Pakistan Journal of Scientific Research
Multidisciplinary & Peer Reviewed
Volume: 4, Number: 2(Suppl.), Pages: 26-41, Year:2025

Article

The Disappearing Struggle in the Age of AI: Re-Evaluating Pedagogy of Algorithms and Problem Solving in Tool-Augmented Learning Spaces

Eman Kamran[1], Unnus Ahmad Usmani[1], Zain Zoy[1], Faiqa Anwar[1,] * and Rabranea Bqa[1]

[1] Forman Christian College, Department of Computer Science, Lahore, 54000, Pakistan; 261956459@formanite.fccollege.edu.pk
* Correspondence: 261956459@formanite.fccollege.edu.pk

Submitted: 20-06-2025**, Revised:** 10-08-2025, **Accepted:** 15-08-2025

Abstract

The problem of the boundary in algorithmic learning that has preoccupied scholarly and practical discussion since the inception of generative AI platforms such as ChatGPT and GitHub Copilot seems to be drifting into the past. In this paper, the study conceptually examines the issue of tool-augmented environments by addressing the choice and degree to which students can bypass the kind of cognitive challenge that in the past was required to achieve the sophistication in recursive logic, complexity analysis or optimization strategies. The introduction to the paper shifts the narrative of AI negatively by conceptualizing a pedagogical re-imagination where AI would not be positioned as an existential threat, but as a co-reasoning partner instead. Divide and Conquer, Sorting, Space and Time Complexity, Dynamic Programming, and Greedy Algorithms case studies are provided, as well as a set of supporting educational interventions that will enable driving critique, reconstruction, and metacognition over time. The notions of recursion, fault injection, prompt engineering, and constraint-based tool design can be viewed as an illustrative example of how productive struggle can be maintained in AI-assisted environments. Focused on such integration of empirical inquisition, the theory of learning, and design-based thinking, the paper proposes a new vision: a vision where automation supplements rather than replaces conceptual mastery, where algorithmic thinking is deeply human, critical, and creative.

Keywords: AI-assisted learning; Algorithms pedagogy; Problem solving; Tool Augmented Learning; Computational thinking; Complexity analysis; Recursive algorithms; Dynamic programming.

1. Introduction

We're in a moment where writing code—or even understanding how it works—is no longer a prerequisite for getting correct results. With tools like ChatGPT, Copilot, and algorithm visualizers becoming standard in students' workflows, many can bypass the slow, frustrating steps of learning to design or debug algorithms altogether. That might sound like a win. But it raises a serious question: what happens when students stop struggling with the material? In the past, the effort to understand

recursive calls, optimizing for time, or structure divide-and-conquer strategies wasn't just about getting the right answer; it was where real learning actually happened. Algorithmic thinking was built through discomfort and struggle. And when AI fills in the gaps too early, that experience of grappling with the problem is what quietly disappears.

This paper argues that we don't need to abandon AI tools—but we do need to rethink how they're used in algorithm education. We focus on six key areas—Divide and Conquer, Sorting, Time Complexity, Space Complexity, Dynamic Programming, and Greedy Algorithms—to explore how AI affects both what students learn and how they learn it. Our goal is to move beyond binary thinking—that is, whether AI is good or bad—and instead offer concrete models that bring back productive struggle in meaningful ways. These include letting students critique AI-generated solutions, reconstruct broken logic, or compare different recursive strategies. If done well, tool-assisted learning can push students toward deeper understanding—not shortcut it.

2. Literature Review

To have an idea of the changing role of artificial intelligence in computer education, it is necessary to understand more than simply having a theoretical knowledge of artificial intelligence mathematics- one must also understand and then examine patterns, research methodologies and debates in computer-assisted learning. As demonstrated in the early [16] and today (via interactive visualizations and prompts designed by AI, the paper comments on both substantial progress and persistent challenges en route to aligning the AI technologies with meaningful outcomes.

The foundations of computer-based problem-solving have in the past been underpinned by hardwired mathematical rules. [16] It is considered one of the cornerstones works that focus on logical methodologies suitable to solve problems called proper reasoning, which is a study that has advancements as a paradigm of computational thinking. [13] has articulated similar sentiments where they argue in favour of the perspective of space complexity, where there would be an academic balance between the limits of memory and time efficiency of our algorithms, all else being equal, which is of particular interest when teaching dynamic programming and recursion in depth. Furthermore, we have also acquired the idea of conditional teaching size because it informs us of the minimum of interaction needed to generalize situation-sets in the learning of students [11]. Although these theoretical developments can give good guidance on how AI-driven instruction can be beneficial, we are still struggling to understand how the learners actually view and understand it internally, since everyone learns in unique ways.

The paper [18] pointed out that the direct answers and perfect code samples have deteriorated the analysis and ability of the learners to comprehend the code logic. As [14], [15] has identified that dpvis, a visual dynamic programming tool, only productive gains occurred when there was some form of visual aids incorporated alongside well-organized prompts. All these findings have demonstrated similar findings, by presenting that AI tools could provide immediate, correct responses, but not be able to do something such as improve a critical aspect of a learner's ability.

Recent research in education and AI has suggested that applying mixed methodology might be a good solution, integrating interactive and performance data. We know that [12] Provided quiet help to users through learning games or simulations, then analyzing the log data and conducting interviews to track their conceptual navigation. While these practices provide us with valuable insights, there is still a major gap in analyzing whether they lead to conceptual mastery and to be able to apply knowledge in new contexts.

Studies have shown better results without considering the actual factor of whether students have actually built the ability to reason, or are they just following patterns. [20] mention that learners barely take on the underlying logic that AI uses to develop correct

and sound answers. [22] has also pointed out the lack of extensive data or primary research conducted on students to see how responsible they are to learn independently. Fixed images and questions can make students just copy instead of thinking. As [17] highlights, true engagement requires interaction.

AI is redesigning the learning environments for code development, moving from inflexible to adaptive, tool-assisted learning. While theory and short-term studies encourage its use, long-term effects on cognitive reasoning are still unclear. For students to actually benefit from these AI tools, we need to pair them with strategies that not only build but also enhance critical thinking abilities.

3. Basic Sequence for algorithms and problem-solving Curriculum in tool-augmented spaces

For the teaching of algorithms in an AI-assisted environment, a reframed curricular arc now emphasizes conceptual labour while acknowledging the realities of tool reliance. This section outlines a restructured sequence for algorithm instruction, grounded in six interdependent areas: Divide and Conquer, Sorting, Complexity (Time and Space), Dynamic Programming, and Greedy Algorithms. Each of these areas is approached not simply as a computational method, but as a pedagogical space where automation can either dilute or deepen student understanding. We want to be clear: our goal is not to introduce abstract content in isolation, but to create opportunities for students to critically engage with intelligent tools instead of memorizing or passively consuming outputs. In the following sections, we reflect on the disappearing struggles traditionally tied to learning through design, and offer practical interventions—such as prompt engineering, fault injection, visualization, and peer-based reconstruction—to help reposition reasoning at the center of algorithmic learning. Together, these sections point toward an instructional model where conceptual understanding and intelligent tooling are not at odds, but co-designed to produce richer learning experiences.

3.1. Recontextualizing Divide and Conquer in Tool-Aided Problem Solving

Divide and Conquer (D&C) is a classic algorithmic paradigm that invites learners to break down a problem into smaller subproblems, solve each recursively, and then merge the solutions—think Merge Sort, Binary Search, or the classic closest-pair problem. When taught traditionally, students manually construct recursion trees, identify base and merge cases, and analyze stack growth. Visualization tools, paper drills, and handwritten dry runs build a robust mental model of recursion's structure and behaviour.

This can be not only cognitively intense, but also very educational when done correctly. For example, when students work through Merge Sort, they are not only writing code, they are also visualizing how the sub-arrays are merged, at what point the stack depth fluctuates, and where the behaviours of the runtime change. Thinking recursively in all contexts will foster algorithmic intuition, enabling deeper learning beyond syntax. This prepares students for less common techniques too, like Karatsuba multiplication or parallel divide and conquer scheduling [2], both of which deepen recursive reasoning and enable more efficient algorithmic performance. Critically, divide-and-conquer doesn't just structure recursive solutions—it forms the architectural backbone of scalable parallel algorithms. As Blelloch illustrates in his foundational work on parallel programming, D&C patterns make it possible to reason about both work and depth in recursive systems, enabling students to transition from sequential logic to massively parallel thinking [3].

However, visualization alone over time was inadequate when used passively. Studies like Al-Thobhani et al demonstrated that when students actively created visualizations for recursive execution, they increased their problem-solving accuracy and improved their depth of thinking when compared to their peers who only viewed visualizations [1]. Their work highlights the pedagogical importance of constructive learning over passive observation.

Students in today's classrooms typically bypass the experience of reconstructing recursion. As AI tools such as ChatGPT or coded assistants create complete D&C implementations instantaneously, students get correct code but often without understanding the interaction between recursion, base cases, and merge logic. Essentially, the learner's role has evolved from being an architect to that of a cross between a user and spectator.

To counter this and reframe D&C pedagogy, instructors must reacquire the mantle of critical thinking over code generation. Students could be challenged by asking questions like:

1. What does the recursion tree tell us about the stack depth and spatial overhead?
2. How would the pivot or division strategies differ in edge situations?
3. If the merging phase fails on duplicates or skewed input, where exactly and why does it fail?

Asking such questions restores cognitive tension to the task and stops blind faith in AI-scripted code. To illustrate this in action, below are examples of learning activities:

1. Recursion Fault Injection: Provide an AI-generated version of Merge Sort that improperly handles duplicate elements (or any non-integer inputs). Students find the bug and correct the merging logic or base-case condition.
2. Variations on the Divide Strategy: Give students a template D&C algorithm, and have them play with different divide sizes, such as using 3-way partitioning instead of dividing in halves, etc., and reflect on how the complexity and memory behaviour change.
3. Recursion Tree Investigation: Provide an AI-generated D&C implementation of Binary Search or FFT. Have the students illustrate the recursion tree and annotate the usage of the stack in relation to the tree, and then compare it to the actual runtime output for N-element inputs.
4. Collaborative Small-Group Reconstruction: Use the "Divide & Conquer" peer-teaching method developed by Samsa and Goller [3]. Place the students in small groups, have each student reconstruct a section of the recursive algorithm (divide logic, base-case, combine step, etc.) and then explicate it to colleagues using collaborative slides or other visual aids.

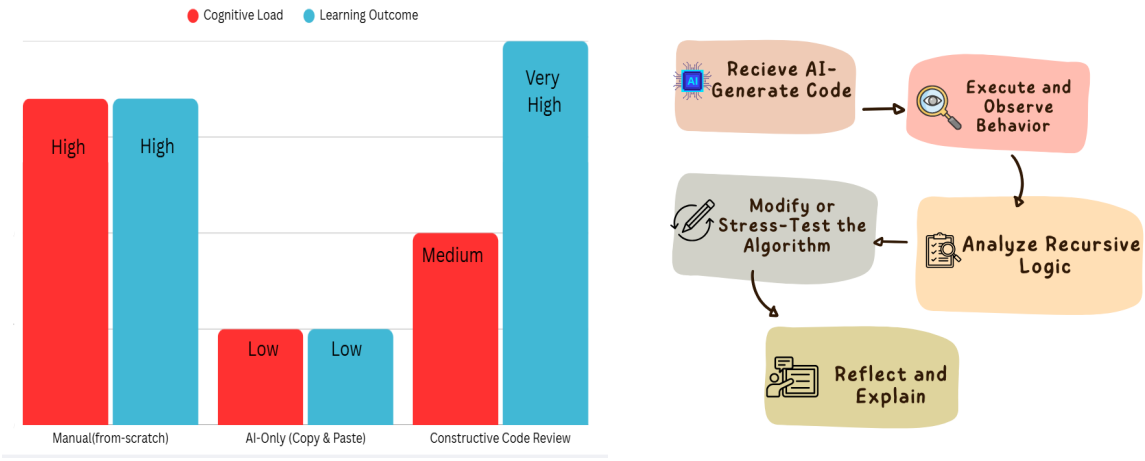
Table 1. Pedagogical Shift in Divide & Conquer Strategy

Traditional Pedagogy	AI-Augmented Pedagogy	Future-Oriented Practices
Draw recursion tree manually	AI generates recursive implementation	Critique recursion structure and divide-merge logic
Hand-coded Merge Sort	Copy AI-generated Merge Sort	Predict failure cases; reverse-engineer logic
Label base-case correctness by	Accept AI's default base-case choice	Test custom base-case variations
Write code from scratch	Rely on templates	Intentionally leave holes for student reconstruction
Simulate recursion mentally	Trust output	Compare recursive runtime behavior with reality

Table 2. Common AI-Generated Divide & Conquer Mistakes and Fixes

Error Type	AI-Augmented Pedagogy	Future-Oriented Practices
Missing base case	Infinite recursion	Manually add base case + test inputs
Incorrect merge logic	Copy AI-generated Merge Sort	Dry-run with sample arrays

Stack overflow with large N	Overwrites elements or fails on duplicates	Test custom base-case variations
Write code from scratch	Recursive depth too deep	Switch to tail recursion or iterative
Hardcoded array sizes	Poor generalization	Replace constants with dynamic lengths



(**a**) (**b**)

Figure 1. (**a**) Comparing Cognitive Load and Learning Outcomes in Manual, AI-Assisted, and Critique-Based Divide & Conquer Learning. (**b**) From Copying to Understanding: A Smarter Way to Learn Divide & Conquer with AI Codes.

3.2. Reframing Sorting Algorithms for Tool-Aware Learners

Sorting algorithms have been more than just code for execution; they provide learners a pathway towards computational thinking. They are designed to build deeper understanding, such as loops, conditionals, recursions, and memory usage. If we look at the brute force nature of Bubble Sort or the recursive nature of Merge Sort, we see that these algorithms help learners grasp time complexity, memory use as well as algorithmic design.

What makes sorting such a powerful teaching tool is the way it allows learners to mentally simulate what code is doing, step-by-step. For example, by tracing the nested iterations in Selection Sort, students begin to internalize what $O(n^2)$ behaviour looks like in practice. With Merge Sort, they begin to see how recursive function calls translate into stack frames, and how local solutions are stitched into global ones. This type of internal modelling—where students don't just run a program but visualize and reason through its structure—is a crucial part of learning to think like a computer scientist.

Research has shown that this modelling becomes more effective when learners make their own representations of algorithms, like making their own visualizations instead of watching other animations. In a Cetin and Andrews-Larson study, students who built their own representations of sorts of algorithms had better conceptual understanding than students who viewed pre-built ones. [4] This is fundamental for understanding how sorting pedagogy is changing in the AI era.

In today's world, that traditional struggle to "figure out" how sorting actually works can almost be avoided entirely. With the emergence of AI tools like ChatGPT, GitHub Copilot, and LeetCode's smart code explainers, students can now generate working versions of Insertion Sort, Quick Sort, or even Radix Sort, with a single prompt. The output

is often not only syntactically correct, but also neat, optimized and described. What would take students hours of consideration and debugging can now be served up in seconds, and while this productivity gain is certainly helpful, it creates a very different relationship to the material for that student. Instead of understanding the "why" and "how" behind the logic, students risk understanding only the "what."

This shift turns students from problem solvers to users of prompts, constructors of knowledge to consumers. As Holstein and Aleven argue, the successful implementation of AI in learning should not replace human cognition; rather, it should enhance cognition—helping students do more, but not doing all the work for them. [5] If the AI completes the entire problem-solving process for learners, then learners will not have to experience the cognitive struggle, which is necessary for building mental models.

To start facing this reality, we're going to have to shift our perspective on sorting. The new focus is less on writing algorithms from scratch, or themselves and more on critically thinking about the structure and strategy behind sorting algorithms. We should be getting students to ask things like: Why does Merge Sort have O(n) space? Why is Quick Sort unstable? How can Quick Sort be stable? How does Radix Sort work on strings or negative numbers? How do we weigh clarity, maintainability, and performance of AI-written code?

If we create classroom activities that recapture this complexity of analysis, we're keeping the richness of teaching sorting. Rather than disrupting a pedagogical conception where generative AI could produce simplified outputs, we're producing hybrid, tool-savvy learners who can reason through code even when it isn't co-created.

Table 3. Comparison of Sorting Algorithms in Tool-Aided Learning Context

Traditional Pedagogy	AI-Augmented Pedagogy	Future-Oriented Practices
Tracing swaps and comparisons	Direct code from AI tools	Debugging and analyzing tool outputs
Manual step-by-step execution	Output-first thinking	Encouraging critical failure testing
Learning via iterative refinement	Copy-editing generated code	Exploring design variations (e.g., pivot, gap sizes)
Comparing multiple algorithms	Choosing one “best” from AI	Analyzing trade-offs across datasets
Debugging errors to learn logic	No bugs in AI outputs	Reverse engineering and intentional editing

Sample activities to incorporate meaningful struggle:

1. Comparing and Analyzing: Show two AI-generated versions of Bubble Sort. Students identify conditional differences or control flow differences and rate them as correct or efficient.
2. Context-Based Selection: Show students an almost sorted dataset. Ask them to select Insertion Sort or Quick Sort and write a short justification based on input characteristics.
3. Heap Sort Adaptation: A standard min-heap implementation of Heap Sort is provided. Challenge students to modify it so it sorts in descending order, and explain the effect on the heap structure.
4. Radix Sort Discussion: Give an AI tool a generic Radix Sort implementation. Students analyze its handling of non-integer data types and suggest improvements for broader data compatibility.

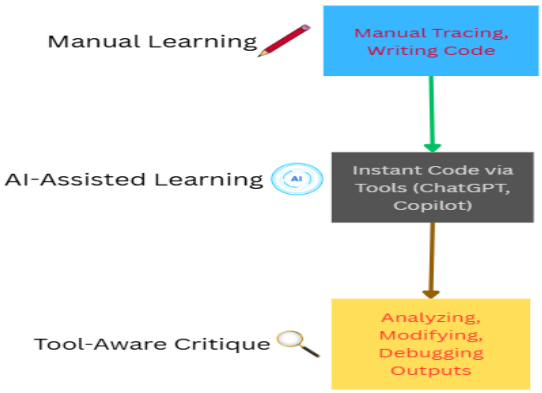


Figure 2. Transition in Student Engagement: From Manual Implementation to Tool-Aware Critique.

Algorithm	Stable?	Average Time	Worst Time	Space	Tool Output Accuracy
Bubble Sort	Yes	$O(n^2)$	$O(n^2)$	O(1)	High
Insertion Sort	Yes	O(n)	$O(n^2)$	O(1)	High
Merge Sort	Yes	O(nlog)	O(nlog)	O(n)	Medium
Heap Sort	No	O(nlog)	O(nlog)	O(1)	Medium
Quick Sort	No	O(nlog)	$O(n^2)$	O(nlog)	High

Table 4. Comparison of Sorting Algorithms in Tool-Aided Learning Contexts

As sorting algorithms become more accessible through AI, they must also become more analytical in their teaching. The role of educators is no longer to guide students through implementation alone but to help them develop insight into structure, performance, and adaptability. By encouraging critique, exploration, and variation, sorting algorithms can continue to serve as a rich learning ground — even in an age where code is just one prompt away.

3.3. Beyond Big-O: Reconstructing Complexity Awareness in Tool-Embedded Learning

When we consider the concept of time complexity not merely as a technical artifact but also a mathematical and conceptual filter, we can see how we have delegated to tools so much that used to require direct mental agitation. In a generation where Copilot auto-completes sorting algorithms, ChatGPT explains Big-O in a paragraph in a way that lacks analytical rigour or details that might be needed for deep understanding, and where the internet runs on algorithm-based neural net models, the problem has become bigger than just misinformation. It is gradually creating a space between us and our analytical skills. The process of performance struggle, the space, and the execution time in question are fading away before us, while we are busy getting our tasks completed, we are losing our abilities. In this paper we can, we will review how the academic basis of computational complexity is being undermined in tool-augmented learning environments, and how we can reboot these tools in a way that preserves the space for developing analytical awareness of complexity, not by withdrawing the tools, but by reimagining how they are used.

3.3.1. The Erosion of Complexity Intuition

Traditionally, mastering the algorithmic complexity was an initiation rite. It made a difference between functional programmers and computational thinkers. Nested loops, recursion trees, and asymptotic notation were not learned by students simply to take

exams but to develop an intuitive feel of how we analyze performance. However, as AI-based assistants are on the upswing, the performance factor dwells beyond the realms of concern. As [6] remarks, it has now become common to find students who complete tasks without engaging in the process of reflection on how a task was completed in terms of algorithm complexity [8]. Also, clarify that these tools allow achieving a successful code generation, yet lower the capacity of students to assess trade-offs in performance.

We can visualize this shift in the following table:

Table 5. Human-Coded vs AI-Generated Tasks

Task Type	Human-Coded (Traditional)	AI-Generated (Tool-Aided)
Sorting Algorithm	Selects algorithms (e.g., Merge Sort) based on data size and complexity.	Uses default '.sort()' with unknown mechanism.
Graph Traversal	Chooses BFS/DFS based on use-case and complexity analysis.	Outputs traversal without explaining performance trade-offs.
Recursive Optimization	Evaluates time-space tradeoffs, chooses recursion or DP appropriately.	Often selects recursion without commenting on complexity.

This is not to argue that tools are inherently educationally harmful. On the contrary, [18] shows that students often feel empowered by AI-generated solutions — but the empowerment is syntactic, not semantic. The tool helps them write, but not necessarily think. As a result, students may arrive at a solution without forming an internal compass of complexity. Not just this many times, the answers generated by AI may seem real or correct, but rather, they are not correct, often called hallucinations. That means tools like ChatGPT and Copilot make up stuff to cover many gaps which are not real. This can create misinformation and confusion among students who use ChatGPT as a primary source of information.

3.3.2. Why Complexity Still Matters

There could be an argument that one side would say: A tool that works, why not keep it simple? The solution is magnitude and durability. In a meta-analysis of 35 empirical studies conducted by [9], tool-assisted learning was shown to increase speed and produce results of higher correctness, but it shows no correlation with conceptual understanding in cases related to complexity. This implies that whoever walks out of an algorithms course will be able to write code, but not have the wherewithal to reason about algorithm choice when faced by a programming task with memory limits, distributed systems, or real-time constraints.

More recent studies provide even more in-depth information on this. A meta-analysis of 51 articles [10] exhibited a substantial effect size of AI tools on basic learning performance ($g = 0.867$), a moderate effect on higher-order thinking ($g = 0.457$), and a critical reflection ($g = 0.456$). In another research, GenAI users performed 6.7 points less than non-users in regular exams. Most importantly, cognitive offloading is strongly and negatively related to critical thinking ($r = -0.68$), whereas the EEG study at MIT Media Lab finds much weaker engagement of the brain when solving problems with the help of AI.

Complexity awareness is also tied to a student's capacity for innovation. Without understanding why, a brute-force approach fails at scale, students are unlikely to arrive at optimized solutions or to think critically about algorithmic ethics, energy consumption, or performance under constrained hardware.

Table 6. Empirical Gaps in Tool-Aided Complexity Learning

Measure	AI Tool User	Non-Users (Baseline)	Source
Learning Performance (g)	+0.867	Baseline	Meta-Analysis by Wang & Fan (2025) over 51 studies
Higher-order Thinking (g)	+0.457	Baseline	Meta-Analysis by Wang & Fan (2025) over 51 studies
Critical Thinking (r)	–0.68	Baseline	MDPI Cognitive Offloading Study (2024)
Neural Engagement (EEG)	Lowest	Highest	MIT Media Lab EEG Study (2023)

To underscore this point, consider the following conceptual figure:

Figure 3.

The cognitive terrain has shifted: students now climb less but arrive faster — though they may not know what mountain they were on.

3.3.3. Reconstructing Pedagogical Approaches

Then how does one reassemble the knowledge of complexity without tool bans? The solution consists of a pedagogical design that uses tools not as shortcuts but as co-instructors. Think of AI coding tools that render the Big-O complexity of the code they write. Or evaluation systems that encourage students to criticize the effectiveness of an AI solution. We do not plan to get rid of AI assistance, but represent metacognitive prompts and performance dashboards with the purpose of training students to challenge the output. [7] propose a framework for scaffolding AI-assisted learning, where tools are paired with reflection checkpoints. Extending this, we propose a three-layer complexity-awareness design:

Table 7. Pedagogical Layers for Rebuilding Complexity Awareness in AI-Rich Environments

Layer	Strategy	Example Feature
Immediate Feedback	Real-time complexity estimates during code generation	Tooltip: "Estimated Time: O(n log n)"
Reflective Prompting	Ask students to evaluate AI's algorithmic choice	Quiz: "Is this the best approach for n > 10^5?"
Comparative Analysis	Present two solutions with trade-offs	Heatmap: Time vs Space vs Readability

We aren't trying to reverse time or de-AI the classroom. We're asking: What kind of learner emerges when convenience is no longer a cognitive crutch but a springboard for inquiry?

Questioning computational performance calls upon the educator to develop, not to jettison, asymptotic thinking. When learning is surrounded by an atomic generation of answers given by tools, it is vital to keep performance analysis at the center of the course. Complexity awareness Redesigning education would help solve the problem of complexity awareness without any hint at a backward journey to the past.

3.4. Repositioning Space Complexity as Concept: Not Computation

3.4.1. Repositioning Space Complexity as a Concept: Not Computation

Space complexity is given secondary consideration in algorithm teaching--as a topic taught after time complexity and as a parameter to be optimized. What pupils are taught is to compute stack frames or memory usage, but hardly why there is structure-bound space consumption. The importance of working in space as a design problem has dwindled even further in the age of AI-assistive technologies, where solutions are available in real-time with the help of such tools as GitHub Copilot and ChatGPT.

To do so, relying on criticisms presented by [13], [11], [12], this area suggests that the notion of space complexity might as well be shifted towards being conceptually restrictive, a way of thinking and abstraction applicable to the design of algorithms, rather than being a measure of effectiveness.

3.4.2. The Metric-First Framing Problem

The space complexity is usually seen as abstruse or irrelevant by students, not because they are incapable of understanding it, but because these concepts are presented without any context that relates them to structure. According to [13], it is mostly simplified to formulaic analyses such as O(n), and it is not associated with design logic. The CS Socially-Just Worlds critique continues by stating that the excessive focus on optimization impairs reflection and access.

The authors in [11] introduce the notion of conditional teaching size, that is, concepts can be comprehended more readily when the internal structures of these concepts are laid out. The complexity of teaching space should thus be more conceptually profound and design-conscious than memory calculations.

Table 8. Complexity Transition of Metric to Mental Model of Shifting Space

Element	Traditional Framing	Conceptual Reframing
Role in Curriculum	Efficiency metric post-solution	Design constraint that shapes abstraction
Student Engagement	Memorization, surface-level	Exploration, structural reasoning
AI Tool Behavior	Bypasses or hides space	Surfaces and explains space-based decisions
Learning Outcome	Faster code	Deeper problem modeling and trade-off analysis

3.4.3. Conceptual AI Tool Design: Examples and Integrations

1. Space-Aware Code Suggestions

   Real-time code generation programs (like GitHub Codelike, which finishes code according to the circumstances) are getting rampant amongst students. The

tools can be augmented so as to provide space-aware tooltips, or overlays, which can explain the memory impact of a specific implementation.

To illustrate a typical example, when a student writes a recursive feature the tool would possibly present an elicit such as:

This is a call which consumes O(n) of the stack. "See Proposed O(1) usage of iterative refactoring."

This metacognitive micro-intervention transforms the AI into a thinking guide that is an auto-completion engine. It helps students develop critical thinking about memory trade-offs, and it helps enforce design awareness even in the very process of coding.

2. Visual Stack and Heap Maps

Learning sites, such as LeetCode and VisuAlgo, may be augmented with live visualizations showing the use of memory to execute algorithms. These tools help students form a concrete spatial intuition of space complexity by exhibiting space growth or heap allocation in real time, as opposed to abstract notation only.

This graph is a comparison of the memory used in two implementations of DFS. Recursive DFS exhibits a linearly growing depth of the traversed stack with increased input, whereas iterative DFS has a flat memory profile. The visualization also brings the behaviour of space to touch and empowers the structural realization.

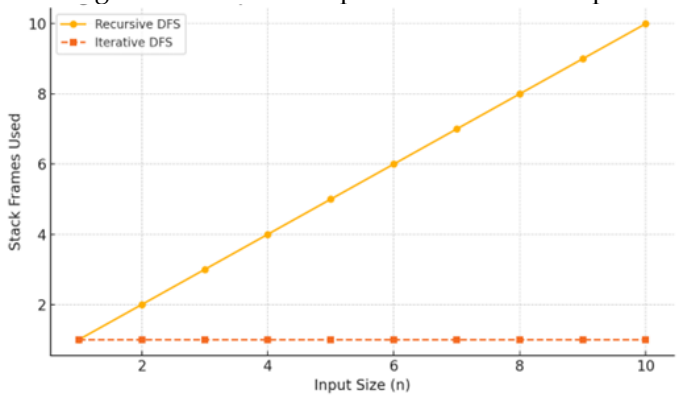


Figure 4. Stack Depth between Recursive and Iterative DFS

3. Post-Solution Reflection Prompts

The reflection prompts which focus on space can be proposed at a platform such as Codio and Khan Academy AMP that suggest products in a step-by-step method of coding that is followed once the solution is provided. Such hints stimulate the learners to be critical to their memory usages and seek alternatives.

The questions may be related to the following example: Are you able to make space usage O(n) instead of $O(n^2)$? What data structure is taking the largest contribution to the memory increase in your solution? Such prompts reinforce conceptual metacognition and align with the idea of implicit scaffolding [12] , where learners are guided toward insight without direct instruction, keeping the learning flow intact while deepening understanding.

4. Future Vision: Constraint-Oriented Learning Tools

We propose a future tool that lets students set a space budget at the start (e.g., O(log n)) and builds awareness around constraints like a canvas size in design.

Table 9. Proposed vs Existing Tool Behavior

Tool	Current Behaviour	Proposed Conceptual Behavior
GitHub Copilot	Completes code	Adds memory usage annotations with reasoning
ChatGPT	Explains time/space only if asked	Proactively surfaces spatial trade-offs
LeetCode	Reports time/space post-submission	Visualizes growth dynamically during solution
Khan Academy AMP	Validates correctness	Adds space-focused prompts post-submission

Table 10. Metric-Based vs Constraint-Based Thinking

Design Mode	Example	Cognitive Impact
Metric-Based	$O(n^2)$ calculated after code	Post hoc evaluation, low reflection
Constraint-Based	O(log n) budget set before design	Real-time trade-off awareness, structural play
Visual Tracker	Heatmap or stack graph	Builds spatial intuition and abstraction skill

5. Theoretical Framework: Why These Tools Work

These innovations are deeply rooted in learning theory:

Vygotsky's Zone of Proximal Development (ZPD): Students learn best when supported just beyond their current level; AI prompts act as real-time scaffolds. **Bruner's Scaffolding**: Visual guides and prompts serve as temporary support structures that gradually fade as students internalize conceptual knowledge. **Constructivist Learning**: Students develop understanding by actively engaging with abstract principles. Conceptual space tools make these abstractions tangible and manipulable. Visual framework showing Vygotsky → Bruner → Constructivism → AI as Bridge to Abstraction

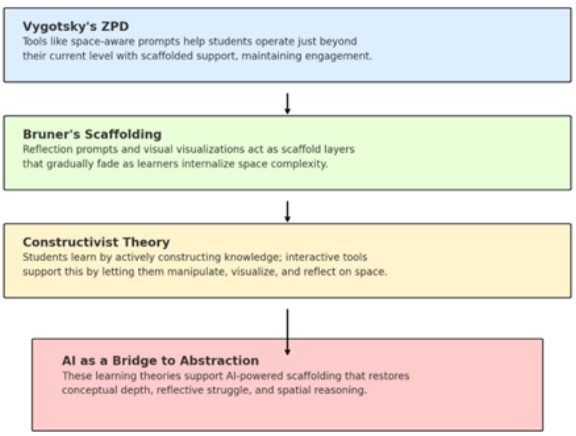


Figure 5. Theoretical Grounding of Conceptual Space Tools

3.5. From Memorization to Mechanization: Rethinking Dynamic Programming Strategy

3.5.1. From Memorization to Mechanization: Rethinking Dynamic Programming Strategy

Traditional approaches to teaching dynamic programming (DP) in computer science education often emphasize memorizing canonical problem templates—such as the "0/1 Knapsack," "Longest Common Subsequence," or "Fibonacci with memorization." Students are typically exposed to these problems through repetitive textbook exercises or online tutorials that emphasize reproducing code structure rather than fostering conceptual grasp of the underlying principles. While they may be taught the terminology of "optimal substructure" or "overlapping subproblems," they rarely develop the capacity to independently identify or derive these properties. Consequently, many students learn to match problems to pre-solved examples, leading to a surface-level understanding of algorithmic thinking [15].

Such rote-learning exercises can be adequate in ensuring short-term evaluation but cannot develop transferable skills of problem-solving. Luckily, a new chance to shape the approach to DP is presented by the introduction of AI-enhanced learning conditions. Intelligent tools accelerate the learning process through intuitive feedback mechanisms and visualization opportunities, saving time and increasing the conceptual instinct of any learner.

Dynamic programming is vertically opaque, but AI-enabled systems allow the latter to be drawn like recursion trees, memorization patterns, and the workflow in solving subproblems. The tools can be used not only to ensure that they are correct, but also to assist students in constructing solutions that they might find hard to understand at first.

For instance, Table below, emerging tools offer diverse ways to scaffold student learning:

Table 11. AI Tools Supporting Conceptual Learning in Dynamic Programming

Tool Name	Functionality	How It Changes Learning Practice
DPVis	Visualizes recursion depth, memorization hit/miss patterns, and call graphs	Makes abstract recursion concrete; students visually track how subproblems overlap
SAKSHM-AI	Uses code parsing and real-time feedback to provide step-by-step explanation	Provides AI-guided debugging and complexity insights as students write or fix solutions

These tools reshape DP learning in three fundamental ways:

From copying to constructing logic: Instead of reusing known code, students are guided to build logical structures independently. From guessing to visualizing recursion: They can observe how recursion unfolds and how memorization optimizes performance. From confusion to feedback: When errors occur, tools explain why—turning mistakes into reflective learning opportunities.

This move can be accommodated in the contemporary approaches to education, which values thinking long-term and metacognition rather than just getting the problems solved. Research indicates that AI-directed teaching can also cause greater reflection of students and the capacity to work out their own explanations [16].

However, there must be a balance between the AI assistance and the possibility of thinking on their own. Although learning may be scaffolded by these tools, they are not

to be used instead of the critical thinking skills needed in dealing with problems that have not been encountered before. Students nevertheless have to be taught to apply DP skills by being able to reason on the basis of challenges without basing everything on automation [16].

3.6. Prompting for Optimality: Greedy Algorithms in the Age of Generative AI

3.6.1. Prompting for Optimality: Greedy Algorithms in the Age of Generative AI

In traditional algorithm classrooms, greedy strategies were a test of disciplined logic: make a locally optimal choice and prove it leads to a global optimum. Students were trained to weigh trade-offs, build counterexamples, and justify correctness. Problems like Activity Selection or Huffman Coding demanded structured reasoning—not just working code.

AI tools like ChatGPT and GitHub Copilot can now generate complete greedy solutions from a single prompt. While this boosts accessibility, it risks collapsing learning into mimicry. [18] found that students often bypass deeper reasoning, treating outputs as templates rather than hypotheses to explore [18].

As an answer, programs such as AI-Lab [19] command students not only to request responses, but also to defend greedy actions, contrast methods, and criticize solutions. In particular, they ought to pose such questions like what is the reason Huffman Encoding mixes the minimal frequencies, or make the AI systems crash on counterexamples. Optimality in this case implies more than being correct in output, but also in insight into when and why the strategy is successful.

Prompting becomes a scaffolded inquiry. The table below (adapted from [21]) shows how prompt type influences cognitive depth:

Table 12. Prompt Types and Cognitive Impact in Greedy Algorithms

Prompt Type	Example	Cognitive Impact
Template	"Give me code for Fractional Knapsack"	Low
Exploratory	"Why does greedy work for Fractional but not 0/1 Knapsack?"	Medium
Diagnostic	"Show when greedy coin change fails"	High
Comparative	"Compare greedy with dynamic programming"	High

This is the core of prompt engineering: crafting queries that evoke reasoning, not just responses [21]. One risk is mistaking correctness for understanding. A greedy solution to Coin Change may work on U.S. coins but fail on (1, 3, 4)—unless prompted to test counterexamples, the flaw is hidden. It describes this learning space as a "Socratic playground," where students challenge, revise, and probe AI answers.

Students using AI tools performed consistently better [22], with tighter score distributions. While not algorithm-specific, this trend signals how AI, when scaffolded properly, can boost outcomes. This raises key questions: Can AI learn greedy strategies from prompts, or does it default to known patterns? Can prompting help AI reject greedy logic where it fails (e.g., Coin Change)? Can we teach AI to justify or critique its own strategy choices [20]? The goal is not to eliminate AI, but to use it as a cognitive partner. Prompting, done right, becomes a form of proof-building—less about syntax, more about

structure. Greedy algorithms, once a test of logic, now ask a different question: Can students still reason when the answer is instantly available?

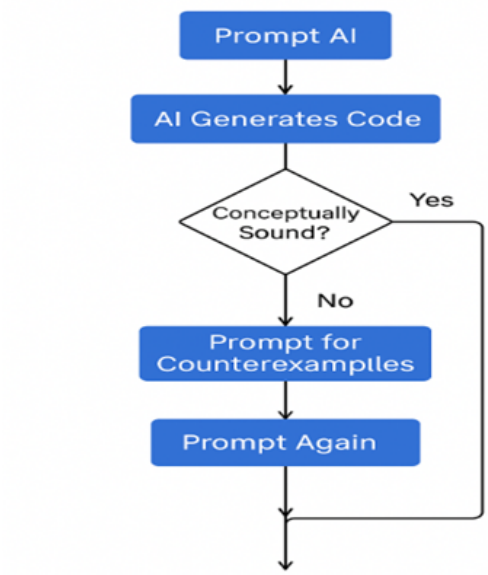


Figure 6. Prompting Loop for Greedy Algorithm Comprehension

4. Assessment And Valuation

Conventional computer science education has not historically placed much value on assessment processes. Before the advent of AI, assessment was typically based on the correctness of the code: does the code run, and does it produce the expected output? Now that students can generate complete, bug-free code using programs like ChatGPT or GitHub Copilot, these surface-level criteria are no longer sufficient. A student may turn in a fully functional recursive sorting method without writing or even understanding any of the code. This calls for a shift in how we assess: rather than focusing on what students produce, we need to pay closer attention to how they got there. Did they edit the AI-generated output? Did they explain the code? Did they use the AI-generated output as a base to debug and improve on? Did they provide a critique of the AI-generated output? Two students may submit the same result, but one student may have a greater depth of engagement with the code than the other student. If we adopt substantial reflection methods designed for students--via inline comments interpreting the logic in their code, verbal walkthroughs of their code, or even designing a set of learning activities prompting students to keep AI-generated code version one in a draft state for debugging purposes--then assessment and learning can combine to serve both a demonstration of understanding and the metacognitive attributes of curiosity, iteration, and awareness of tool use. In an assessment approach like this, the goal shifts toward evaluating growth over perfection—focusing on learning, not just the final product.

5. Conclusion And Future Work

As AI tools become more ubiquitous in the information workspace of a computer science student, educators are presented with the challenges of retaining the cognitive burdens that algorithmic problem-solving used to demand. This paper has highlighted that concepts such as divide and conquer, sorting, and algorithmic complexity need to be re-conceptualized—not abandoned—to be meaningfully educationally situated in tool-augmented contexts. We have suggested that instead of fighting AI, we should design learning experiences that promote student questioning, critiquing, and building on what

these tools create. We have suggested that future work look for structured classroom interventions that use AI as more than a means to a quick answer, and utilize it as a co-reasoning partner. How can AI tools support rather than replace thinking? What new forms of assessment or engagement become possible when students are taught to work with intelligent systems in a critical way? These questions will require us to apply pedagogical imagination and technical insight. Moving forward, as our work environments change, so too must we change how we teach and how we think about teaching algorithmic understanding.

References

1. Al-Thobhani, N.G. Visualization of recursion. *Preprint* 2023. https://doi.org/10.13140/RG.2.2.31206.91202.
2. Cormen, T.H.; Leiserson, C.E.; Rivest, R.L.; Stein, C. Introduction to Algorithms, 4th ed.; MIT Press: Cambridge, MA, USA, 2022.
3. Blelloch, G.E. Programming parallel algorithms. *Commun. ACM* 1996, *39*, 85–97. https://doi.org/10.1145/227234.227246.
4. Cetin, I.; Andrews-Larson, C. Learning sorting algorithms through visualization construction. *Comput. Sci. Educ.* 2016, *26*, 27–43. https://doi.org/10.1080/08993408.2016.1160664.
5. Baecker, R. Sorting out sorting: A case study of software visualization for teaching computer science. In *Software Visualization: Programming as a Multimedia Experience*; Stasko, J., Domingue, J., Brown, M., Price, B., Eds.; MIT Press: Cambridge, MA, USA, 1998; pp. 369–381.
6. Becker, B.A.; Denny, P.; Finnie-Ansley, J.; Luxton-Reilly, A.; Prather, J.; Santos, E.A. Programming is hard—Or at least it used to be: Educational opportunities and challenges of AI code generation. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education (SIGCSE '23)*, Toronto, ON, Canada, 15–18 March 2023; pp. 500–506. https://doi.org/10.1145/3545945.3569759.
7. Zastudil, C.; Rogalska, M.; Kapp, C.; Vaughn, J.; MacNeil, S. Generative AI in computing education: Perspectives of students and instructors. In *Proceedings of the 2023 IEEE Frontiers in Education Conference (FIE '23)*, College Station, TX, USA, 18–21 October 2023; pp. 1–9. https://doi.org/10.48550/arXiv.2308.04309.
8. Denny, P.; Luxton-Reilly, A.; Prather, J.; Stephenson, C.B. Computing education in the era of generative AI. *Commun. ACM* 2023, *67*, 56–67. https://doi.org/10.48550/arXiv.2306.02608.
9. Alanazi, M.; Alsaidi, A.; Alrasheed, M.; Alharbi, M. The influence of artificial intelligence tools on students' academic performance: A study at Saudi universities. *Computers* 2025, *14*, 185. https://doi.org/10.3390/computers14050185.
10. J. Wang and W. Fan, "The effect of ChatGPT on college students' academic performance," *Humanit. Soc. Sci. Commun.*, vol. 12, Art. no. 621, 2025, https://doi.org/10.1057/s41599-025-04787-y.
11. García Piqueras, M.; Hernández-Orallo, J. Measuring social intelligence in artificial agents. In *Machine Learning and Knowledge Discovery in Databases: Research Track. ECML PKDD 2021*; Oliver, N., et al., Eds.; Springer: Cham, Switzerland, 2021; Volume 12975, pp. 667–683. https://doi.org/10.1007/978-3-030-86486-6_41.
12. Podolefsky, N.S.; Moore, E.B.; Perkins, K.K. Implicit scaffolding in interactive simulations: Design strategies to support learning. *arXiv* 2013, preprint. Available online: https://arxiv.org/abs/1306.6544 (accessed on 18 August 2025).
13. Ghent, K.N. *An Introductory Survey of Computational Space Complexity*. Bachelor's Thesis, Trinity University, San Antonio, TX, USA, 2020.
14. Lee, S.; Prasad, A.; Vuong, R.D.-C.; Wang, T.; Han, E.; Kempe, D. DPVis: A visual and interactive learning tool for dynamic programming. In *Proceedings of the ACM Technical Symposium on Computer Science Education (SIGCSE TS '25)*; to be published. Available online: https://doi.org/10.48550/arXiv.2411.07705 (accessed on 18 August 2025).
15. Lee, S.; Prasad, A.; Vuong, R.D.-C.; Wang, T.; Han, E.; Kempe, D. Supporting dynamic programming: An interactive tool for learning. *arXiv* 2024, unpublished. Available online: https://arxiv.org/abs/2411.07705 (accessed on 18 August 2025).
16. Pólya, G. *How to Solve It: A New Aspect of Mathematical Method*, 2nd ed.; Princeton University Press: Princeton, NJ, USA, 1957.
17. Naps, T.L.; Rößling, G.; Almstrum, V.; Dann, W.; Fleischer, R.; Hundhausen, C.; Korhonen, A.; Malmi, L.; McNally, M.; Rodger, S.; Velázquez-Iturbide, J.Á. Exploring the role of visualization and engagement in computer science education. *ACM SIGCSE Bull.* 2002, *35*, 131–152. https://doi.org/10.1145/782941.782998.
18. Dickey, E.; Bejarano, A.; Garg, C. Innovating computer programming pedagogy: The AI-Lab framework for generative AI adoption. *arXiv* 2023, preprint. Available online: https://doi.org/10.48550/arXiv.2308.12258.
19. Dickey, E.; Bejarano, A.; Kuperus, R.; Fagundes, B. Evaluating the AI-Lab intervention: Impact on student perception and use of generative AI in early undergraduate computer science courses. *arXiv* 2023, preprint. Available online: https://doi.org/10.48550/arXiv.2505.00100.
20. Tetteh, J.; Zielosko, B. Greedy algorithm for solving the multi-dimensional knapsack problem: A case study. *Entropy* 2025, *27*, 35. https://doi.org/10.3390/e27010035.
21. Liu, H.; Li, C.; Wu, Q.; Lee, Y.J. LLaVA: Large language and vision assistant. *arXiv* 2023, preprint. Available online: https://doi.org/10.48550/arXiv.2304.08485 (accessed on 18 August 2025).
22. Qin, S.; Zhao, M. Artificial intelligence in education: Research overview and teaching practice evaluation. *Educ. Inf. Technol.* 2023, *28*, 1075–1090. https://www.preprints.org/manuscript/202311.0106/v1.