available: https://nces.ed.gov/programs/coe/indicator_cgg.asp.

[5] N. A. Gage, K. S. Lierheimer, and L. G. Goran, "Characteristics of Students With High-Incidence Disabilities Broadly Defined," *Journal of Disability Policy Studies*, vol. 23, no. 3, pp. 168–78, 2012, doi: 10.1177/1044207311425385.

[6] M. R. Snodgrass, M. Israel, and G. C. Reese, "Instructional supports for students with disabilities in K-5 computing: Findings from a cross-case analysis," *Computers & Education*, vol. 100, pp. 1–17, Apr. 2016, doi: 10.1016/j.compedu.2016.04.011.

[7] S. Wille, J. Century, and M. Pike, "Exploratory Research to Expand Opportunities in Computer Science for Students with Learning Differences," *Computing in Science & Engineering*, vol. 19, pp. 40–50, Jun. 2017, doi: 10.1109/MCSE.2017.43.

[8] M. Israel, "Teaching computer science to K-8 students at risk for academic failure: Research findings and implications for practice," Lynne and William Frankel Center for Computer Science, Department of Computer Science, Ben Gurion University of the Negev, Beer Sheva, Israel, Jul. 2016, Accessed: Apr. 09, 2021. [Online]. Available: https://ctrlshift.mste.illinois.edu/files/2013/08/CS-for-All-slides-July-2016-FINAL.pdf.

[9] Outlier Research & Evaluation, “Teaching Practices Guide: Improving Accessibility for Students with Learning Disabilities & ADHD: The Computer Science Principles (CSP) Course.” Accessed: Apr. 06, 2021. [Online]. Available: outlier.uchicago.edu/accessCSP/.

[10] M. Israel and L. A. Delyser, “It’s About the All: The Role of Including Students with Learning Disabilities in Computer Science Education,” Oct. 04, 2019.

[11] “K-12 Computer Science Framework,” 2016. Accessed: Apr. 09, 2021. [Online]. Available: http://www.k12cs.org/.

[12] M. Armoni, “On Teaching Abstraction in Computer Science to Novices,” *Journal of Computers in Mathematics and Science Teaching*, vol. 32, no. 3, pp. 265–84, 2013.

[13] J. Valenzuela, “How to develop computational thinkers,” *International Society for Technology in Education*, Sep. 22, 2020. https://www.iste.org/explore/how-develop-computational-thinkers (accessed Apr. 12, 2021).

[14] D. Statter and M. Armoni, “Teaching Abstraction in Computer Science to 7th Grade Students,” *ACM Transactions on Computing Education*, vol. 20, no. 1, p. 8:1-8:37, Jan. 2020, doi: 10.1145/3372143.

[15] “Characteristics of Children with Learning Disabilities,” National Association of Special Education Teachers, 3, n.d. Accessed: Apr. 25, 2021. [Online]. Available: https://www.naset.org/fileadmin/user_upload/LD_Report/Issue__3_LD_Report_Characteristic_of_LD.pdf.

[16] Learning Disabilities Association of America, “The Ins and Outs of Learning Disabilities,” Sep. 25, 2013. https://ldaamerica.org/info/the-ins-and-outs-of-learning-with-ld/ (accessed Apr. 12, 2021).

[17] J. Perrenet and E. Kaasenbrood, “Levels of Abstraction in Students’ Understanding of the Concept of Algorithm: the Qualitative Perspective,” Bologna, Italy, Jun. 2006, vol. 3(2), pp. 270–74, doi: 10.1145/1140124.1140196.

[18] J. Perrenet, J. F. Groote, and E. Kaasenbrood, “Exploring Students’ Understanding of the Concept of Algorithm: Levels of Abstraction,” Caparica, Portugal, Jun. 2005, vol. 37(3), pp. 64–68, doi: 10.1145/1151954.1067467.

[19] D. P. Flanagan and S. G. Dixon, “The Cattell-Horn-Carroll Theory of Cognitive Abilities,” *Encyclopedia of Special Education: A Reference for the Education of Children, Adolescents, and Adults with Disabilities and Other Exceptional Individuals*. John Wiley &

Sons, Jan. 22, 2014, Accessed: Apr. 25, 2021. [Online]. Available: https://onlinelibrary.wiley.com/doi/full/10.1002/9781118660584.ese0431.

[20] *Individuals with Disabilities Education Act*. 20 U.S. Code § 1400. 2004.

[21] "Programming Languages and Learning: A quick primer on human-factors evidence in programming language design," *The Quorum Programming Language*. https://quorumlanguage.com/evidence.html (accessed Apr. 15, 2021).

[22] M. Park and T. Lash, "The Collaborative Discussion Framework." Ctrl-Shift, Apr. 03, 2015, Accessed: May 07, 2021. [Online]. Available: https://ctrlshift.mste.illinois.edu/2015/04/03/collaborative-discussion-framework/.

[23] S. Wille, "Improving Accessibility in Computer Science for Students with Learning Disabilities and Attention Deficit Disorders (Part 3 of 3)," *CS for All Teachers*, Feb. 03, 2017. https://www.csforallteachers.org/blog/improving-accessibility-computer-science-students-learning-disabilities-and-attention-1 (accessed Apr. 06, 2021).

[24] A. Anderson and R. Rumsey, "Channeling energy using bodily-kinesthetic intelligence: helping children with ADHD," *Physical & Health Education Journal*, vol. 68, no. 3, 2002, Accessed: May 10, 2021. [Online].

[25] S. McLeod, "Piaget's Theory and Stages of Cognitive Development." Simply Psychology, Dec. 07, 2020, Accessed: Apr. 12, 2021. [Online]. Available: https://www.simplypsychology.org/piaget.html.

[26] Y. Feaster, L. Segars, S. K. Wahba, and J. O. Hallstrom, "Teaching CS Unplugged in the High School (with Limited Success)," presented at the 16th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education, Darmstadt, Germany, Jun. 2011, doi: 10.1145/1999747.1999817.

[27] R. Taub, M. Ben-Ari, and M. Armoni, "The Effect of CS Unplugged on Middle-School Students' Views of CS," in *ACM SIGCSE Bulletin*, Jul. 2009, vol. 41(3), pp. 99–103, doi: 10.1145/1595496.1562912.

[28] S. D. Lynch, J. H. Hunt, and K. E. Lewis, "Productive Struggle for All: Differentiated Instruction," *Mathematics Teaching in the Middle School*, vol. 23, no. 4, pp. 194–201, Feb. 2018.